Programming the PET/CBM

Programming the PET/CBM

Covers most software aspects of
the COMMODORE PET and CBM range
of microcomputers and peripherals

TOPICS INCLUDE:

Complete reference sections on
BASIC and 6502 machine language
(introductory and advanced level)
/Tape, disks, printers, plotters/
Memory, storage, syntax, routines
/Efficient business systems
/Reference sections on each ROM

by RAETO WEST

a LEVEL LIMITED publication

Raeto West

Programming the PET/CBM

Raeto West

Contents

TABLE OF CONTENTS

CHAPTER 10: THE TRANSITION TO MACHINE-CODE

CHAPTER 11: PROGRAMMING THE 6502 MICROPROCESSOR

CHAPTER 12: ALPHABETIC REFERENCE TO 6502 OPCODES

CHAPTER 13: ROM ROUTINES AND THEIR USES

CHAPTER 14: EFFECTIVE 6502 PROGRAMMING

CHAPTER 1: INTRODUCTION AND OVERVIEW

1.1 Introduction and plan of the book

The purposes of this book are to teach competent programming and provide a comprehensive reference text on the PET/CBM range of microcomputers. These aims are not entirely compatible: virtually everyone interested in these machines begins with BASIC and progresses to machine-code, but, on the other hand, for completeness it is often necessary to mix both types of program. Comparative beginners will therefore find themselves skipping quite large sections of temporarily difficult text. I have included demonstration routines in BASIC (Chapter 5), 6502 machine-code (Chapter 12), and disk, tape, and printer programming (Chapters 7 and 8). To reduce the chance of mis-keying, these routines have been kept as short as possible; in this way it is possible to learn by doing, by experimenting at the keyboard to get the feel of the commands, without the tedium associated with entering long illustrative programs.

Commodore's most recent machines, the VIC home computer and the MMF 'Micro-mainframe' are not dealt with here, partly for reasons of space. VIC has many things in common with CBM microcomputers, MMF rather fewer. My rule has been to try to cover most of the common configurations of hardware which exist at present and are likely to exist in the fairly near future. For this reason little space has been given to modems, hard disks ('Winchesters') and networks, while tape and diskettes are explained in depth. I've documented each of the three versions of CBM BASIC issued to date, although with a bias to the later versions. This may seem rather wasteful - until questions of compatibility between ROMs arise.

1.2 Conventions

Most CBM machines switch on in upper-case/ graphics mode, and except in few cases, mainly 8032 disk commands, BASIC is printed in upper-case characters here, which also distinguishes BASIC keywords from the normal text. BASIC can of course appear in lower-case on the VDU, if the mode is changed, a fact which may cause confusion to programmers unused to this dual display. Machine-code and BASIC, entered from the keyboard in the usual way, use mostly unshifted keys.

CBM BASIC has special screen-editing commands, which appear within quotes as reversed characters. (See Chapter 2). For increased readability I have printed these in square brackets - [HOME], [CLEAR], and so on. Chapter 13 has a LIST routine to perform this task automatically for BASIC.

The only other non-standard notation is the use - for machine-code only! - of round brackets as a shorthand for a 2-byte indirect address. For example, I have written (2A) to denote the two-byte number held in locations 2A and 2B, taking the first byte as low and the second as high, in accordance with 6502 logic. Similarly, (FFFE) is a convenient way to refer to the interrupt address, held in FFFE and FFFF.

Spelling of computer terms is more-or-less American. Occasionally BASIC terms are written in lower-case, when used in a general sense, not specifically BASIC. For example, 'printing to screen' can use PRINT or some machine-code equivalent, and 'peeking' could mean PEEK or a machine-code command like LDA.

1.3 Sources of information

Manuals CBM's product manuals are widely recognized to be unhelpful; this is one of the reasons for the existence of this book. MOS Technology (now a part of the Commodore Semiconductor Group) produces reasonable manuals on 65xx series hardware and 65xx programming.

Magazines, journals In the U.K. the largest-selling small computer journals are Practical Computing and Personal Computer World. These are not particularly CBM-orientated. Printout was, but is no longer, exclusively about the CBM. Compute! deals with 6502 machines (Apple, Atari, PET/CBM) and is the best magazine for the non-beginner. Micro has machine-code articles on the 6502 and 6809. Byte magazine and kilobaud-Microcomputing are two other well-known general microcomputer publications; other market niches are covered by (for example) Creative Computing and Dr Dobbs' Journal. All but the first three of these magazines are American. There are also periodicals aimed at the education market, the home computer and games market, the technical hardware market, and what might be called the uninformed businessmen's market.

There are four weekly 'throwaways' in the U.K. at the time of writing (Computing, Computer Weekly, Datalink and Computer Talk) of which Datalink is most interested in microcomputing.

User groups and newsletters Commodore in Canada produces 'The Transactor', which is useful and informative. The U.S Commodore Newsletter (called 'Interface') is less good. The U.K. equivalent was called the PET Users Club Newsletter, later abbreviated to CPUCN, and renamed 'Commodore Club News' in mid-1981. Like all periodicals, it is episodic and fragmentary (I have lost count of the number of reviews of word processor packages). However, it is responsive to its readers' requests.

User groups are the best source of up-to-date information. IPUG ('Independent PET Users Group') has many branches in the U.K. and many experienced software and hardware people. Other groups include SUPA ('Southern Users of PETs Association') and the Association of London Computer Clubs, a loose organization of groups which meet in polytechnics, universities and community centres, and is not specifically CBM. **Books*and other publications** Osborne/McGraw-Hill's 'PET/CBM Personal Computer Guide' is issued with PETs sold in the U.S. It is currently in its third edition, edited by Jim Strasma. This omits machine-code, which is covered in a number of books, of which a few are explicitly PET: 'Hitch-Hikers Guide to the PET' for example. Some books appear to be available only in the U.S., for example Gregory Yob's 'PET User Manual'. Nick Hampshire has written three (of a projected ten) books for Commodore U.K., including 'Library of PET Subroutines' and 'PET Graphics'. 'The PET Revealed deals mainly with hardware and the BASIC 1 PET; other hardware books are listed at the end of Chapter 8.

Several compendium-type books exist, for example by IPUG, by CPUCN, and by Printout. The 'Channel Data Book' is an American compilation of PET/CBM products and packages. The 'Computerist's Guide' is an indexed survey of the contents of most of the microcomputer magazines, arranged by topic. Commodore produce a 'Software Encyclopedia', essentially an uncritical list of every type of software package.

1.4 Acknowledgements

I have gone to some lengths to test and check the information in this book, and in fact believe it to be more reliable than most on this subject. Nevertheless there are certain to be errors, and I apologize for any inconvenience or puzzlement which may be caused. The usual disclaimer applies: I cannot accept responsiblity for failures in software or hardware which may be based on suggestions found in this book.

There are many company names, trade marks, and business names mentioned in the book; CBM ('Commodore Business Machines'), MMF ('Micro-Mainframe'), PET ('Personal Electronic Transactor') and VIC ('Video Interface Chip') are all trade marks of Commodore Business Machines. PET/CBM is a general way of referring to Commodore's microcomputers with both keyboard and screen, and equipped with Microsoft BASIC.

Charles ('Chuck') Peddle, the designer of the PET/CBM and also, apparently, the 6502 chip, deserves a special mention at this point, although his path has diverged considerably from Commodore's.

---

*There are many general books on computers. Chris Evans wrote popular books on the supposed impact of microprocessors. The technical side of chips was dealt with (e.g.) in 'Scientific American'. Critics of applications include Joseph Weizenbaum, a Professor at M.I.T. Gerry Weinberg is well-known (e.g. 'The Psychology of Computer Programming'), taking a conventional, optimistic viewpoint. Philip Kraft on the other hand has examined de-skilling by management, and women's status within the industry. (Sartorial iconographers might note that Weinberg is always depicted bearded and pullovered, but Kraft neatly-suited). Some journalists have drawn attention to the role of cheap labour in the Far East in chip manufacture. Academic computing's domination by software theoreticians has been attacked by only one hardware-based writer that I know of, Ivor Catt, who called programmers 'updated clerks'. (See e.g. 'Computer Worship).

1.5 PET/CBM hardware and family tree

APPROXIMATE CHRONOLOGY OF COMMODORE MICROCOMPUTERS

1975	65xx chips: by Rockwell, MOS Technology	
1977	KIM: single-board 6502-based microcomputer	
1978	2001-8: 8K RAM, built-in cassette, 8" 40-column white screen, small keyboard. BASIC 1. (ROM -19, issued to replace -11, cures screen edit bug).	
1979		
	2001-16, 2001-32: 16K or 32K RAM, 8" 40-column green screen, large keyboard, no cassette. BASIC 2 ('Upgrade ROM') including monitor. Later renamed 3008, 3016, 3032 with **'BASIC 3'.**	2000 series printers. 2040 disk drives (DOS 1, sequential files only). 3040 disk drives (DOS 1.2, Shugart).
1980		
	4008, 4016, 4032: 8" 40-column green screen, large keyboard, similar to previous except for BASIC 4.	4040 disk drives (DOS 2.1, including relative files).
	8032 32K RAM, 12" 80-column green screen, extra keys, beeper. BASIC 4 (includes CBM disk commands). (ROM -23, issued to replace ROM	8050 disk drives (DOS 2.5, Micropolis).
1981	-19, cures bug in DS$).	4022 printer (=MX-70).
	4008, 4016, 4032: Made with 12" 40-column green screen only, with extra keys, beeper.	8250 disk drives (DOS 2.7, Tandon).
	VIC 22 columns, color with external TV, sounds.	
	MMF 64K extra RAM in 16 switchable blocks from $9000-9FFF, 6502/6809, RS232 and high-speed	
1982	RS232, many languages, existing and under development at Waterloo Universtity).	
	? BASIC 5 with BCD arithmetic? 40-column VIC, discontinued 40-column CBM? Color CBM?	

The table summarises most of the hardware developments of Commodore to date. I have omitted some of the printers. See Chapter 2 for more information on the differences between BASIC ROMs, which are also mentioned in passing throughout much of the book. Chapter 6 deals with disk drives, and Chapter 7 with the commands introduced in BASIC 4.0. Printers and other hardware are explained in Chapter 8. A significant difference between 12" and 8" models is the CRT controller chip: see Chapter 9 on this, which also covers the built-in 'beeper'.

Internal layouts   The diagram is a rough guide to the layout of the main chips and ports on the printed circuit boards of the early PET, the 8" screen 3000 and 4000 series CBMs, and the 12" screen 4000 and 8000 series CBMs.
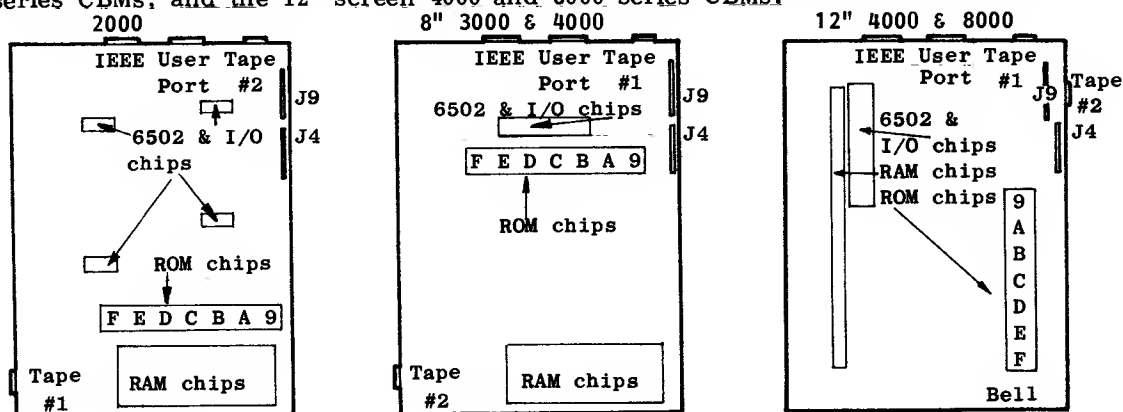
CHAPTER 2:   BASIC AND HOW IT WORKS

2.1   Keyboard, screen and screen-editing   The keyboard and screen are described in detail in Chapters 8 and 9 respectively. These devices offer the most direct communication with the machine. The keyboard is decoded by a 6520 chip and ROM software; the screen memory is organised in a straightforward memory-mapped way, in which sequential RAM locations correspond to screen positions moving left to right and down. Screens in the CBM have 25 rows. 40-column and 80 column screens therefore require 1000 and 2000 RAM locations respectively. The screen starts at location $8000 in each case, exactly half-way in the memory-map. The entire 4K from $8000 - $8FFF is allocated to the screen, and the address-lines connected so that the upper part of this block duplicates the lower. (So $8000-$83FF and $8400-$87FF are not distinguished from each other in 40-column machines, for example, and a poke or peek to $8000 has the same effect as a poke or peek to $8400). A few bytes are left over in RAM which do not appear on the screen: 24 in 40 column machines, 48 in 80-column, because 1024-1000 = 24 and 2048 - 2000 = 48. Tables of hexadecimal and decimal values of screen locations are printed in Chapter 9. It is worth memorizing the figure 32768 (=$8000), which is the location of the top-left of screen. Try POKE 32768,33 for instance.

     Screen editing is the process by which characters on the screen are altered and moved from the keyboard. PET/CBM has a number of special keys for this purpose, which are fairly self-explanatory. The main complication is the use of the quote (") to hold screen-editing characters in storage in BASIC. When this is done, the character appears as a meaningless graphics symbol, and is printed in the usual consecutive sequence without having its usual effect, such as clearing the screen. The exception to this exception occurs with a few keys, like 'Delete', which have to work both in quotes and out; the resulting editing system has a few anomalies, which make it less easy than might be the case to perform editing tasks. However, it is still noticeably easier than some rival systems. Commodore's manuals and some books go into great detail on this; it is much more easily explained by demonstration and trial than by the written word. Try the examples which follow if you are uncertain about screen editing; without covering every possible aspect, they incorporate most features.

     (i) Editing a line without quotes. Switch on the machine, so Commodore's BASIC message appears. Press [HOME].* The message may be edited, by (say) moving the cursor right several positions, then inserting spaces. The end of the line moves right; eventually, when it is 80 characters long (88 with VIC!) it will not expand more.

     (ii) Using quotes. Type PRINT " and a series of miscellaneous keys including editing characters. The effect of [RVS], [RVSOFF], [HOME], [CURSOR DOWN], and the rest can be explored in this way. On pressing Return, the line is processed and printed. With practice it is easy to produce quite complicated layouts; PRINT "[HOME]* [DOWN]*[DOWN]*" prints three asterisks diagonally from the top left of the screen.

     (iii) Editing a line with quotes. Type 1 PRINT "BASIC" so the cursor now is positioned after the second quote, and quotes mode is off. Backspace the cursor one position, and type several [INSERT] characters; the second quotation mark will move right. Now type the [DELETE] key several times. Delete characters, appearing as reversed Ts, fill the space. Press Return, the type RUN Return, to see the effect of these characters. LIST will redisplay the line.

     (iv) Shift-Return and the ESCape key. Return moves the cursor to the next line and causes the edited line to be processed - i.e. incorporated into BASIC or executed in direct mode. Shift-Return moves the cursor without causing processing. The ESCape key (12-inch screen machines only) has an analogous effect from within quotes, turning off the quotes mode and the reverse mode, so the effect is identical to that obtained from Shift-Return combined with cursor moves back to the original line.

     (v) BASIC editing. LIST displays a line, or range of lines, from BASIC. Any line may be edited in any way; for example, if the linenumber is changed and Return pressed, a duplicate line is produced within the program. An isolated number erases the corresponding BASIC line, if there is one.

---

*In most of this book I have conventionally represented the special characters by a name in capitals within square brackets. (Chapter 13 has a routine which lists programs in this way). This is far more readable than a single graphics character which is its equivalent.

LIST appearance of CBM special characters:

Mode	⇓	⇑	⇐	⇒	RVS	RVSO	HOME	CLR	TAB	DEL	INS
Lower-case	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
Upper-case	▨	▢	▨	▨	▨	▪	▨	▢	▨	▨	▨

2.2 Entry and storage of BASIC

BASIC can run in either of two modes: direct (or immediate or 'calculator' mode) or program ('stored') mode. If a line begins with a number, it is treated as a program line, and stored in memory with other program lines until it is run. If a line does not start with a number it is executed immediately Return is pressed. The principles on which immediate mode runs are identical to those which apply to stored programs; in this and the following sections we shall therefore mainly consider BASIC programs.

A program may be examined with the LIST command. However, this provides no clue to the way BASIC is organised in RAM, since it involves an elaborate process of decoding. (See Chapter 13). We can look at BASIC *in situ* using either BASIC or the machine-code monitor in these ways:

(i) X=0: FOR J=1025 TO 1200: POKE 32768+X, PEEK(J): X=X+1: NEXT is a simple direct mode line which displays several hundred bytes of BASIC at the top of the screen. (Lower-case mode - POKE 59468,14 - gives the clearest representation). The bytes are not easily deciphered, although text (in quotes) is clear enough. In the two examples, the first has more text (including REMs) than the second.

```
⊒▨ ROM4 RELOCATING ′PRINT USING′ ROUTINE "▁T ⊤ |
▨▨▨▨ 59468,12: ▮ "_____" :▮ UNDERLINE (SHIFT-$) TIDIES TITLE▨▨▨▨ "▨▨▨▨▨
▮ "machine code is now loaded into sys 826 and sys 881.H "▨load the program to
1022,128!▨"▨▨▨DUM 3.4▨▨⊤▨"RICK LEONN ▪"▨▨▨PROMINICO LTD.▨′▨▨▮"VANCOUVER
7"▨▨▨▨▨▨▨▨▨▨▨▨ 15,8,15▨▨▨R$⊤|(13)▨ ▨$▨5$⊤"_____⊥L6$⊤"_____▨|▨8▨7$⊤"
____|BL8$⊤"_____▨ ▨▨9$⊤"_____▪VL0$⊤"_____▨▨ ▨1$⊤"_____✝
```

(ii) After entering the monitor (SYS 4 is easiest) we can display bytes in hexadecimal form from $0400 onwards. With the program 10 PRINT"HELLO" in memory, we get this:

```
.: 0400 00 (0E 04)(0A 00) 99 22 48 ←—∅4∅E = Next line's link pointer (∅ here).
.: 0408 45 4C 4C 4F 22 (00)(00 00) ←—∅∅∅A = BASIC linenumber (1∅ here).
.: 0410 AA AA AA AA AA AA AA AA      ∅∅ = End-of-line ; ∅∅∅∅ = End of program.
```

The ASCII characters for "HELLO" are visible in there, but so is much else.

The table on the following page gives a complete breakdown of the storage of lines of BASIC, excluding the linenumbers and connecting details. All the components are stored in ways which exclude ambiguous interpretation. Literals are held within quotes, or after REM or DATA, and are not treated like the remaining BASIC. Numbers, as in GOTO 1000 or X=89.8, are also held as ASCII strings, so that the 1000 of GOTO 1000 occupies 4 bytes. Punctuation (commas, colons, semicolons, but not full stops which are used as decimal points) is held as single bytes; so are the special BASIC characters of %, $, (, and). Variable names use alphanumeric characters; the initial is always alphabetic, to avoid confusion with numerals. Finally, the keywords themselves are held in compact form, as single bytes; see the table. These are called 'tokens'. Slightly confusingly, single-byte keywords like <,=,* and / are also tokenised into alternative single bytes. Tokens always exceed 127; the high bit of the byte is on, and this enables machine-code to immediately recognise a token. This feature is common to Microsoft BASICs.

A BASIC program is a 'linked list' or 'chain' of individual program lines. Unless specially modified, BASIC starts at $0400 with a zero byte and is held in consecutive locations up in memory. Each line starts with a 2-byte link address, which is an absolute address pointer to the link address starting the next line. This is followed by the linenumber, also in 2 bytes. In each case the low byte is first. Each line is terminated by a zero byte, and in addition 2 more zero bytes mark the end of the program, so a link address of zero denotes the end. As we shall see, BASIC is supported by a set of pointers which monitor important features as a program runs. If these are modified, various non-standard effects can be realized.

The link addresses, linenumbers, tokens and so on can be identified with practice quite easily; the one-line program above has had its marked to show how they are arranged. Again, this is standard Microsoft, as is the use of the zero byte to mark the end of a line. (It is not universal; Apple Integer BASIC uses 1 to mark ends of lines, and has an offset pointer, with maximum 255, to the next line).

PET/CBM INTERNAL STORAGE OF BASIC

32	20	sp	64	40	@	128	80	END	160	A0	CLOSE	192	C0	TAN
33	21	!	65	41	A	129	81	FOR	161	A1	GET	193	C1	ATN
34	22	"	66	42	B	130	82	NEXT	162	A2	NEW	194	C2	PEEK
35	23	#	67	43	C	131	83	DATA	163	A3	TAB(	195	C3	LEN
36	24	$	68	44	D	132	84	INPUT#	164	A4	TO	196	C4	STR$
37	25	%	69	45	E	133	85	INPUT	165	A5	FN	197	C5	VAL
38	26	&	70	46	F	134	86	DIM	166	A6	SPC(	198	C6	ASC
39	27	'	71	47	G	135	87	READ	167	A7	THEN	199	C7	CHR$
40	28	(	72	48	H	136	88	LET	168	A8	NOT	200	C8	LEFT$
41	29	)	73	49	I	137	89	GOTO	169	A9	STEP	201	C9	RIGHT$
42	2A	*	74	4A	J	138	8A	RUN	170	AA	+	202	CA	MID$
43	2B	+	75	4B	K	139	8B	IF	171	AB	-	203	CB	GO*
44	2C	,	76	4C	L	140	8C	RESTORE	172	AC	*	204	CC	CONCAT*
45	2D	-	77	4D	M	141	8D	GOSUB	173	AD	/	205	CD	DOPEN
46	2E	.	78	4E	N	142	8E	RETURN	174	AE		206	CE	DCLOSE
47	2F	/	79	4F	O	143	8F	REM	175	AF	AND	207	CF	RECORD
48	30	0	80	50	P	144	90	STOP	176	B0	OR	208	D0	HEADER
49	31	1	81	51	Q	145	91	ON	177	B1	>	209	D1	COLLECT
50	32	2	82	52	R	146	92	WAIT	178	B2	=	210	D2	BACKUP
51	33	3	83	53	S	147	93	LOAD	179	B3	<	211	D3	COPY
52	34	4	84	54	T	148	94	SAVE	180	B4	SGN	212	D4	APPEND
53	35	5	85	55	U	149	95	VERIFY	181	B5	INT	213	D5	DSAVE
54	36	6	86	56	V	150	96	DEF	182	B6	ABS	214	D6	DLOAD
55	37	7	87	57	W	151	97	POKE	183	B7	USR	215	D7	CATALOG
56	38	8	88	58	X	152	98	PRINT#	184	B8	FRE	216	D8	RENAME
57	39	9	89	59	Y	153	99	PRINT	185	B9	POS	217	D9	SCRATCH
58	3A	:	90	5A	Z	154	9A	CONT	186	BA	SQR	218	DA	DIRECTORY
59	3B	;	91	5B	[	155	9B	LIST	187	BB	RND	219	DB	
60	3C	<	92	5C	\	156	9C	CLR	188	BC	LOG	220	DC	---See
61	3D	=	93	5D	]	157	9D	CMD	189	BD	EXP	221	DD	Notes---
62	3E	>	94	5E	↑	158	9E	SYS	190	BE	COS	222	DE	
63	3F	?	95	5F	←	159	9F	OPEN	191	BF	SIN	223	DF	

Notes: (i) Valid BASIC bytes from 0-127, in bold type, are space, " # $ % ( ) , and . in order, followed by 0-9, : ; and A - Z. The zero byte is valid as an and-of-line and end-of-program marker. On LIST, bytes from 96-127 appear as duplicates of the characters 32-63, but, like the italicised characters above, cause ?SYNTAX ERROR.

    *(ii) Valid bytes from 128 - 255 are BASIC tokens; and GO is omitted from BASIC 1, while CONCAT and the following keywords are omitted from BASIC<4. Bytes beyond the end of the table list as apparent duplicates of keywords in BASIC<4, and as error messages and garbage in BASIC 4. Note that Shift-K (BASIC 1), Shift-L (BASIC 2), and Shift-[(BASIC 4 - may not be on the keyboard!), all cause LIST to stop with ?SYNTAX ERROR. Spurious keywords can LIST but will not run.

    (iii) The quotation mark, CHR$(34), can of course legitimately precede any character.

When a BASIC program is entered at the keyboard, the contents of the line in which Return is pressed are transferred to a buffer. This is 80 characters long, and can hold one line; BASIC 1's buffer was in the zero-page ($0A - $5A), but later BASIC versions moved it to $0200 - $0250. After the line has been moved, it is scanned for keywords ; any that are found are converted into tokens. The tokenised line is then merged into the program in memory, its position determined by its linenumber. The tokenisation process can be watched (see Chapter 13) with the aid of a machine-code routine which displays the input buffer at the top of the screen. In direct mode, the line is executed in the input buffer; this enables a line like PRINT "[CLEAR]HELLO" to run from the start to the end, even though it is erased from the screen as it runs. 40-column BASIC has provision in it to distinguish 40-character lines from 80-character lines; a screen-line table of 25 bytes holds a value for each line to indicate whether two lines have been conceptually connected by the screen editor. Note also that short forms of keywords are acceptable. These are listed in Chapter 5. They provide a way

to enter lines which otherwise might be overlength. Provided that the line doesn't exceed 80 characters, this is acceptable, although when LISTed the same line will be hard to edit, since it will overflow the end of the 80-character line. The order of the keywords in the table determines whether an abbreviation is possible; if there is any ambiguity, the interpreter picks the first in the table. So E shift-N enters END, and F shift-O enters FOR; but R shift-E is READ, RESTORE needing RE shift-A. INPUT# can be entered as I shift-N, but INPUT cannot be abbreviated by this method. PRINT is only available in a short form because '?' is specially written in to the interpreter.

2.3 Variables, variable storage, and pointers

A 'variable' is an algebraic idea: a symbol stands for a quantity (or string of characters). Microsoft BASICs have three variable types: numeric, integer, and string. The interpreter distinguishes between them by testing for a character after the alphanumeric characters which make up the name. '\$' and '%' represent string and integer variables respectively. If there is no special character, the variable is numeric or 'real'. The presence of '(' denotes that the variable is subscripted. CBM BASIC allows multi-dimensioned arrays; the individual arguments are separated by commas. Three array types exist, distinguished by the same type declarators as simple variables.

Interconversion between variable types is automatic as far as numerals are concerned; string-to-numeric conversion and vice versa requires special functions. For example, L%=L/256 automatically rounds L/256, and checks that the result is in the signed, 2-byte range (-32768 to 32767) to which CBM integers are confined. And L\$=STR\$(L) and L=VAL(L\$) or L%=VAL(L\$) convert numerals to strings and vice-versa, subject to certain rules (see Chapter 5). Two other interconversion functions are CHR\$ and ASC , which operate on single bytes and enable expressions which would otherwise be treated as special cases to be processed. Q\$=CHR\$(34) assigns the quote to variable Q\$; and 10 GET X\$: IF X\$="" GOTO 10 / 20 IF ASC(X\$) = 13 GOTO 100 / ETC. tests for Return, which is only possible with the aid of these byte-level commands.

Variables' names are subject to these rules:
1. The first character must be alphabetic.
2. The next character may be alphanumeric.
3. Any further alphanumerics are valid, but not considered part of the name.
4. The next character may be % or \$, denoting integer or string respectively.
5. The next character may be (, denoting a subscripted variable.
6. A name cannot include reserved words, as the translater will treat them as keywords and tokenise them. Note that reserved variables (TI, ST, DS, DS\$) can be incorporated in names, as they are not keywords.

All these rules simply have the purpose of removing ambiguity and making storage convenient and fast. If (say) 1A were a valid variable name, 100 1A=1 would require special syntactical treatment to distinguish it from 1001 A=1. And if other symbols than alphanumerics were permitted, so that B= were a valid name for instance, again this could cause problems. We shall see very shortly why names of length 2 are used.

The next page has a table of names; some are valid, others are not. Italicised text indicates the presence of a keyword, making the name unacceptable. All those names without italics are perfectly usable; but care has to be taken to avoid using what is in fact one variable under the impression that it is two or more; for example, NUMBER and NUMERAL are legitimate variables, but both could be replaced by NU, and a program which 'thinks' they are different will give surprising results.

Even with valid names, some ambiguity is possible, particularly if a program is 'crunched' so that all spaces are removed (except in quotes). The next section has examples.

Variables, in either direct mode or program mode, are stored after the program currently in memory; the space is known to be there, and as a program runs variables are created and modified in this area. Strings,because of their dynamic nature, do not fit tidily into this scheme, and are stored in two parts, a name with a pointer, and the string pointed to; with most variables' manipulations involving strings, RAM has to be checked to ensure there is room to store the next string. Chapter 5, in DIM and FRE and elsewhere, discusses storage. Before looking at the system of pointers, let's examine the RAM storage of each type of variable. These can be peeked in exactly the same ways that BASIC programs can be. There is a complication that the actual values stored may vary; a BASIC program peeking values which follow itself may produce different results at different times. Provided we avoid minor confusions of this sort we can investigate the way in which BASIC variables are stored.

EXAMPLES OF LONG NAMES FOR VARIABLES

ADD	DOLLAR	LIMIT	PENCE	*TOP*
AGE	*END*	LINES	PERCENT	*TO*TAL
AMOUNT	ESCAPE	*LOAD*	PIA	*TO*WN
ANSWER	ESTIMATE	LOCATI*ON*	PLACE	TRACK
ARRAY	E*VAL*UATI*ON* LOW		*POSITION*	TYPE
AVAILABLE	EXTENT	LOWER	POUND	UNDER
AVERAGE	FILE	MACHINE	PRICE	UNIT
BAD	FINAL	MARGIN	PRIMARY	UPPER
BEST	FINISH	MARK	*PRINT*	*VAL*UE
BETTER	FIRST	MARKUP	PRODUCT	VARIABLE
BIT	FLASH	MASS	PROFIT	VARIATI*ON*
BLOCK	*FOR*M	MEAN	QUANTITY	VARIETY
BRANCH	*FOR*MULA	MEASURE	RATE	VERTICAL
BYTE	*FOR*WARD	METER	RECO*R*D	VIA
CALCULATI*ON*	FOUND	METRE	REFERENCE	WAGE
CAL*END*AR	FRACTI*ON*	MINUTE	RE*OR*DER	WEIGHT
CANCEL	FUNCTI*ON*	MO*NE*Y	REVERSE	W*OR*D
CATA	*GOO*D	MO*N*TH	RIGHT	W*OR*ST
CENTER	GUESS	NEVER	ROOT	YEAR
CENTRE	HEX	*NEW*	ROUNDING	
CODE	HORIZO*N*TAL	*NOT*E	SALARY	
COMM*AND*	HOUR	NOW	SALES	
COMMENT	IEEE	NUMBER	SEARCH	
CO*NT*ENTS	IN	NUMERAL	SECO*ND*	
CO*N*TROL	INCOME	NUMERA*TOR*	SECO*N*DARY	
CO*R*RECT	INDEX	OFF	SECT*OR*	
C*OST*	*INPUT*	OK	SKIP	
DATA	*INT*EGER	OLD	SOLUTI*ON*	
DATE	*INT*EREST	*ON*	*STANDA*RD	
DAY	INVENT*ORY*	*OR*DER	S*T*ART	
DECIMAL	INVESTMENT	OUT	S*TA*TEMENT	
*DEFA*ULT	INVOICE	OUTPUT	S*TOC*KS	
DENOMINAT*OR*	ITEM	OVER	S*T*RING	
DERIVATIVE	KILO	PACK	SUBSTITUTE	
DEVIATI*ON*	LAB*OR*	PAGE	SUB*TOTA*L	
DIAMETER	LABOUR	PARAMETER	SUM	
D*I*FFERENCE	LAST	PARTS	SURPLUS	
DIVIDE	LEFT	PAUSE	TABULATE	
DISCOUNT	LE*N*GTH		*TI*ME	
			*TI*TLE	

Simple variables Every non-array variable occupies 7 bytes of RAM following its program, or, in direct mode with no stored program, in BASIC's RAM space starting at $0401. In addition, strings occupy the top of RAM. BASIC 4 strings are stored with a 2-byte pointer back to their names. Of the 7 bytes, the first two hold the name. The high bit of each may be set or unset, giving 4 permutations of effectively the same name; in this way, the variables A, A%, A$, and FN A are distinguished by the interpreter. At run time, an expression like A=4 causes the entire table of variables to be searched, if A is not present, and A to be set up at the end of the current table. For this reason, BASIC may be noticeably faster if variables are defined in order of importance. Note that all four types of variable are stored together; there is no separation of strings from real numbers, for example. Note also that arrays are stored after the simple variables; their range is defined by an extra pointer. This is necessary because arrays would slow variables' search times by spoiling the consistency with which 7 can be added to each simple variable's pointer to find the next. At any rate, this is standard Microsoft. Consequently, new variables, defined after arrays, cause the entire array structure to be moved 7 bytes up RAM, which may generate strange delays, and is a further reason to define variables at a program's start. The storage system is rather wasteful: 3 bytes are unused with integer-type variables, 2 with strings, and 1 with function definitions.

Subscripted variables These are segregated from simple variables, and constructed differently: each array has an offset pointer to the next array, since obviously all arrays are not the same length. Microsoft's system saves space compared with simple variables: integer arrays, in particular, are very efficient in space usage. It also avoids the possibility of confusion between simple variables and arrays, which otherwise could arise.

Storage of CBM variables

Variable type:	Name:		Details of storage:				
				MANTISSA			
Floating-point	ASCII	ASCII or 0	EXPONENT	M1	M2	M3	M4

Sign bit

Integer	ASC+128	ASC+128 or 128	HI BYTE	LO BYTE	0	0	0

Sign bit

				POINTER			
String	ASCII	ASC+128 or 128	LENGTH	LO BYTE	HI BYTE	0	0

			POINTER TO DEF'N		POINTER TO VARIABLE		INITIAL
Function def'n	ASC+128	ASCII or 0	LO BYTE	HI BYTE	LO BYTE	HI BYTE	OF VAR.

The table shows all four types of simple variable. The name carries an implicit type declaration; thus a name consisting of the values 71 and 199 (decimal) is GG$, and a name consisting of 65 and 0 is A. Taking these in turn, note that a floating-point number's value is

$$\text{SIGN} * 2^{\text{EXP}-129} * (1 + \frac{M1}{128} + \frac{M2}{128*256} + \dots)$$

which can be expressed in various ways. (See e.g. Chapter 16, and Chapter 5 on VARPTR). This is a standard floating-point format. Integers are held in signed, 2-byte form, with range -32768 to 32767. The value may be found from this formula:

(HI AND 127)*256 + LO + (HI>127)*32768.

For example, HI=0 and LO=100 stores an integer variable of value 100; HI=255 and LO=156 stores -100. (The two expressions add to 0 with overflow).

The string name is held with a pointer to the start of the string, which continues up memory for length LEN. (See LEN in Chapter 5). BASIC 4 differs from earlier BASICs in that each string has a pointer, which points to the string's name lower in RAM. This is to facilitate memory freeing; see Chapter 5 on FRE for this.

BASIC 4 STRINGS: S T R I N G | POINTER: LO | HI |

Main pointer   Pointer back to LEN of itself in low RAM

A function definition has two pointers; one to the definition in the body of the BASIC program, and one to the floating-point dependent variable. They point just after the '=' sign and to the exponent byte respectively. The final byte is garbage, generated when the definition is set up, and is not used.

Strings and function definitions, unlike numeric variables, can be defined so that their pointers indicate some point within BASIC. If a new program is loaded and run, retaining these values (i.e. by LOAD from within a program), the pointers will no longer indicate correct values, so a string of this sort will be garbage, and a function is likely to give a ?SYNTAX ERROR message. Strings can be moved into high RAM using X$=X$+"" and the equivalent for other strings, but functions must be redefined as a rule.

Subscripted variables (arrays)

	OFFSET		NO. OF	LAST DIM+1			FIRST DIM+1			
ARRAY NAME	LOW	HIGH	DIMS	HIGH	LOW	...	HIGH	LOW	...... DATA	

The diagram shows the layout of all three array types. The high-bit conventions for type are identical to those for simple variables (there is no equivalent to the function definition). The 'offset' figure is the total length of the low-RAM part of the array; we shall see how this is calculated. The 'number of dimensions' figure is 1 for a one-dimensional array, e.g. A(x); 2 for a two-dimensional array like C(x,y) and so on.

A set of pairs of bytes holds the value of DIM+1; since dimensions are counted from the zeroth element. Finally, we have the data. This is held in 5-byte batches (reals), 3-byte batches (strings) and 2-byte batches (integers). It is exactly similar to that for simple variables, except that spare bytes are not wasted. For example, the string array data consists of sets of 3 bytes, consisting of the length of each string in the array and its pointer. Strings are, of course, held in high RAM or in the body of a BASIC program. The variables, or pointers, are held in strict sequence, which is ascending order of argument, with the lattermost variables changing least frequently. For example, DIM A(1,2) stores its variables in the order

A(0,0) A(1,0) A(0,1) A(1,1) A(0,2) A(1,2) , and DIM X(1,1,2) in the order

(0,0,0) (1,0,0) (0,1,0) (1,1,0) (0,0,1) (1,0,1) (0,1,1) (1,1,1) (0,0,2) (1,0,2) (0,1,2) and (1,1,2). The position of any one item of an array can be calculated; $X(a,b,c)$ is at $a + b*(1+dim_1) + c*(1+dim_1)*(1+dim_2)$ for instance.

All of the above can be checked using simple BASIC; a program of this sort both sets up a variable and prints RAM contents:

```
10 BB%=100
20 FOR J=1084 TO 1090: PRINT J;CHR$(PEEK(J));" ";PEEK(J): NEXT
```

Line 10 can define any variable; the values of J in line 20 will need juggling unless J is defined in terms of the end-of-program pointer.

The length occupied by an array is easy to calculate (the figure is identical to that of its own offset pointer). The number of bytes is:

$5 + 2*$NUMBER OF DIMENSIONS $+ (DIM_1+1)*(DIM_2+1)*...*(DIM_N+1)*2,3,$ or $5,$

the figure depending on the array type (integer=2, string=3, real=5). In addition, the strings of a string array must be included, and, in BASIC 4, 2 bytes for each string. Examples: X$(1000), defined so that each X$(n) string has length 10, occupies

5 + 2 + 1001*3 + 1001*10 = 13020 bytes, plus 2002 bytes = 15032 in BASIC 4.
    A%(50,50), which holds about 2500 integers, occupies

5 + 2*2 + 51*51*2 bytes = 5211 bytes.

BASIC pointers There are seven principal pointers in Microsoft BASIC. PET/CBM has:

START OF BASIC (usu. 1025)	($28)	(40 dec)	($7A)	(122)
END OF BASIC/ START OF VARIABLES	($2A)	(42 dec)	($7C)	(124)
END OF VARIABLES/ START OF ARRAYS	($2C)	(44 dec)	($7E)	(126)
END OF ARRAYS	($2E)	(46 dec)	($80)	(128)
START OF STRINGS	($30)	(48 dec)	($82)	(130)
END OF STRINGS	($32)	(50 dec)	($84)	(132)
TOP OF MEMORY	($34)	(52 dec)	($86)	(134)

The bold figures apply to BASICs 2 and 4; the order of these pointers is low byte followed by high byte, following the 6502 itself. Knowledge of these locations enables the top of memory (normally fixed when the machine is turned on) to be lowered, thus creating extra RAM space protected from BASIC. See HIMEM & LOMEM in Chapter 5. Arrays can be erased by changing the pointers: see the 'Scatter Sort' in Chapter 5. BASIC can be made to start at other locations than 1025, and so on. This program, for BASIC>1, reports the current values of these pointers within a program. As it stands, two simple variables (X and FN DE(X)) exist, but others may be added earlier in the program and the results watched. The right-hand column of the table is BASIC 1.

```
5000 DEF FN DEEK(X) = PEEK(X) + 256 * PEEK(X+1)
5010 PRINT "        START OF PROGRAM"; FN DEEK(40)
5020 PRINT "END OF PROGRAM/START OF VARIABLES"; FN DEEK(42)
5030 PRINT "   (LENGTH OF PROGRAM ="; ( FN DEEK(42) - FN DEEK(40) ) ; "BYTES )"
5040 PRINT
5050 PRINT " END OF VARIABLES/START OF ARRAYS" ; FN DEEK(44)
5060 PRINT "(NUMBER OF VARIABLES ="; ( FN DEEK(44) - FN DEEK(42) ) / 7 ; ")"
5070 PRINT
5080 PRINT "  END OF ARRAYS/START OF FREE RAM"; FN DEEK(46)
5090 IF FN DEEK(44) = FN DEEK(46) THEN PRINT "        (NO ARRAYS EXIST)"
5100 PRINT
5110 PRINT "        START OF STRINGS"; FN DEEK(48)
5120 PRINT "          END OF STRINGS"; FN DEEK(50)
5130 PRINT "          TOP OF MEMORY"; FN DEEK(52)
5140 PRINT
5150 PRINT "DATA STATEMENT POINTER"; FN DEEK(62)
```

Because these pointers mark the boundary between one set of data and another, it follows that the upper limit over a range is exclusive, not inclusive. A 32K machine has a top-of-memory indication of $8000 on switchon, but this means that $8000 is an upper limit which is not reached, so characters don't appear in the top left of screen. These pointers can all be seen by entering SYS 4 and displaying bytes from 0028 on, with .M 0028 0030.

By defining the variables' area to coincide with the screen, we can watch variables being set up in real time. The program prints the current operation on the top line of the screen, and awaits a keypress before each piece of processing:

```
100 POKE  42,40: POKE  43,128 :REM START OF VARIABLES = $8040 (2ND LINE)
110 POKE 52,207: POKE  53,135 :REM TOP OF MEMORY = $83E8 (BTM RIGHT OF SCREEN)
120 CLR                       :REM MAKES POINTERS ALL SELF-CONSISTENT
130 PRINT "[CLEAR]": POKE 59468,14: REM LOWER-CASE MORE READABLE
200 DIM VA(20) : GOSUB 1000   :REM SUBROUTINE AWAITS KEY (E.G. SPACE BAR)
210 A=1234 : GOSUB 1000       :REM WATCH ARRAY MOVE, 'A' APPEAR
220 DIM ST$(20): PRINT [HOME] "ST$(20)": GOSUB 1000 :REM PRINT TO SCREEN TOP
230 ... ETC ...

1000 GET X$: IF X$="" GOTO 1000
1010 RETURN
```

80-column CBMs require a slightly modified program if the full screen is to be used; and BASIC 1 requires different POKEs in lines 100 and 110 − see table.

The dimensioning of arrays, and filling with null variables, can be watched; so can assignments of all types of variables. Strings fill down from the top of memory, and start again near the top when space temporarily runs out. If several different strings are assigned to the same string variable, FRE can be watched as it moves the most up-to-date value into as high RAM as can be managed.

2.4 BASIC syntax

BASIC is sometimes described as 'English-like'; in fact the resemblance is tenuous. Its syntax has to be learnt, like that of any other computer language. BASIC is a rather ad hoc language, and a comprehensive account of its syntax is made difficult because the interpreter allows great latitude in a program. For example, is RETURN or GOTO 10 valid, if there is no subroutine or no line 10 respectively? How can the correct syntax of READ ... DATA ... RESTORE be defined? Is NEW:!*? valid? The usual approach is to define the individual components of BASIC using some form of the Backus-Naur notation, but I shall spare my readers this experience. The account following outlines the major features of BASIC in a purely descriptive way.

Numerals and literals These are actual numbers and strings, not variables. Examples of the first are 0, 2.3 E-7, 1234.75, and −744; examples of the second are "hello", "ABC123", and "%!£/" where the quote symbols are delimiters (not part of the literal). The rules which determine the validity of these forms are complex; generally, numbers are valid if they contain 0-9, +, −, E and . in certain combinations. Thus, imaginary numbers (e.g. 2i+3j) are not accepted, and 3 E 2E 1 (i.e. $3 * 10^{20}$) and 1.2.3 are not accepted. The only point likely to cause difficulty is the use of E to mean '10 raised to the power ...'. Strings can include any CBM ASCII character; tricky characters can be manipulated with the CHR$ function. However, some characters − 13 (Return) and 0 (null) for example − produce unusual side-effects.

Variables At any moment, a variable must equal a numeral or string; the default values are 0 and the null character respectively. (See Chapter 5 on CHR$ for a discussion on CHR$(0) and "", each of which can be considered a null string). A variable, as the name is supposed to imply, can be changed to other valid values.

Operators (or 'connectives') Binary operators connect two items of the same type, giving a single new item; unary operators operate on a single item, generating a new one of the same type. The CBM numeric operators are completely standard, and are identical in type and hierarchy to those of FORTRAN. The string operators and logical operators are less standard:−

	Binary		Unary	
Numeric	+ − * /		Numeric	+ −
String	+		String	..none..
Logical	AND OR < = >		Logical	NOT

'Dyadic', 'monadic', and 'Boolean' are synonyms for 'binary', 'unary', and 'logical'.

Parentheses   Parentheses (round brackets) signal the translator to process the following data as a unit, completed only when the corresponding right parentheses have been found. Intermediate calculations are stored on the 6502's stack.

Functions   Some of the BASIC keywords are valid only when followed by an expression in parentheses; they may be used on the right of assignment statements or as part of an expression under evaluation. Numeric functions include SQR, LOG, EXP, and SIN; string functions include LEFT\$, MID\$, and RIGHT\$. PEEK, although not a function in the usual deterministic mathematical sense, has the syntax of a numeric function and is considered to be one.

Expressions   An <u>Arithmetic expression</u> is a collection of numeric functions, numerals, real and integer variables, connected with operators and parentheses, and always used in an assignment statement or with PRINT, PRINT#, or CMD. For example:

     SQR(VAL(Q\$(2,3)) + M%) + SGN(Z)*(X>4)

     A <u>String expression</u> is a collection of string functions, literals and string variables, connected (optionally) with parentheses and/or the only string operator, which is '+'. For example:

     STR\$(25) + MID\$("HELLO" + Y\$,3,4) + CHR\$(N)

     A <u>Logical expression</u> evaluates to 'true' or 'false'; it may contain relational operators (<,=,>) and/or logical operators. For example:

     (A=4) OR NOT (21=X)

There is not a sharp distinction between this type of expression and an arithmetic expression. The same routine evaluates them both, which makes possible constructions like   PRINT 1>2   and   ON 2 + (P=Q) GOTO 100,200. See Chapter 5 on AND, NOT, and OR.

Statements   A statement is a syntactically correct portion of BASIC separated by an end-of-line marker or a colon from other statements. All statements begin with a BASIC keyword, or, where LET has been omitted, with a variable. There are some peculiar cases; for example, IF A=B THEN is a statement because its syntax is accepted. (Note: keywords are sometimes called 'statements'). Types of statement include:

     <u>Assignment statement</u>   LET variable = expression. LET is optional. Here, the '=' symbol is used differently from the relational operator '=', and it is distinguished in some computer languages (e.g. ALGOL) by being written ':=' and read 'becomes ...'.

     <u>Conditional statement</u>   IF condition THEN ... . See Chapter 5 on IF.

     <u>Control (or 'sequential') statement</u>   Alters the program's flow of control. GOTO, GOSUB, RETURN, STOP are examples of keywords.

     <u>Input statement</u>   fetches data from a device or from a DATA statement. INPUT, INPUT#, GET, GET#, and READ are the relevant keywords.

     <u>Loop (or 'block' or 'compound') statement</u>   enables many statements to be executed in a block; this is really a structured programming concept, only applicable to CBM BASIC in a loose sense to FOR ... NEXT loops and subroutines.

     <u>Output ('print') statement</u>   sends data to tape, disk, screen, or other output device. See PRINT and PRINT# in Chapter 5 for an account of formatting, tabulation, evaluation of functions, and so on.

     <u>Remark (or 'comment') statement</u>   In BASIC, REM followed by any information, which is ignored by the computer but useful from the point of view of documentation of the program. Lines which are never executed perhaps come into this category:
0 GOTO 100/ 1 VERSION #1/ 100 REM BODY OF PROGRAM never executes line 1.*

     <u>Type conversion statement</u>   converts between string variables and literals/ real variables and numerals/ integers and numerals, using such functions as ASC, CHR\$, INT, STR\$, VAL.

Program lines are made up from statements. Each line is preceded by a zero byte, a link address, and a line number, and terminated by a zero byte. The line itself may contain tokenised keywords (all with their high bit set), double quotes, literals within the quotes, screen editing characters with the high bit set, \$,%, or (type declarators, variables, parentheses, numeric strings in ASCII, punctuation (;:,), ASCII strings in comment statements and DATA statements, and other items, for example '#' as part of GET# and non-standard BASIC used with a modified GETCHAR routine, typically !, or \$.

---

*The slash symbols (/) are a space-saving device, enabling several lines of BASIC or machine-code program to be printed as though only one line were occupied. When this sort of program is keyed in, obviously Return takes the place of '/'.

2.5 Manipulating BASIC and its variables

Pointers, link addresses and linenumbers An ordinary BASIC program is stored as this diagram indicates. The starting address is $0400 (=1024), each line has a 2-byte link pointer and 2-byte linenumber, and is terminated by a zero byte. Normally, no zero bytes appear within a BASIC line, and the linenumbers are all different, in ascending order, and less than $FF00 (=65280). Each link pointer points to the next link pointer in memory, and the chain proceeds regularly upwards, until a zero link signals the end of the program. Any of these features can be modified, either in BASIC or machine-code, enabling non-standard results to be achieved. Conversely, such functions as renumbering, searching BASIC and compressing BASIC can be written when the storage mechanism is understood. Modified BASIC is likely to be more-or-less unstable; it may be difficult to edit, for example.
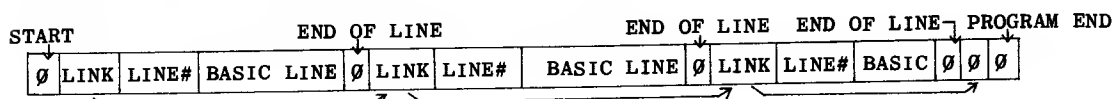
The link addresses and linenumbers are quite easy to locate in either BASIC or machine code; they can also be examined by entering the monitor and reading the memory dump from $0400 onward. This BASIC routine illustrates the principles:

```
10 A=1025
20 L=PEEK(A) + 256*PEEK(A+1): IF L=0 THEN END
30 PRINT "LINK POINTER IS " L ;
40 PRINT "  LINENUMBER IS " PEEK(A+2) + 256*PEEK(A+3)
50 A=L: GOTO 20
```

When RUN, A=current link, L=next link; the program prints both items for every line. The machine-code equivalent, illustrated by this outline routine, uses an intermediate double-byte address to store link addresses. In ROM, the routines at C522/ C52C/ B5A3 for BASICs 1/2/4 search BASIC for a given linenumber, typically when executing GOTO. The short program here carries out a small part of that operation, skipping through the link pointers to the end of the program.

```
        LDA 28      ;A AND X HOLD
        LDX 29      ;START-OF-BASIC
L1      LDY #01     ;Y IS OFFSET
        STA 5C      ;(5C) IS A TEMP.
        STX 5D      ;POINTER
        LDA (5C),Y  ;IF LINK'S 2ND
        BEQ L2      ;BYTE=0, EXIT
        TAX         ;GET NEW X ...
        DEY
        LDA (5C),Y  ;... AND NEW A
        JMP L1      ;AND CONTINUE
L2      RTS
```

Chapter 5 has several examples of this. See for example the 'tiny renumber' routine, which changes all linenumbers which lie within a requested range, by poking the new values for the linenumbers directly into RAM. As another example, look at this BASIC search routine, which prints the linenumbers of all lines which contain the contents of the first line (e.g. line 0) of the program.

```
62000 A=1025: B=256: J=1029: X=PEEK(J): REM X IS FIRST CHARACTER OF LINE 0
62010 P=PEEK(J): IF P=X THEN GOSUB 62500
62020 IF P<>0 THEN J=J+1: GOTO 62010
62030 IF PEEK(J+2)=0 THEN END       : REM END OF PROGRAM FOUND
62040 J=J+4: A=PEEK(A) + B*PEEK(A+1): GOTO 62010 : REM UPDATE LINK AND J
62500 K=1                           :REM TEST REST OF LINE 0 FOR MATCH
62510 Y=PEEK(1029+K): IF Y=0 THEN PRINT PEEK(A+2) + B*PEEK(A+3): RETURN
62520 IF Y=PEEK(J+K) THEN K=K+1: GOTO 62510
62530 RETURN
```

This routine is written without loops, in a form suited to direct conversion into machine code, which is enormously faster than BASIC in this case. The point of the routine is to scan only the BASIC line, while keeping track of the link pointers; line 62510 prints out a linenumber when all the characters in line 0 match some part of BASIC. It is necessary to remember the way in which BASIC is stored in routines like this one; for example, 0 PEEK(1025) will cause all occurrences of PEEK(1025) to be recorded, but 0 EEK(1025) is not tokenised and will probably find nothing.

The actual contents of BASIC may be changed in a systematic way. The short BASIC routine on the next page inserts carriage return characters into REM state-

ments, when REM is the first keyword in a line.

```
50000 A=1025: B=256
50010 IF A=0 THEN END
50020 IF PEEK(A+4)<>143 THEN A = PEEK(A) + B*PEEK(A+1): GOTO 50010
50030 POKE A+5,13: POKE A+6,13: A=PEEK(A)+B*PEEK(A+1): POKE A-2,13: GOTO 50010
```

It operates by searching for the tokenised form of REM (=143 in decimal), and putting three Returns into the REM line.

Note that arrays in memory can be scanned in a similar way. The only difference is that an offset, not an absolute pointer, is used:

```
10 DIM N(7),MM(50),X1$(200),JJ%(6),Q$(19)
20 S=PEEK(44)+256*PEEK(45): E=PEEK(46)+256*PEEK(47): REM START, END FOR BASIC>1
30 PRINT "NAME OF ARRAY: " ;CHR$(PEEK(S)); CHR$(PEEK(S+1))
40 O=PEEK(S+2)+256*PEEK(S+3): S=S+O:          : REM O=OFFSET
50 IF S<E GOTO 30                             : REM S POINTS TO NEXT ARRAY
```

SORT in Chapter 5 uses a machine-code version of this.

The following pair of BASIC subroutines changes the link addresses of lines in their own programs. The first alters a pointer so that a line is skipped; that line is also renumbered 0. It is likely to become visible on editing. When RUN, the hidden line is processed normally, although LIST and GOTO cannot find it.

```
50000 A=1025: B=256
50010 INPUT "CONCEAL LINE AFTER";X
50020 FOR R=1TO1E8: IFPEEK(A+2)+B*PEEK(A+3)<XTHEN A=PEEK(A)+B*PEEK(A+1):NEXT
50025 IF PEEK(A+2) + B*PEEK(A+3)>X THEN PRINT "NON EXISTENT LINE": END
50030 XS=A: REM START LOCN OF LINE X
50040 YS=PEEK(A) + B*PEEK(A+1): REM START OF FOLLOWING LINE
50050 X1=PEEK(YS): X2=PEEK(YS+1): REM LINK ADDRESS BYTES OF NEXT LINE
50060 POKE XS,X1 : POKE XS+1,X2 : REM LINK ADDRESS STRADDLES LINE AFTER X
50070 POKE YS+2,0: POKE YS+3,0  : REM AND PREVIOUS LINE IS NUMBERED 0
```

This second routine demonstrates how CRUNCH can compress BASIC lines together, making them longer than the normal maximum of 80 characters. It must be positioned at the start of BASIC; when it runs, a range of linenumbers is asked for, and these lines are combined into one longer line by deleting link addresses and pointers, putting in colon separators, and adjusting the initial link address to span the entire line. If the line's length exceeds 251, it will be difficult to edit; it will run, however, in most cases, though not if REM is too far from the end of the line.

```
0 INPUT "COMBINE LINES FROM,TO";L,U: C=1025: B=256: E=PEEK(42)+B*PEEK(43)-4
1 LT=PEEK(C+2)+B*PEEK(C+3): PRINT LT;
2 IF LT<L THEN C=PEEK(C)+B*PEEK(C+1): GOTO 1
3 IF LT>L THEN PRINT "LINE NOT FOUND": END
4 LINK=C: C=C+4
5 Q=PEEK(C): IF Q<>0 THEN C=C+1: GOTO 5
6 IF PEEK(C+1)+PEEK(C+2)=0 THEN END
7 LT=PEEK(C+3)+B*PEEK(C+4): PRINT LT;
8 IF LT>U THEN C=C+1: POKE LINK,C-INT(C/B)*B: POKE LINK+1,C/B: GOTO 4
9 POKE C,ASC(":"): FOR J=C+1 TO E: Q=PEEK(J+4)
10 POKE J,Q: NEXT: E=E-4: GOTO 5
11+ --REST OF PROGRAM--
```

If the pointers to the start of BASIC are altered, BASIC can be stored in other places than the usual $0400; for example, it could start at $1000, leaving a large amount of RAM free for other purposes. Similarly (see HIMEM & LOMEM in Chapter 5) the pointers to the top of memory can be changed.

    POKE 40,1: POKE 41,16: POKE 4096,0:NEW

Sets BASIC>1 to start at $1000. The zero byte at the very start is necessary; without it, ?SYNTAX ERROR will be generated. To return to normal, enter

    POKE 40,1: POKE 41,4: POKE 1024,0: NEW

(NEW, or CLR, is the easiest way to ensure the pointers are consistent). A program of this sort may be saved, with its machine-code, by moving the start pointers back to the normal value; the first line of the 'normal' program must be something like

```
0 POKE 41,16:RUN
```
which will run the main program correctly.

The variables themselves may be manipulated: see e.g. VARPTR in Chapter 5. The entire collection of RAM variables can be saved as a RAM image; for example, a large integer array may be saved and later reloaded, providing rapid access to a lot of numeric data. Strings are less easy to handle, because they are not held in the fixed way in which numerals are. This technique is not very easy, since any change in the program length or in the number of variables will cause the data not to match its pointers. Reloading is also made more difficult than it might be by CBM BASIC's tendency to restart programs which use LOAD.

When a program is edited, CBM BASIC always resets the pointers relevant to the variables. In fact the variables are still present, if the new program is shorter than the old; so if the pointers are poked with their previous values, all the variables will be recovered; the only exceptions may be strings held within the program and function definitions.

2.6 LOADing and RUNning BASIC

LOAD or DLOAD followed by RUN is the normal method of running CBM BASIC; the only automatic RUN facility is provided by Shift-Stop, which LOADs and RUNs the first program on tape or CBM disk depending on the version of BASIC in ROM. Both LOAD and RUN are covered in detail in Chapter 5, and DLOAD is explained in Chapter 7. The overlay feature of each load command, when in program mode, is also outlined. RUN executes some initialisation before entering a loop which processes statements consecutively. Before every statement, the Stop key is tested, and the end-of-program byte is checked for (without this, each program would need END) at the end of each line. By dropping some of these subroutines, the execution time of BASIC can be improved; this requires a RAM routine, probably called by a SYS command, to perform the functions of RUN.

Numeric routines are mostly carried out using two 'floating-point accumulators' of 6 bytes each, and some other RAM storage areas in the zero-page. Strings are constructed in the top of memory. The 6502 stack is used by GOSUB and FOR, each of which puts several bytes of data in store on the stack; see Chapter 5. Also, evaluations which include parentheses for priority put intermediate results on the stack. An unexpected ?OUT OF MEMORY ERROR can result if the stack is asked to hold too much data.

```
1 PRINT (1+(2+(3+(4+(5+(6+(7+(8+(9+(10+(11+(12)))))))))))))
```
causes such an error. The limits of the stack are determined by a combination of the number of GOSUBs, FOR loops, and parentheses at any one time.

As each statement is executed, the CONT pointer is updated. In this way, whenever Stop is pressed, CONT can resume the program, since a record is kept of the statement last executed.

2.7 Optimising BASIC

The principal optimisation problem likely to be met with in BASIC is making a program run as fast as possible. (The other problem - shortage of space - I am assuming to be a matter of correct initial design). Input/output, to disks and especially to tape, is slower than processing in RAM; slow printers can also impose a drag on a system. The BASIC program itself can be accelerated using the methods in CRUNCH (see Chapter 5), and the subroutine management techniques in GOSUB (Chapter 5). These rely on knowledge of the way BASIC works to avoid small cumulative losses of time. GOTO can be optimised ensuring that the destination line is as near the start of the program as possible, *or* has a linenumber whose high byte exceeds that of the GOTO line. Some CBM manuals have a section on this subject (almost word-for-word identical to a similar section in Apple manuals). Apart from the routine compression methods of CRUNCH, the most significant timesavers are (i) the use of variables, not constants, and (ii) the deliberate setting up of variables in the best order (i.e. most popular first) at the start of a program. As a simple example,

```
10 FOR A=0 TO 5000: B = B + 1: NEXT
```
takes about 15% longer than:-
```
10 B=0:L=1:FORA=0TO5000:B=B+L: NEXT
```
The point about using variables is that the numerical value is already stored in floating-point form, so the time spent in the conversion process is saved. Generally, loops are likely to make the most difference to running-time, and one-off routines such as exit routines and error messages the least. This program enables single BASIC state-

ments to be timed, so the reader can experiment in this area:

```
PROGRAM TO MEASURE PROCESSING TIME WITH BASIC-

10 N = 100: T1=0: T2=0
20 T1 = TI              :REM STORE THE TIME ...
30 FOR I = 1 TO N
40 :
50 NEXT
70 T2 = TI - T1         :REM ... SO NOW T2 IS THE TIME TAKEN BY LOOP 30 - 50.
80 T1 = TI              :REM STORE THE TIME ...
90 FOR I = 1 TO N
100 :X=123.456
110 NEXT
120 T2 = TI - T1 - T2 :REM T2 = TIME TO EXECUTE LINE 100 AFTER THE COLON.
130 PRINT 1000 * T2 / (60*N) "MILLISECONDS"
1000 REM **********************************************************************
1001 REM *               EXECUTE AND TIME COMMAND(S) IN LINE 100             *
1002 REM *                                                                   *
1003 REM * NOTE THE LEADING COLON, TO ALLOW CORRECTION FOR LOOP PROCESSING *
1004 REM *                                                                   *
1005 REM * CHECK:- ZERO MILLISECONDS SHOULD APPEAR WITH 100: ALONE           *
1006 REM *                                                                   *
1007 REM * INCREASE THE VALUE OF N IN LINE 100 IF THE INSTRUCTION IS FAST    *
1008 REM * NB: SEVERAL LINES OF CODE CAN ALSO BE TESTED WITHOUT DIFFICULTY   *
1009 REM * NB: DEFINING VARIABLES AT START AVOIDS SEARCH TIME ERRORS         *
1010 REM **********************************************************************
```

BASIC<4 has a well-known drawback in the long time spent freeing strings in memory. This means that large arrays (e.g. X$(500)), however convenient for storage of easily recovered data strings, are prone to cause prolonged delays; FRE takes about 1 second with 100 strings, 10 seconds with 350 and 100 seconds with 1100 - see Chapter 5 for a formula. Chapter 4 has details on minimising these delays.

2.8 Differences between ROMs

The major differences between ROMs are listed below. Generally, later ROMs can run all earlier programs, but earlier ROMs may not have some features assumed in later programs. Programs using machine-code calling ROM routines or specific RAM locations are unlikely to transfer between machines. BASIC 4's two versions, 40- and 80-column, are dissimilar in some ways, the 40-column version retaining some features of BASIC 2.

Differences:	BASIC 1	BASIC 2	BASIC 4
RAM map	Input buffer in zero-page	Input buffer $0200 - $0250; more 0-page pointers	
			Tape buffer #2 partly used
ROM map	C000-FFFF	C000-FFFF	B000-FFFF
	Apart from kernel addresses, almost all ROM entry points differ (Ch. 15).		
Monitor	RAM only (see manual)	Machine-language monitor present in ROM	
Interrupt	60 Hz	60 Hz	50 Hz (12-inch models)
Other		General improvements (e.g. LIST).	

Differences which may affect BASIC programs:			
Keywords		GO	GO,DS,DS$, & disk commands
Syntax	Spaces in keywords valid*		
Arrays	See DIM (Ch. 5) for bugs		
IEEE			Improved
Screen			Fast screen; more editing chrs.
Strings		FRE slow	FRE fast (see Ch. 5)
Tape	Data file bugs (Ch. 8)	Data file handling improved	

*In BASIC 1, 'IF 10=LE THEN PRINT "10"' and 'IF F OR G GOTO 100' generate ?SYNTAX ERROR as 'LET' and 'FOR' respectively are assumed. BASIC>1 does not scan tokens in the same way (hence the need for GO). However, in all BASICs there is scope for ambiguity: 'IFY=GORXTHENPRINT"ERROR"' , 'IFS=TANDUGOTO50', and   'Y=TORU' illustrate this.

CHAPTER 3:   PROGRAM AND SYSTEM DESIGN

3.1   General introduction

This chapter explains some of the techniques and thought-processes required to write programs and systems. Chapter 4 provides examples, mostly in BASIC. Chapter 17 has examples and suggestions involving actual systems; the intermediate chapters deal with the hardware and software knowledge required to actually do the job.

Designing a system is a tricky process which is unlikely to be successful without a considerable amount of experience, unless a system is fairly small and informal, and either unimportant or easy to reconstruct in the event of disruption. The difference between small systems and those consisting of many programs operating on a large database, with full validation and crashproofing, and with checking and recovery procedures, is enormous. Obviously it is necessary to assess whether a proposed system is feasible at all, and the optimum amount of work to put into it. Since this book is largely about the PET/CBM, we can leave aside the difficult problems of deciding between rival machines. We can also ignore the special problems of programming external hardware, for example in process control, which is a minority interest. By and large our concern is with a computer, tape and/or disk storage, and probably a printer. What can such a combination of hardware do? Experienced programmers, naturally, already know. For those less experienced, we can subdivide the replies into three categories: results which can be achieved easily, those which are difficult, and those which are impossible. In the first category we have standard packages, if they exist. Sometimes several packages may be able to share data. The absence of programming effort does not, of course, guarantee success. Programs requiring calculations, when the formulas are known, are usually fairly easy; anything from architecture to zoo nutrition might be required. Any type of alphanumeric data can be stored and retrieved, though not necessarily rapidly; dictionaries, tables, price-lists, technical words, names, can be filed and recovered, provided the storage capacity of tape or disk is allowed for. Small business programs, with reasonable crash-proofing, are possible if the processing demands aren't large: invoices and mailing-lists for example. Payroll programs are possible in 4K, in some developing countries. Tidy formatting and output is not a big problem. Nor are slowish graphics.

The second category includes anything really fast. Graphics; fast searches in memory; rapid updating, input, formatting, and output usually require machine-code, which is more difficult than BASIC. Any disk reading or writing which uses a key other than the record number, and is fast, will need to be thought out carefully. Completely crashproof and validated input is not easy. Data may be coded, abbreviated and packed in many ways to save storage space, and so store more data than may seem to be possible at first sight. Where many programs operate on the same data, the order in which they are run may need internal checking. Data checking programs may be needed which provide an assurance that the data on a disk is self-consistent. Some programs may require annual updates, or need to be easy to modify. All these things are comparatively time-consuming and difficult to write. As the workload increases, the viability decreases: sorting the names in a telephone book, performing simulations of atmospheric physics, calculating the payroll of thousands of people, may be impossible. The machine cannot program itself, understand English, correct errors in a specification of a system, or work while switched off in a corner.

Typical complaints (about computer systems generally) are illustrated by these quotations from a medical man: 'They lead to more clerical work, not less... produce sheaves and sheaves of that printout stuff... VDUs are very slow; you can't just read a patient's record, you have to type it in... you could lose all the data! The whole lot!'. And an export manager: 'The biggest disaster is the so-called informal specification. We assumed we were speaking the same language... the program takes days. We'd seen programmes on television where the results come up instantly...'. Retailers are often asked for their 'standard stock control package and PAYE payroll package'; often these do not exist. I have stressed the possibilities of failure, because it is important to realise that this can occur. In practice, the direr prophecies of mass business failures due to microcomputers have not come true: systems which are clearly useless remain unused, and the risks inherent in risky systems are not taken. I don't want to imply, by my mention of this topic, that CBM hardware is unreliable; comparative figures are unavailable, and all computers are liable to hardware problems and software

bugs, and these may be an unpleasant shock to those accustomed to the facade of smooth-running efficiency presented by data processing departments.

3.2   Designing programs

The general idea of BASIC is simple: the program does what it's told, starting at the beginning and continuing to the end, occasionally encountering a GOSUB and executing a subroutine, or encountering a GOTO and jumping to a program line. The conceptual difficulty with programming is the need to understand what the separate commands do. Only when they are more-or-less grasped is it possible to tell the computer what to do. As a simple example, consider a set of short reports being printed by an ordinary computer printer; at the end of each one, a 'top of form' command has to be issued, whereupon the paper is shifted in preparation for the start of the next report. Suppose some reports take several pages, and the printer has no automatic facility to leave a few lines at preset intervals. Then it is necessary to keep a running total of the number of lines printed, and to check this number after printing each line; if the total equals a preset value, 'form feed' is issued, and the total reset to zero, to be used for the next page. Typical complications include lines which belong in batches, and are not to be separated, page numbers, running totals, and titles dependent on the last line of the previous page. In this way, an apparently straightforward task of programming can become complex.

There are many theories on the 'best' programming methods. For example, 'top-down' programming designs the main flow first, then the subsidiary routines, while 'bottom-up' programming starts with the subroutines. But 'structured programming' is undoubtedly the major buzzword. There are several versions of this, ranging from the avoidance of 'GOTO', through the use of nested routines, to the attempt to match the structure of the data, as it is filed, with the program. CBM BASIC lacks the syntax to apply such techniques directly, but they can be simulated. The object is to produce programs which are easily read, so that in turn they can be changed or reused with little difficulty. In practice (in my opinion) programmers' methods are always ad hoc and chaotic, and maintainability of programs is possible (if at all) only because programs are tidily arranged in routines with heavy commentary. Similarly, flowcharts, once regarded as highly scientific, are widely regarded as obsolete, replaced for the most part by pseudo-programming languages. But it is not obvious why one form of notation should be superior to another; the sad fact is that any complex program will remain complex in whatever way it is written down. For these reasons, I suggest that the reader treats 'definitive' announcements on these subjects with scepticism.

There are two types of non-linear program flow: a loop (when the program jumps back repeatedly to an earlier point in the program; forward jumps are essentially still linear), and a branch (when differing parts of a program are selected according to the results of some test. Several flowchart representations are:-

Flowcharts of loops   Flowchart of branch / casentry
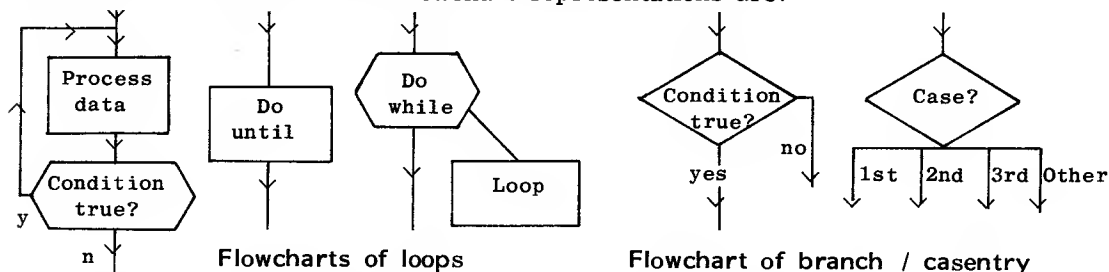
There is a British Standard on flowcharting. For our purposes it is sufficient to denote branches by a diamond (or similar) shaped box, usually containing the condition as a question, and processing by a rectangular box in which are written details of the processing. Arrowed lines indicate the direction of flow of control. Detail may be at the level of single instructions, or at almost any level of vagueness, depending on whether the object is to present a detailed or overall picture of the program. In CBM BASIC, a loop is usually of the form   FOR A=B TO C STEP D ... NEXT A   with an implied count from B to C in steps of D. Changing the variables within the loop is apt to prove confusing. The orthodox structured forms of DO WHILE and DO UNTIL do not count, but wait until a condition is no longer true and a condition becomes true respectively. These forms can be simulated easily in BASIC; for example, a construction like:-

```
DO WHILE LINECOUNT<50
        PERFORM ROUTINE TO PRINT LINE AND INCREMENT LINECOUNT
ENDDO
```

can be written in this way (or many others):-

```
FOR A=1 TO 1000
LINECOUNT=LI+1
IF LI=50 THEN A=1000: GOTO x
        GOSUB y TO PRINT LINE
x NEXT
```

And the casentry construction can be written as a series of IF statements or, in situations where a variable takes values 1,2,3,... , as ON ... GOTO or GOSUB. With a little practice, all this becomes straightforward. When flowcharting, to avoid tangling of lines it is usual to adopt a direction convention. Typically, the general direction is down the page, with loops branching back anticlockwise and forward jumps clockwise to avoid clashes. The diagram below gives typical extra symbols which may be included in this sort of chart.
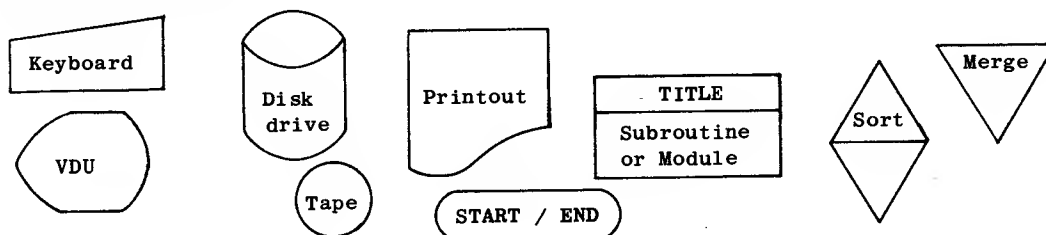
These symbols are based on notation for large computers; the disk isn't very like a floppy disk, and the tape is a spool rather than a cassette. But the general idea is clear enough. Other types of chart include those with subprograms connected by reference labels, rather than lines. A page number and label marks each jump and branch. This technique is suitable for machine-code flowcharts, which are unlikely to have tidy loop structures. The 'Nassi and Schneiderman' notation is topologically identical to a flowchart, but is rearranged to increase the space for explanatory detail. It has 'process boxes' of four types: condition (normally binary); loop with test after processing; loop with test before processing; and a plain processing box.

Enter Surname	
Does it exist in the array? N Y	
'Not found'	Print name, address, comment
	Until no more
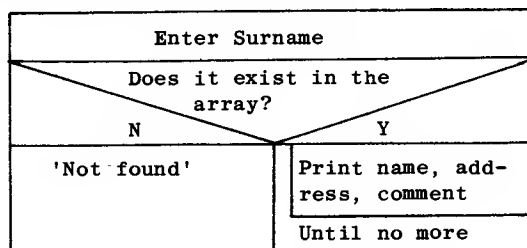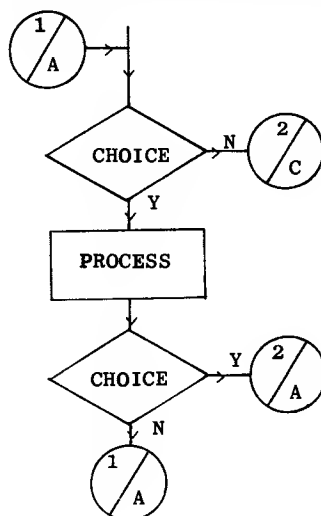
There are innumerable techniques, each with local variants and modifications, and the purpose of this section is to give some idea of the appearance of the resulting documentation. Any sizeable program will be far more complex than the simple examples presented here, and may occupy several pages of 'text'.

The internal detail of a program may be documented and clarified in various ways. Firstly, subroutines may be handled in a systematic way: they can be documented (see Chapter 4) and arranged within the program to maximise efficiency (see GOSUB in Chapter 5). In principle, standard subroutines are a possibility*. Variables' names can be selected in some systematic, meaningful way, within the

---

*MUSE (Micro Users in Secondary Education) has standards intended to enable easy interconversion of programs between machines. (See e.g. Ed'l Comp'g,July '80). N Hampshire has a book of 'Standard Subroutines' for PET/CBM, using linenumbers 10000-30000. A McGraw-Hill book has 'BASIC scientific subroutines for all computers'.

limitations inposed by the fact that only the two leading characters distinguish between names. (See Chapter 2). Line-number maps, including subroutines, can be useful in navigating long BASIC programs; and conversely, intricate programs with many GOTOs may be deciphered *in extremis* by simply writing down all the linenumbers in execution sequence, perhaps revealing islands of code which are never used. The logical process which a program carries out is also depictable in many ways. A condition table is one method (see diagram) which in principle can be drawn up without any programming knowledge, to be turned into a program as a routine task. The patterns of Ys and Ns, which should cover all possible combinations of the conditions, correspond to one or more actions, marked with 'x'.

Conditions:	Stock > reorder level?	Y	Y	Y	N	N
	Stock minus stock out > reorder level?	Y	N	N	N	N
	Stock out > stock?	N	N	Y	N	Y
Actions:	Issue stock	x	x	–	x	–
	Issue reorder request	–	x	x	–	–
	Part issue stock / increase commitments	–	–	x	–	x

'Data-structured design' is another methodology, associated, particularly in the U.K., with Michael Jackson. Its object is to simplify matters by matching file structure to program structure. If BASIC compilers come to be widely used, techniques of this sort will become more applicable to BASIC than they are at present. Before describing (in outline) the tenets of this school of thought, we must clarify the idea of a computer 'file'. CBM disk and tape files are described in detail in Chapters 6 and 8 respectively, but a few words of introduction are necessary. In the usual office sense of the word, 'opening a file on Mr Smith' means either looking at Mr Smith's records or starting a new folder of details on him. This is *not* a computer 'file'. In the computing sense, a 'file' is a collection of many records, which for convenience have a name assigned to them, and which are more-or-less similar in content. A 'name-and-address file' contains details not only of Mr Smith, but of many other people. 'Opening a file' means preparing the computer to read or write individual records from or to the file. A simple example might consist of a file with (a) a header record, i.e. a single record, holding perhaps the date on which the file was last used; (b) a consecutive set of records, of which some are to be printed, and others are not. These would be distinct in some way; for example, items might be marked as deleted, or as having fallen below the reorder level. (c) A trailer record might mark the end of the file, typically holding totals. The diagram shows the structure of this file, with a standard box notation:

A structured program to process this file is illustrated in the second chart, which gives a general picture of the processing without much detail. The modules and subroutines, if they are sufficiently commented and REM'd within the program, ought to make detailed processing fairly easy to follow. Note the correspondence between the program and the data structure.

```
                    ┌──────────┐
                    │   FILE   │
                    └──────────┘
      ┌──────────┬──────┴──────┬──────────┐
┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐
│ HEADER   │ │ 'LIVE'   │ │ 'DEAD'   │ │ TRAILER  │
│          │ │ RECORD   │ │ RECORD   │ │          │
└──────────┘ └──────────┘ └──────────┘ └──────────┘
```

CONTROL LEVEL

```
                  ┌────────────────────┐
                  │  PROGRAM CONTROL   │
                  └────────────────────┘
   ┌────────┬────────┬─────┴─────┬────────┬────────┐
┌──────┐ ┌──────┐ ┌──────┐ ┌──────┐ ┌──────┐
│ OPEN │ │ READ │ │ READ │ │ READ │ │CLOSE │
│FILES │ │HEADER│ │UNTIL │ │TRAILER│ │FILES │
│      │ │      │ │ END  │ │      │ │      │
└──────┘ └──────┘ └──────┘ └──────┘ └──────┘
```

MODULES

```
              ┌───────┬───────┐
           ┌──────┐ ┌──────┐
           │'LIVE'│ │'DEAD'│
           └──────┘ └──────┘
          ┌────┴────┐
    ┌────────┐ ┌────────┐
    │ READ A │ │PRINT A │
    │ RECORD │ │ RECORD │
    └────────┘ └────────┘
```

SUBROUTINES

Algorithms An algorithm is a set of rules which (if the algorithm works!) generate a solution to a problem. Taking care with algorithms will improve the logical accuracy of programs and probably their speed and efficiency. Typical algorithms deal with sorting, merging, and similar large-scale processing, down to the details of rounding, page throws, and date processing. As concrete examples, let's briefly consider five types:

(i) Linear programming. This is a technique for maximising a linear combination of variables subject to certain restrictions. It is not easy, or necessary, to understand the steps involved, which slowly but surely grind out the solution.

(ii) Warnsdorf's Rule provides a means to generate complete knight's tours round a chessboard. The rule is: move the knight to the square with the fewest exit squares. This often (not always) gives a solution. There is no real justification for the rule; it gives an attack on the problem, without an indication of whether its solutions are only a subset of the total of solutions, or of the procedure to follow when the rule finds several squares which are equally legitimate.

(iii) Decision-tree pruning is a technique used in the analysis of games (e.g. chess) by computer, where the 'tree' of moves and replies has a colossal number of 'branches'. When any 'branch' is assessed as 'worse' than some other branch, no further time is spent on that 'branch'. (The 'alpha-beta algorithm' is an example).

(iv) Sorting. Dates stored in the form DDMMYY or MMDDYY may be sorted three times, by year, month and day. YYMMDD requires only one sort.

(v) 3-Dimensional 'tic-tac-toe' or (U.K.) noughts and crosses has a variation in which the first player to make a line loses. An algorithm for the first player is: start at the centre, then make all moves exactly opposite to the opponent's. This ensures that the first player cannot lose. (It doesn't prove that a draw is impossible).

Formal logic is sometimes helpful in simplifying complex conditions which have to be met: see Chapter 5 on AND, OR, and NOT.

3.3 Designing systems

'Systems Analysis' has no necessary connections with computers. The approach is to examine exactly what you'd want a computer to do, taking particular note of the 'odd 10%', or whatever figure applies, of oddments, exceptions, and special cases. Useful clarification may result irrespective of computers, the mental effort producing results which are unexpected, economical, and neat (in the words of Prof. Parkinson). Translation of the result to a computer may nevertheless be unsuccessful. Typical mistakes include allocating insufficient space for data, so some figures are too large to fit into a file; failure to test the timing of a system, in which case the performance may fall off dramatically as data is added; adding new features during development, of a type likely to increase the number of bugs in the system. (For example, an 'escape' key might be introduced to take the operator back to the start of the system, if the wrong part of a program has been inadvertently called. The incomplete data already set up may cause unforeseen errors). File layout is important if any sort of elaborate technique is to be used (i.e. anything other than sequential access or, with disks, access of relative records by record number). Once a database is set up, apparently simple operations like sorting on some unusual field, not allowed for in the design, or deleting or inserting records, may simply take too long to be workable. The aim must be to achieve a flexible design, since it is all but impossible to think out all the implications of a system beforehand, and in any case may not be cost-effective with cheap computers.

A complete system typically has a *menu* of options; entering a numeral or letter at the keyboard calls either a new program from disk, or enters a subprogram within the program which holds the menu and some program responses. In this way, functions of the system can be partitioned up in discrete, tidy units. A separate routine may handle each of the three operations of adding records, deleting records, and amending records, for example; another batch of programs might handle inventory reports, invoicing reports, outstanding orders, and so on. Microcomputer systems are usually *interactive*. This means that files are modified at the time data is keyed in. The alternative type of design is that of *batch systems*. These are common in mainframe (i.e. big computer) environments, the idea being to store data on file, and later run a program to check this data and add it to the current file, updating it by the batch of new data. In the same way, output can be 'spooled', saved on a file for later printing in mass. This is an efficient way to use a big machine, since successions of jobs can be run, and the computer doesn't waste time awaiting input from terminals. There may be insufficient tape or disk storage space with small machines to make batch processing possible. Note however that from the security point of view, running separate

batches may be preferable to direct updates, because, if a check shows that the files contain 'corrupted' (i.e. wrongly written, scrambled) data, the previous copy of the files and the new data can be re-run.

The relation between data storage – in RAM, tape or disk – and the frequency with which it is accessed is one of the main features of system design: see Chapter 17 for examples of this and the related problems of the use of printers for 'hard copy'.

Specific computer techniques   (i) Data compression and codes. It is possible, and may be necessary, to save storage space by encoding data. The following chapter has routines to compress integers to half their length, and to combine many on-off flags into a single number.

(ii) Checkletters and checkdigits. These guard against wrong input by providing a test for self-consistency, typically for use with a reference number of a client or item. Chapter 4 has examples.

(iii) Sorting. The capacity to sort data and store it in sorted order is important in large-scale data processing for two reasons. First, reports, printouts, and lists may be required in order – typically alphabetic. Secondly, the knowledge that data is sorted enables much faster processing to be possible than would otherwise be the case. Merging new data with old typically requires the matching of two sorted files; in this way, at any moment only two records need to be compared to determine whether the new record is to be inserted into the file, used to update its existing equivalent, or ignored temporarily while the main file is read again. And searching data by the 'binary chop' method – equivalent to opening a telephone book in the middle, checking the name sought against the middle name, and continually halving the size of the chunk of text which must hold the target name – needs sorted data. Chapter 4 outlines some important aspects of sorting.

3.4   Timing, 'sizing', and checking systems

When considering the practicability of a large system, it is often worthwhile to write programs to generate 'dummy' data, to simulate a full file. This data can be generated with the help of RND, with which both numbers and alphabetic strings of data can be constructed. (With CHR$ in the case of strings). By testing for inequality, strict ascending or descending sequences are easy to simulate. In the light of tests on this data, improvements in the logic or file-structuring may be suggested.

Estimating the storage capacity to run a system is relatively straightforward: in the simplest case, all records are the same length, so the product of the maximum number of records and the record-length gives the solution. This figure can usually be reduced by data-compression techniques, at the cost of extra programming time. Sequential files, in which records can differ widely in length, obviously occupy space in proportion to the average record length. Disk systems usually reserve some storage for their own operating system, to hold directories and so on, and this must be taken into account if space is short. In addition, the pair of disk drives in most systems are operationally distinct, so that the data may have to be held in a subdivided form on two (or more) disks. When this happens, it is of course important to ensure that each disk independently has sufficient room for its own quota of information.

Testing systems is not particularly easy. (See Chapter 17 on this subject). The writer, however, does at least have informed knowledge which should ease the pin-pointing of likely errors. On the other hand such knowledge may simply result in unconscious or conscious avoidance of areas known to be suspect. For this reason, the user is often asked to supply test data and try it in the system, and to check that its results are correct. This process will often expose assumptions which the programmer has wrongly made, but it is unreasonable to expect such testing to be thorough. There may be parts of programs which are not tested; and systematic errors may not be revealed, because the combinations of data which show up the error happen not to be entered. Systematic errors, in which, for example, every 44th record is lost, or records of length 254 are corrupted, or items on an invoice after the tenth are duplicated, are nearly always caused by programming errors. Unfortunately the triggering combinations of circumstances may be sufficiently complicated to produce errors apparently at random. Apart from testing every part of each program at least once, and ensuring that test data gives consistently correct output, commercial programming practice is to try to minimise program errors by insisting on standard methods, heavy documentation, and 'walkthroughs'. The latter are a kind of group criticism of a programmer's design, as a result of which the programmer is supposed to improve his or her program. The effectiveness of such methods remains in some doubt.

CHAPTER 4: EFFECTIVE PROGRAMMING IN BASIC

4.1 Specific BASIC problems and solutions

This section deals with the following topics:

4.1.1 Subroutines and documentation

Subroutines are used to handle an enormous variety of processing tasks: setting scrolling windows on the screen, printing error messages, inputting and formatting data, reading passwords, reading a record from a disk file, and so on. If they are to be usable as standard subroutines, a certain amount of documentation is helpful. The example converts a hexadecimal number into a decimal, and prints the answer. All the variables used by the subroutine are listed, with an example or two to illustrate the method of use. If the subroutine itself called other subroutines, these too would be listed. Note that the documentation occupies far more space than its routine.

```
550 REM***  ONE LINE HEXADECIMAL TO DECIMAL CONVERTER ***
555 REM
560 REM    CONVERTS STRING OF 4 HEX DIGITS INTO DECIMAL NUMBER AND PRINTS RESULT
565 REM       USES J, L, L%, L$
570 REM       ALL THESE ARE ALTERED BY THE ROUTINE
575 REM
580 REM       EXAMPLE OF USE:
585 REM       L$="ABCD" : GOSUB 600 :        PRINTS 43981
590 REM
595 REM
600 L=0:FORJ=1TO4:L%=ASC(L$):L%=L%-48+(L%>64)*7:L$=MID$(L$,2):L=16*L+L%:NEXT:PRINTL:RETURN
```

A similar decimal-to-hex conversion routine follows; this uses the same four variables, but the relevant variable on entry is L, not L$.

```
500 L=L/4096:FORJ=1TO4:L%=L:L$=CHR$(48+L%-(L%>9)*7):PRINTL$;:L=16*(L-L%):NEXT:RETURN
```

4.1.2 Checkdigits and checkletters

are (usually) suffixes, computed by an algorithm, which are appended to important alphanumeric data. Typically, the data involved is a reference number or some key number in a system. The composite data is made internally consistent, so that keying-in errors can be detected. As an example, consider International Standard Book Numbers (ISBNs). These consist of 9 digits followed by a checkdigit of 0-9 or X. The 9 digits are codes for the publisher and the title; the checkdigit is computed by multiplying each numeral in turn by 10,9,8,..,4,3,2 and adding the result. The remainder after division by 11, when subtracted from 11, is the checkletter (except that 10 becomes X, and 11 becomes 0). It is true that any ten random numerals have 1/11 chance of forming a valid ISBN, so the system is not foolproof. But the point is that the most common input errors are protected from entry to the system, if the computer is programmed to test the checkletter. There are two common typing errors: the first is the entry of a completely wrong single value (e.g. 7 instead of 1), and the second is the transposition of two adjacent keys. Because of the system of weighting, and the use of the prime number divisor, either of these mistakes is entirely preventable. Another algorithm assigns 23 characters, A-W, as checkletters, depending on the result of division by the prime number 23. As a refinement, 'O' becomes 'X' and 'I' becomes 'Y'.

Because this form of validation is easy to implement with computers (it is too arduous for human operators) a checkdigit system may be well worth implementing; without it, whole sets of data may be miskeyed because of some misunderstanding about the layout of an item number or customer number.

If ISBN$ is a string of nine numerals (without spaces), this routine computes the ISBN:

```
10 CT=0:  FOR L=1 TO 9: CT = CT + (11-L)*VAL(MID$(ISBN$,L,1)): NEXT
20 CD$=STR$(11 - CT + INT(CT/11)*11): REM 11 MINUS REMAINDER OF CT DIVIDED BY 11
30 IF VAL(CD$)=11 THEN CD$=" 0"       : REM ALLOWS FOR CBM'S STRANGE STR$
40 IF VAL(CD$)=10 THEN CD$=" X"
50 PRINT ISBN$ + CD$                  : REM FULL ISBN
```

4.1.3 Codes.
BASIC logical functions use 16 bits in all. If we forget the negative first bit, we can hold up to 15 on-off flags in a single real or integer variable. We can test any single bit with:

```
IF FL AND 2↑N THEN ...      :REM WHERE N = 0 TO 14
```

And we can reverse any bit, leaving the rest untouched, with:

```
FL = FL - 2↑N *(2*((FL AND 2↑N)=0) + 1)
```

This technique is useful in storing, in a compact form, data which might otherwise be written to a file as 'Y' or 'N', or some other pair of alternatives.

4.1.4 DATA: processing steps; and relocating DATA subroutines.
The following coin analysis program, which converts a number of wages/ salaries into their breakdown by notes and coin, shows one method for dealing with irregular steps: the values, of which there are seven here, are stored in an array:

```
10 DATA 7,10,5,1,.5,.1,.05,.01       :REM 7 U.K. DENOMINATIONS
20 READ NUMBER OF DENOMS: DIM CN(NU), QU(NU):REM COIN/NOTE DENOMS AND QUANTITIES
30 FOR J=1 TO NU: READ CN(J): NEXT    :REM READ DENOMINATIONS INTO ARRAY
40 INPUT "NUMBER OF EMPLOYEES"; EMPLOYEES: DIM SALARIES OF (EMPLOYEES)
50 FOR J=1 TO EM: INPUT SALARY OF (J): NEXT
100 FOR J=1 TO EMPLOYEES
110 FOR K=1 TO NUMBER OF DENOMS
120 X%=SAL(J)/CN(K): SAL(J)=SAL(J)-X%*CN(K): QU(K)=QU(K)+X%
130 NEXT: NEXT
200 FOR J=1 TO NU: PRINT CN(J) "=" QU(J): NEXT
```

Strictly, to avoid any possibility of rounding error, line 50 could include
:SA(J) = SA(J) + CN(NU)/2: NEXT, adding in this example ½p to each salary. Line 10 can be replaced by any currency combination, provided the denominations are in order, and the first DATA value is the total number of denominations. Note that DATA statements can be made relocatable; this avoids problems which can arise when new DATA statements are inserted before existing ones. READ operates purely sequentially, so the introduction of new data may spoil previously correct routines. One method is:

```
10000 REM STANDARD SUBROUTINE WITH 'DATA'
10010 RESTORE
10020 FOR L=1 TO 1E10: READ X$: IF X$<>"SEARCH M/C" THEN NEXT:REM READ 'TIL NAME
10030 REM *** READ DATA HERE ***
10040 RETURN
10050 DATA SEARCH M/C,100,0,45,34,66: REM ETC.
```

4.1.5 Date processing.
We have three date routines here: the first calculates the day of the week given the date, the second calculates days-between-dates, and the third

```
1 REM *************** ZELLER'S CONGRUENCE ****************************
2 REM *         FINDS DAY OF WEEK FOR ANY DATE                       *
3 REM ***************************************************************
4 REM * 'CENTURY' IN ITALIAN SENSE: 19 FOR 20TH CENTURY             *
5 REM * IF WE ASSUME 19, LINE 50 BECOMES:                           *
6 REM *      50 J = INT(2.6*M - .19) + D + Y +INT(Y/4) - 34         *
7 REM *                                                             *
8 REM * DATES MAY BE TESTED FOR IMPOSSIBILITY BY AN ADDITIONAL ROUTINE *
9 REM ***************************************************************
10 DATA SUN,MON,TUE,WED,THU,FRI,SAT
20 FOR J = 0 TO 6: READ D$(J): NEXT :REM TABLE OF DAYS OF WEEK
30 INPUT "DAY,MONTH,YEAR,CENTURY"; D,M,Y,C
40 M = M-2: IF M<1 THEN M=M+12: Y=Y-1: REM LEAP YEAR ALLOWANCE
50 J = INT (2.6*M - .19) + D + Y + INT(Y/4) + INT(C/4) - 2*C
60 J = J - INT(J/7)*7
70 PRINT D$(J)
```

is a short validation routine, which checks that a combination of day, month and year is valid, allowing for leap years (but not for 1600, 2000 etc. not being leap years).

ROUTINE TO CALCULATE NUMBER OF DAYS BETWEEN DATES

```
10 DATA 0,31,59,90,120,151,181,212,243,273,304,334: REM DAYS ELAPSED
15 DIM D(12)
20 FOR J=1 TO 12: READ D(J): NEXT: REM DAYS ELAPSED BY MONTH; NOT LEAP YEAR
99 REM *** NOTE U.S. USAGE IS M,D,Y BUT U.K. USAGE IS D,M,Y ***
100 INPUT "DATE1"; D,M,Y: GOSUB 2000
105 DX = DE
110 INPUT "DATE2"; D,M,Y: GOSUB 2000
115 DY = DE
116 PRINT DY-DX
120 GOTO 100
1990 REM ***********************************************************************
1991 REM * DAYS ELAPSED BETWEEN DATES SUBROUTINE.  THIS FUNCTION COMPUTES *
1992 REM *  DAYS SINCE AN ARBITRARY EARLY DATE IN THE CENTURY, USING        *
1993 REM *  DAY OF MONTH + DAYS ELAPSED DURING YEAR + DAYS IN CENTURY       *
1994 REM *  WITH CORRECTION FOR PAST, AND POSSIBLE PRESENT, LEAP YEARS.     *
1995 REM ***********************************************************************
2000 DE = D + D(M) + 365*Y + INT ((Y-1)/4) - ((INT(Y/4)*4=Y) AND (M>2))
2010 RETURN
```

```
6200 OK=-1 AND Y>81 AND Y<85 AND M>0 AND M<13 AND D>0 :REM Y,M,D INTEGERS ONLY
6210 OK=OK AND D<32+(M=4 OR M=6 OR M=9 OR M=11)+(M=2)*(3+INT(Y/4)*4=Y))
```

Line 6200 tests for a year of '82 to '84; obviously other values may be substituted.

4.1.6 Error messages are used to signal to the operator that an error has been made. This short routine prints the message in reverse at the bottom of the screen, then deletes it after a short delay. EM$ holds the message, (e.g. 'IN SALES CODE' or 'INVALID DATE'), which is preceded by *** ERROR on the screen:

```
12000 rem ** error message (max.length 19) with delay loop and remove **
12005 print"[home][down][down][down]";:for I=1to10:print"[right][down][down]";:next:print
      "[revs]*** ERROR "em$" [rvso]";
12010 for I=1 to 2500:next
12020 for I=1tolen(em$)+11:print"[left] [left]";:next
12025 return
```

4.1.7 Hard and soft coding. 'Hard coding' means that important parts of a program use constants; 'soft coding' means variables are used. Soft coding is usually easier to modify, but slightly more trouble to write. See the second example under MID$ in Chapter 5 as a specimen. Section 4.1.4's coin analysis program, in which a simple change in a DATA statement can convert a program to run with any set of currency denominations, illustrates the same lesson.

4.1.8 INPUT of data. Chapter 5 (under INPUT) and Chapter 2 outline the problems of the ordinary INPUT statement, and include cures, notably for the crash when Return alone is pressed. (The easiest solution is POKE 3,1 or POKE 14,1 or POKE 16,1 for BASICs 1,2, and 4 respectively).

In order to input commas within strings, elaborate techniques using GET are necessary, of which the following is an example. When GOSUB 70 is called within a program, a reasonably crashproof input results (with a flashing cursor), returning the string as ZZ$. Line 76 allows for the 'delete' key. As we shall see on the next page, this subroutine is a very small-scale version of a completely watertight INPUT.

```
69 REM ** SPECIAL INPUT ROUTINE FOLLOWS, WHICH RETURNS STRING ZZ$ **
70 ZZ$ = "": POKE 548,0 : REM LOCATION=167 WITH BASIC 2 & BASIC 4.(FLASHES CURSOR).
72 GET ZA$: IF ZA$="" THEN 72
74 IF ASC(ZA$) = 13 THEN PRINT" ";: POKE 548,1: RETURN
76 IF ASC(ZA$) = 20 THEN GOTO 84
78 ZZ$ = ZZ$+ZA$
80 PRINT ZA$;
82 GOTO 72
84 IF LEN(ZZ$) > 1 THEN ZZ$=LEFT$(ZZ$,LEN(ZZ$)-1): GOTO 80
86 IF LEN(ZZ$) = 1 THEN ZZ$="" : GOTO 80
88 GOTO 72
```

The BASIC routine on the next page (not for the faint-hearted!) is a successful input routine which is fully parameterised and has the following characteristics:

VARIABLES   'F' prefix refers to screen format:
             FT=TOTAL NUMBER OF ITEMS TO BE INPUT FROM THE SCREEN
             FC=NUMBER OF CURRENT ITEM; ALWAYS <= FT & FL=LOWEST ITEM INPUT
             FH%(), FV%(), FL%(), and FS$() hold horizontal position and vertical
             position of start of item/ maximum length/ type of field. The 'type'
             may be a string ("S"), integer ("I"), or 2-decimal point number ("N")
             'J' prefix refers to input from screen:
             JH, JV, JL, and J$ = current horizontal, vertical, length, and type.
             J1$ is a single character, J1 its ASCII value, and JS$ the current
             input being built up. J$() holds the array of FT inputs from the
             screen. Finally, JD is a decimal-point counter.

SUBROUTINES 100 HTAB & VTAB USING JH & JV COORDINATES; SEE CHAPTER 5
             120 GET NON-INITIAL CHARACTER WITH FULL VALIDATION
             140 NUMERAL PROCESSING ROUTINE (ENSURES DEC. PT. CORRECT)
             160 GET INITIAL. PERMITS USE OF '<' AND '>' FOR BACK/FORWARD STEP
             190 REPRINT 2 D.PT. NUMBER, ADDING '.' AND ZEROES IF ABSENT
             200 PRINT 'CURSOR', A SINGLE GRAPHICS CHARACTER
             220 DELETE SINGLE CHARACTER, REPLACE WITH SPACE
             250 ** INPUT ROUTINE **
             300 PROCESS STEPS: '<' BACK, '>' FORWARD, WHERE POSSIBLE

The length of each variable is defined, so screens of the sort illustrated in section 9.3 can be used - there is no need to follow each input by a blank line. Short demonstration routines (below) show how the routine is used. Unfortunately, flexibility in input is not very easy to achieve. The routine ignores characters which are not numerals, alphabetics or punctuation. The double-quote (") is ignored, and must be replaced by the single quote ('), because of problems which may arise in strings which contain a quote. All upper-case keys are ignored, except for alphabetics; shift-space is converted to space, and shift-return to return. In this way, fields which are to be compared or searched, which may appear different to the computer because space (ASCII 32) is held differently from shift-space (ASCII 160), are held correctly, and shift-return, which typists naturally regard as identical to return, is treated as a normal return. The 'cursor' is a static graphics character, which does not flash. It can be controlled by the keys '<' and '>', which step through the fields on the screen either back or forwards. The cursor control keys are not used, since they are unfamiliar to typists. The previous values entered in each field are displayed, to be overwritten by new values if desired (but not otherwise), which speeds input. Finally, input of integers allows only 0-9; input of strings allows all alphanumerics and punctuation marks; and input of real numbers assumes two decimal places, and will not allow input which infringes this. For example, if the length of a number is specified as 6, 999.99 is the largest number which may be input; the attempt to enter 9999 will be disallowed. The decimal point, followed by 00, is automatically inserted if omitted.

The first part of the example program defines six inputs; these are (i) a single letter, which must be A or B; (ii) three integers of maximum length 2, which make up a date; (iii) a string of length 25, perhaps a name or comment; (iv) a string of maximum length 3, which, if 'YES', causes the screen of data to be accepted, and processing to continue. (Otherwise, '<' is used to go back to amend some entry). In practice, thirty or so separate entries can be made easily from a single screen.

```
10000 DATA S,I,I,I,S,S :REM TYPES. NOTE THAT N=2 DECIMAL PLACE NUMBER.
10010 FOR J=0 TO 5: READ FS$(J): NEXT   :REM FILL ARRAY OF TYPES
10020 DATA 1,2,2,2,25,3:REM LENGTHS OF EACH INPUT
10030 FOR J=0 TO 5: READ FL%(J): NEXT   :REM FILL ARRAY OF LENGTHS
10040 DATA 20,10,13,16,4,3 :REM HORIZONTAL START POSITIONS - TYPICAL VALUES
10050 FOR J=0 TO 5: READ FH%(J): NEXT   :REM FILL ARRAY OF HORIZONTAL POSITIONS
10060 DATA 2,5,5,5,10,24    :REM VERTICAL START POSITIONS   - TYPICAL VALUES
10070 FOR J=0 TO 5: READ FV%(J): NEXT   :REM FILL ARRAY OF VERTICAL POSITIONS
```

This routine must be run before any input takes place. A further subroutine prints the screen details from which the input will be made: again, see section 9.3 for a screen layout, which incorporates variables. Assuming the strings are stored in the array J$(), as in the example following, the screen printing subroutines looks like this:

```
100 REM HTAB, VTAB USING JH AND JV COORDINATES; THEN RETURN
120 GET J1$: IF J1$="" THEN 120
122 J1 = ASC(J1$)
124 IF J1>127 THEN IF J1<193 OR J1>218 THEN J1=J1-128:J1$=CHR$(J1)
126 IF J1 = 13 OR J1=20 THEN RETURN
128 IF J$="S" THEN IF J1<32 OR J1=34 THEN J1$="":RETURN
130 IF J$="I" THEN IF J1<48 OR J1>57 THEN J1$="":RETURN
132 IF J$="N" THEN GOSUB 140
134 RETURN
140 IF J1<46 OR J1>57 OR J1=47 OR (JD>0ANDJD=LEN(JS$)-2) THEN J1$ = ""
142 IF JD>LEN(JS$) THEN JD=0
144 IF J1=46 AND JD<=LEN(JS$) AND JD>0 THEN J1$=""
146 IF J1=46 AND JD=0 THEN JD=1+LEN(JS$)
148 IF J1=46 AND JD>JL-2 THEN J1$=""
150 IF J1<>46 AND JD=0 AND LEN(JS$)>JL-4 THEN J1$=""
152 RETURN
160 GET J1$: IF J1$="" THEN 160
163 J1=ASC(J1$):IF J1>127 THEN IF J1<193 OR J1>218 THEN J1=J1-128:J1$=CHR$(J1)
166 IF J1=13 OR J1=20 OR J1=60 OR J1=62 THEN RETURN
169 IFJ$="S"THEN IF J1<32 OR (J1>127ANDJ1<160) OR J1>223 ORJ1=34THEN J1$ = ""
172 IF J$="I" THEN IF J1<48 OR J1>57 THEN J1$=""
175 IF J$="N" THEN IF (J1 <> 46 AND J1<48)OR J1>57 THEN J1$ = ""
178 IF J$="N" AND J1=46 THEN JD= 1
181 IF J1$="" THEN 160
184 RETURN
190 IF JD=0 THEN JS$=JS$+".":PRINT".";:JD=LEN(JS$)
192 IF JD>LEN(JS$)-2 THEN JS$=JS$+"0":PRINT"0";:GOTO 192
194 IF LEN(JS$)<JL THEN FOR L = LEN(JS$)TOJL-1:JS$=" "+JS$:NEXT
196 RETURN
200 PRINT"[LEFT] [REVS]4[RVSO]";:RETURN
220 GOSUB 100: PRINT"[LEFT] ";: RETURN
247 REM
248 REM ** INPUT ROUTINE FOR STRINGS, INTEGERS, & 2 D.P. NUMERALS
249 REM
250 JS$="": JD=0: JH=FH%(FC): JV=FV%(FC): JL=FL%(FC): J$=FS$(FC)
253 GOSUB 100: GOSUB 200: GOSUB 160
256 IF J1=13 AND JS$="" THEN GOSUB220:GOTO250
259 IF J1=60 OR J1=62 THEN GOSUB300:GOTO250
262 IF JS$="" THEN FOR L = 1 TO JL:PRINT" ";:NEXT
265 IF JS$="" THEN FOR L = 1 TO JL:PRINT"[LEFT]";:NEXT
268 IF J1=13 AND J$="N" THEN GOSUB 190: GOTO 277
271 IF J1=13 AND J$="I" THEN GOSUB 194: GOTO 277
274 IF J1=13 AND LEN(JS$)<JL THEN FOR L = LEN(JS$)TOJL-1:JS$=JS$+" ":NEXT
277 IF J1=13 THEN GOSUB 220: RETURN
280 IF J1 = 20 THEN IF LEN(JS$) < 2 THEN PRINT "[LEFT] [LEFT]": GOTO 250
283 IF J1 = 20 THEN JS$ = LEFT$(JS$, LEN(JS$) -1):PRINT "[LEFT] [LEFT]";: GOTO 295
286 IF LEN(JS$)>=JL THEN J1$=""
289 JS$ = JS$ + J1$
292 PRINTJ1$;
295 GOSUB 120: GOTO 268
300 GOSUB 220
305 IF (FC=FL AND J1=60) OR (FC=FT AND J1=62)THEN RETURN
310 IF J1=60 THEN PRINTJ$(FC):FC=FC-1
315 IF J1 =62 THEN PRINTJ$(FC): FC=FC+1
320 RETURN
```

Parameterised crashproof 'INPUT' routine

```
2000 PRINT "[CLEAR][RVS] TITLE [RVSOFF]"
2010 PRINT : PRINT "   ENTER TYPE (A or B): "; J$(0)
2020 PRINT : PRINT : PRINT "    DATE: "; J$(1); J$(2); J$(3)
2030 PRINT : PRINT : PRINT "[RVS] ENTER FULL NAME:- [RVSOFF]": PRINT "    "J$(4)
2040 PRINT "[DOWN][DOWN] ...[DOWN] Check: Entry OK? "
```

This method is useful where repeat entry of data is wanted. If the data is one-off, or the previous values aren't carried over from entry to entry, the screen will be similar, but the expressions in J$() will be omitted, as J$(5) is here (because its only function is to wait for 'YES').

Finally, in addition to these preliminary routines, the actual input itself is made by a loop; this is necessary to permit free movement between fields during input. The example should make the process, and the inbuilt possibility of extra validation in addition to that by type, reasonably clear:-

```
1000 GOSUB 2000                                           :REM PRINT SCREEN
1010 FC=0: FT=5                                           :REM SET LOW/HIGH LIMITS
1020 GOSUB 250: OK=-1                                     :REM GET INPUT FROM SCREEN
1030 IF FC=0 THEN IF JS$<>"A" AND JS$<>"B" THEN OK=0:REM VALIDATE FIRST ITEM
1040 IF FC=1 THEN DD$=JS$                                 :REM DDMMYY ASSUMED HERE
1050 IF FC=2 THEN DM$=JS$                                 :REM VALIDATION ROUTINE CAN
1060 IF FC=3 THEN DY$=JS$:                                :REM BE USED (SEE 4.1.5)
1070 IF FC=4 THEN GOSUB 500      :REM SOME SORT OF VALIDATION, SETTING OK=0 OR -1
1080 IF NOT OK THEN GOTO 1020                             :REM REINPUT IF NOT OK
1090 IF FC=FT AND JS$="YES" GOTO 1500                     :REM EXIT AT BOTTOM OF SCREEN
1100 IF FC=FT THEN GOTO 1020                              :REM CARRY ON IF NOT "YES"
1110 J$(FC)=JS$                                           :REM STORE VALUE IN J$()
1120 FC=FC+1: GOTO 1020                                   :REM CARRY ON WITH NEXT ITEM
1500 REM CONTINUE PROCESSING WITH FULLY-CHECKED DATA
```

Single-character input fills RAM remarkable rapidly, so BASICs earlier than 4 will give trouble with memory-freeing if there are many strings in use. (See FRE, and Chapter 2). Suppose we input ABCD. Two sets of strings build up in memory, so RAM looks like this: ABCDDABCCABBAA, where each individual GET takes one byte, and each composite string takes up one more byte than it did previously. A little algebra gives $\frac{1}{2}n(n+3)$ bytes for a string of length n. So a 25-byte entry uses 350 bytes. At this rate, automatic FRE in memory occurs often. If this is a problem, as it may be when using BASIC<4, palliatives vary from restructuring the program so that data is held in RAM by poking and peeking, to holding several strings as one, separating out the individual strings with MID$ when they're needed. (If the number of strings is reduced to one-third of its previous value, garbage collection is about nine times faster).An alternative is to temporarily dissociate the bulk of string variables: In BASIC 2, this means the contents of ($34) are replaced temporarily by those of ($30), moving the 'top of memory' to the 'bottom of strings'. Only those variables used in the routine are affected by FRE, which is usually much faster. To recover the remaining strings, the original top of memory pointers must be replaced. The addresses in decimal are 52 and 53 ('top of memory') and 48 and 49 ('bottom of strings'). BASIC 1's pointers are different (see Ch. 15). *NOTE: see Ch. 17 for Commodore's 'Stondord doto entry environment'.*

4.1.9 The keyboard buffer is dealt with in Chapter 8, section 8.8. Chapter 5 also has some examples: see AUTO and DEL, amongst others. This example is a routine to convert machine-code into DATA statements, for later use as part of a machine-code loader. After the input of the start and end addresses – obviously necessary – and the starting linenumber, data statements are printed on the screen and incorporated in BASIC in direct mode. The key to the program is to note that line 60030's END does not actually end the program; a [HOME] and two Returns are forced into the keyboard buffer, and since the screen holds something like this:

```
63000dA169,0,133,148,169,32,133,2,165,0,
201,80,176,86,165,1,201,50,176,80,169
1= 63000+1: s= 847: E=903: goto 60000
```

on END, the cursor is homed and two returns entered; the effect is identical to that achieved by entering these three keys at the keyboard. Values are for BASIC>1.

```
1 print"[clear]DATA STATEMENT GENERATOR
10 input"start location";s
20 input"end location";e
30 input"linenumber";l
60000 print"[clear]"mid$(str$(l),2)"dA";:g=peek(54)+256*peek(55)
60010 forj=s to e
60020 ifpos(0)+peek(196)>77thenprint"[left] ":print"[home][down][down]l="l"+1:s="j":e="e
     ":goto"g
60030 ifpos(0)+peek(196)>77thenpoke623,19:poke624,13:poke625,13:poke158,3:end
60040 printmid$(str$(peek(j)),2)",";
60050 next
60060 print"[left] ":poke623,19:poke624,13:poke158,2:end
```

4.1.10 Numeral packing and unpacking is a space-saving measure, sometimes useful when disk space is limited. It is also rather time-consuming to implement, and slows down the program's running to some extent. Two complementary subroutines (next page) convert a numeral string (e.g. "12345"), held as NS$, into a packed form NP$, and vice-versa. In effect the number is stored to base 100. Lines 80 and 410 contain 32; the object of this is to avoid some codes, e.g. CHR$(0) and CHR$(13), which may

not store successfully.

1. UNPACK PACKED STRING NP$ INTO INTEGER NS$:

```
80 NS$ = ""; FOR L = 1 TO LEN (NP$): NI$ = STR$(ASC(MID$(NP$,L,1))-32)
82 IF LEN(NI$)<3 THEN NI$=" 0" + RIGHT$(NI$,LEN(NI$)-1): GOTO 82
84 NS$ = NS$ + RIGHT$(NI$,2): NEXT L: RETURN
```

2. PACK INTEGER STRING NS$ INTO PACKED STRING NP$:

```
400 NP$=""; IF INT(LEN(NS$)/2)*2 <> LEN(NS$) THEN NS$=CHR$(32) + NS$
405 FOR L = 1 TO LEN(NS$) STEP 2
410 NP$ = NP$ + CHR$(VAL(MID$(NS$,L,2))+32)
415 NEXT L
420 RETURN
```

DEMONSTRATION ROUTINE:

```
1000 INPUT NS$: GOSUB 400: PRINT"PACKED VERSION IS "NP$
1010 GOSUB 80: PRINT"UNPACKED VERSION IS "NS$
1020 GOTO 1000
```

PACKS NUMBERS OF FORMAT 99999.99 WITHOUT THE DECIMAL POINT:

```
480 NS$= LEFT$(ND$,5) + RIGHT$(ND$,2)
484 GOSUB 400
488 RETURN
```

4.1.11 Rounding is the process of converting and representing a number in a less accurate, but more convenient, form: $10 plus 15% is $11.50; $10.45 plus 15% is $12.0175; to two decimal places these are 11.50 and 12.02 respectively. (I have not considered the question of relative accuracy here, i.e. accuracy to a certain number of significant digits). A good rounding routine may format the number to a known length with leading spaces, insert (for example) '.00' after a plain integer, and put in a leading zero in the case of numbers less than 1. Poor routines may put the decimal point in the wrong place, produce spurious values, or print characters like 'E', on occasion. Alignment may be difficult, and zeroes not treated as a special case.

```
DEF FN P(X) = INT(LOG(ABS(X)+.001)/LOG(10))
```

is intended to calculate the number of places before the decimal point; but there may be very occasional errors in the calculations of the logarithms. This expression:

```
DEF FN R(X) = INT(100*X + .5)/100
```

rounds X to the nearest 2 decimal places: adding .5 has the effect of converting a number with decimal component greater than .5 into the next highest number on INT. This, on PRINT X, gives the usual 1.3 (not 1.30) for 1.3, and 1 (not 1.00) for 1.

The following more comprehensive routine is intended to round and format numbers as suggested above. Apart from intermediate variables, the routine uses L to store the number to be rounded, RQ ('rounding quantity') as a measure of accuracy, and L2 to determine the type of rounding. RQ=100, for example, rounds to 2 decimal places, and RQ=1000 to 3. When RQ=100, L2=.005 rounds to the *nearest;* L2=0 rounds *down;* and L2=.995 rounds *up.*

```
92 L=INT(L*RQ+L2)/RQ: JS$=STR$(L): JS$=MID$(JS$,2)
93 JL=LEN(JS$): IF JL>2 THEN IF MID$(JS$,JL-2,1)="." GOTO 96
94 IF JL>1 THEN IF MID$(JS$,JL-1,1)="." THEN JS$=JS$+"0": GOTO 96
95 JS$=JS$+".00"
96 IF LEFT$(JS$,1)="." THEN JS$="0"+JS$
97 IF LEN(JS$)<11 THEN FOR J=LEN(JS$) TO 10: JS$=" "+JS$: NEXT
98 RETURN
```

Line 92 computes a rounded string, without a leading space.
Line 93 branches on numbers like 123.45, 9999.99, 1.23, and .67.
Line 94 adds a zero to numbers like .5, 123.4, and 99999.9.
Line 95 converts integers to 2 dec. pt. form, e.g. 1234 into 1234.00.
Line 96 adds a leading zero to numbers like .5, .12.
Line 97 adds leading spaces up to a predetermined length (11 characters here).
The routine is intended for positive numbers > .01.

BASIC rounding routines always have a residual uncertainty about them, because the effects of rounding by the calculation routines aren't certain. Chapter 5's PRINT USING avoids this difficulty, since it edits the number before output; it is also faster. Whenever a rounding routine is to be used, unless it has been previously tested, it is good practice to write a test routine to generate numbers to be rounded; either at random or in a sequence. It is usually impossible to test each individual value.

4.1.12 RAM data storage has two forms: data may be poked and peeked in some fixed part of RAM, typically near the top, or it may be processed by arrays in the normal way, but differ from normal file-handling in being loaded and saved directly from RAM. The first method is useful in association with machine-code: a set of names, key numbers or indexes can be searched in RAM virtually instantaneously, cutting down on disk or tape use. The second approach also cuts down on input/ output, and, provided that the whole of a batch of data fits RAM, can lead to very efficient processing; for example, a 10K program can coexist with (say) 10000 integers stored in 20K of arrays, and both the program and data could be loaded from tape, providing economical processing of quite a large amount of data. The technique is fairly tricky. As we saw in Chapter 2, the program starts in RAM at $0400 and is followed by a block containing all the variables, string pointers, and function definitions so far encountered in the running of the program; after this comes a block of arrays and string array pointers. If we have integer arrays only, and if every variable is set up already, the position of the integer arrays is known, so that they can be saved and reloaded freely. Programs using this method will have a layout of this sort:

```
Set (or LOAD) pointers to the correct positions for variables and arrays
LOAD stored arrays of data
-----
Menu
-----
Menu option to save stored data to disk/ tape
-----
```

The first time round, with no variables in memory and no data yet on disk, a starting up procedure is necessary. This involves (a) entering all the variables in direct mode in optimum sequence, e.g. J=0:KK=0:IN$="". (b) Dimensioning all arrays. (c) GOTO the line after 'LOAD stored arrays of data'. The menu will be displayed, and all the variables are in place. The program must be STOPped to peek the pointers needed to save and to reload. If the program is edited, this process will have to be repeated, since the position of the data varies with the program length.
Section 4.1.17 has an example of this method in use.

4.1.13 Searching is necessary whenever a file structure provides no way of calculating the position of a record. Chapter 6 has a long section on disk files, which looks at this problem. With CBM disks, 'relative files' (accessed by record number) or direct access files (which must be specially written) enable a record to be found very rapidly; sequential files of any length are much slower. But often the record number of a relative file may not be known, or may be less convenient than (say) entering a name or phone number and waiting for the corresponding record to be read. Chapter 6 explains how such files may be subdivided, so the searching process is accelerated.
We may distinguish between searches in RAM and those which read data from disk. In the first type, machine-code searches are so fast that the data need not be ordered or arranged in any way. It is fast enough, normally, to scan from the start to the finish, without elaboration. Section 6.7 has a fairly long example, including both BASIC and a machine-code subroutine. However, when searching from disk, this may be too slow. As we saw in Chapter 3, under these circumstances a search which converges on the sought value is usual. The 'binary chop' is the best-known, and is easy to program. (The 'Fibonacci search' is faster, but less easily programmed). It requires that its data be in sorted order. This diagram shows how the convergence takes place:

ITEM NUMBER IN SEQUENCE:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
NUMBER OF SEARCHES TAKEN:	4	3	4	5	2	5	3	4	5	1	4	3	4	5	2	4	5	3	4	5

using the algorithm on the next page, and applying it to 20 items of data. We can calculate the average number of searches used by the binary chop, by amount of data:

NUMBER OF ITEMS OF DATA:	50	100	200	500	1000	2000	4000	9000
AVERAGE NUMBER OF SEARCHES:	5	6	7	8	9	10	11	12

```
x      Input and validate item to be searched for (say, K$ = key item).
       N1 and N2 set to current low and high record numbers
y      R = INT((N1+N2)/2)                        :REM CALCULATE NEW MID-POINT
       Read the appropriate field of record no. R; say R$
•      IF R$=K$ GOTO z                           :REM FOUND IT!
       IF N1>=N2 THEN PRINT "RECORD NOT ON FILE": GOTO x:REM NON-EXISTENT
       IF R$>K$ THEN N2=R-1: GOTO y              :REM REVISE UPPER LIMIT DOWN
       N1=R+1: GOTO y                            :REM REVISE LOWER LIMIT UP
z      Continue processing the record
```

This schematic program of the binary chop search is, I hope, self-explanatory. N1 and N2 converge, sandwiching the correct value of R between them. Note that records needn't be disk-based; they could as easily be a sorted array in RAM, in which case the test line would read IF R$(R)=K$ GOTO z. Try out this technique before implementing a large system, generating test-data with a program, and timing the result. It may be too slow, depending on the disk system and size of file.

4.1.14 Sorting is an important operation in commercial data processing. (COBOL has a SORT verb). Chapter 5 has a collection of routines, mostly in BASIC, with notes. The first example, the 'tournament' sort, is unlike all the others in computing individual results singly, so that results can be printed continually, before all the values are ordered. Most sorts wait until the entire batch of data has been ordered, and this can be irritating to wait for and slightly worrying, as the machine may appear to do nothing for long periods. The 'bubble' sort has achieved fame through being very slow. It operates by checking neighbouring values in the array, interchanging those which are out of sequence, and repeating this process until the sort is guaranteed, or until any pass takes place without a transposition, depending on the algorithm. That in Chapter 5 (section 5.3) has a test in line 620 which uses a 'finished' flag. The sort is assumed to be in ascending order, and after every pass another value is positioned at its correct value at the 'top' of the heap, unless, with a partly-sorted set of data, many items are simultaneously sorted. To illustrate the idea, seven figures in the left-hand column are shown sorted (in five passes) in the right-hand column.

4	7	7	7	7
7	4	6	6	6
1	6	4	5	5
3	1	5	4	4
5	3	1	3	3
2	5	3	1	2
6	2	2	2	1

Starting at the bottom of the set of data, each item is compared with its immediate neighbour and interchanged if it is out of sequence. The process is repeated to a distance up the data which depends on the previous number of passes; the underlined digit represents the top limit in each pass. With n items of data, a maximum of $n + (n-1) + (n-2) + \ldots$ passes is required, making about $\frac{1}{2}n^2$ in all. On this basis it is often said that the bubble sort takes time proportional to the square of the number of items to be sorted. However, the correct time is very sensitive to partial ordering of the data. The graph at the end of SORT shows that new items, added to an already sorted array, then bubble sorted together, is very fast; in fact, under these circumstances, the bubble sort is one of the fastest possible, since it does little more than check that each item is correctly related to its neighbour, which is necessary in any sorting system. The machine-code sort operates on string arrays, changing the pointers where appropriate, and using the identical comparison to that of BASIC, for consistency. It does not sort the zeroth element, which can therefore be used as a title or reminder. If new items are to be sorted in, keep a number of null or blank elements at the start of the array. As the diagram illustrates, high values (e.g. 6) can rise quickly from the bottom, but low values (e.g. 1) are slow in descending. Note finally that the machine-code can be made to sort from the second, third, ..., characters of the string, rather than the first, by changing $FF in $032E (BASIC 1), or $7FB6 (BASIC>1) to 0 (second), 1 (third),... A demonstration BASIC routine is provided with the machine-code. Of the other sorts, the Shell-Metzner and Quicksort are well-known; the former performs many small bubble sorts on longitudinal subsets of the data; the latter compares data with a 'pivot value', putting the result into one or other 'stack' depending on the result. It may run out of space; if so, dimension the array in line 40 with a larger value. The 'scatter' sort is an attempt to mimic human sorting: a subsidiary array is used, into which data is first roughly sorted, on some a priori basis, for example with the As at the beginning, Zs at the end, and others in between. Then this array is sorted thoroughly. Its use of RAM is too great to permit the method to be very useful on micros.

4.1.15 String handling. CBM BASIC has three closely related string functions, LEFT$, MID$, and RIGHT$, each of which extracts a substring from a string. Chapter 5 has examples of the use of each function, and an additional function INSTRING$, which helps illustrate machine-code string handling. Strings can be represented by variables or literals (e.g. X$ or "XYZ"), and also by the type-conversion functions CHR$ and STR$. Substrings can be concatenated (=chained) together with the binary operator '+', and in fact any conceivable rearrangement of strings is possible with + and the LEFT$, MID$, and RIGHT$ commands. In many cases, MID$ alone can be used. Note, however, that a string's length cannot exceed 255 bytes, because of the storage method used by BASIC. Typical string processing includes the following:

(i) The use of extended, composite strings. The components need not be the same length, but for ease of programming this is usual.

```
x$="ᵇᵇᵇSundayᵇᵇᵇMondayᵇᵇTuesdayWednesdayᵇThursdayᵇᵇᵇFridayᵇSaturday"
print mid$(x$,(d-1)*9+1,9)
```

Each substring is 9 bytes long (ᵇ represents one space character), because the longest component is "Wednesday". The second expression prints a substring of length 9 corresponding to the d'th day's name, where d=1 to 7.

(ii) Padding a string with leading or trailing spaces, so that alignment is automatic on printing out. The obvious way is to add individual spaces:

```
FOR J=LEN(S$) TO 19: S$="ᵇ"+S$: NEXT :REM PADS STRINGS OF LENGTH <20 TO 20
```

A quicker and more elegant way (which also uses less RAM, and is therefore better with BASIC<4) is to add the entire substring in a single chunk:

```
S$ = LEFT$("ᵇᵇᵇᵇᵇᵇᵇᵇᵇᵇᵇᵇᵇᵇᵇᵇᵇᵇ",20-LEN(S$)) + S$: REM PADS STRING TO LENGTH 20
```

(iii) Scanning a string for certain alphanumerics. In such activities as checking a response for accuracy in foreign-language (or English!) teaching, and playing hangman, a FOR ... NEXT loop can examine the string. Let's consider hangman, the word-matching game, where W$ is the target word, L$ a guessed letter, which, if it exists within W$, appears in the display D$. Typically, W$ will be selected by some such routine as this: RESTORE: FOR J=1 TO RND(1)*201: READ W$: NEXT: REM ASSUMES 200 WORDS Then D$ is generated with: D$=LEFT$("--------------------",LEN(W$)). This gives a string of hyphens of the same length as the target word. We now put: D$="ᵇ"+D$+"ᵇ", which is a slight subtlety, enabling us to use only single-line processing, without having to take account of special cases when the first or last letter has been selected. Now, for each letter L$,

```
FOR J=1 TO LEN(W$)
IF L$=MID$(W$,J,1) THEN GOSUB x: PRINT "[HOME]D$ :REM ASSUMES DISPLAY AT TOP
NEXT
```

W$ is scanned from beginning to end; if a match is found, the string D$ is revised and printed over its previous value. If a letter occurs several times in W$ the process repeats, but is fast enough for the process not to be visible. The subroutine which updates D$ has to insert L$ within D$ at the correct position defined by variable J:

```
x   D$=LEFT$(D$,J)   + L$ + RIGHT$(D$,LEN(D$)-J-1):  RETURN
```

(iv) Note on BASIC 4: A rare bug may occur when concatenating more than two strings, and when fewer than $300 bytes of RAM are free; the string is corrupted.

4.1.16 Validation is the process of checking that data is of the correct type, without necessarily guaranteeing the actual value. A date 19/19/82 is invalid, but if it is accepted may cause processing errors, and so will be rejected by most systems. The date 3/5/82 is valid, but may not be correct. Similarly, '20' may be an acceptable entry for a sum of money, but 'twenty' may not.

The simpler forms of validation repeat the request for data in the event of an incorrect entry:

```
100 INPUT "DISK DRIVE NUMBER"; D$        :REM D$ WILL ACCEPT ANYTHING
110 D=VAL(D$): IF D<>INT(D) OR D<0 OR D>1 GOTO 100 :REM INTEGER 0 OR 1 ONLY.
```

More sophisticated checking may include error messages (see 4.1.6) and soft-coding to enable acceptable entries to be modified. This batch of subroutines has tests for four variables, and was used with a crashproofed INPUT routine:

```
500 if js$="Y" or js$="N" then return
503 ok=0:em$=" Y or N only" :gosub 800: return
510 ni$="ABCDEFGJKMPTVWX": for I=1 to len(ni$):if js$=mid$(ni$,I,1) then return
513 next: ok=0:em$="in sales code":gosub800:return
520 ni$="04123": for I=1 to len(ni$): if js$=mid$(ni$,I,1) then j5=I: return
523 next: ok=0: em$="in VAT code":gosub800: return
530 if (asc(js$)>192 and asc(js$)<219) or asc(js$)=32 then return
533 ok=0: em$="in Foreign code":gosub800: return
```

4.1.17 Arrays (subscripted variables) provide a powerful extension to the usual system of simple variables, and are well worth mastering for any serious application. The principle is to provide a whole series of strings or numbers with a single name, using a subscript to distinguish the separate elements. Chapter 5 (see DIM) has information on the use of arrays; Chapter 2 explains their storage methods and the pointers which keep track of the data. Arrays of numbers, subject to their own rules of addition, subtraction and multiplication, are called 'matrices': see Chapter 16 on this. We can think of arrays as belonging to one of two classes: 'one dimensional' and 'multi-dimensional'. The latter are conceptually more difficult, so it makes sense to start with the first type:-

One-dimensional arrays are variables with a single subscript, which may take any value from 0 to the dimension of the array in DIM. (If no DIM statement was used, a default value of 10 is assigned). Unless an item is specifically assigned a value, it will be stored as 0 (numeral) or the null character (string array). The array can be visualised as a set of consecutively-numbered pigeon-holes, which are filled with a data-item, numeric, integer, or string, by the usual methods of assignment.

```
10 INPUT N: DIM A$(N): FOR J=0 TO N: INPUT A$(J): NEXT
```
inputs the size of the array, then a series of elements to fill it, and can be regarded as the array version of INPUT. Similarly the stored results can be output by
```
20 FOR J=0 TO N: PRINT A$(J): NEXT
```
A typical application of these arrays is the *look-up table*. For example, an array might hold opcodes for machine-code: A$(0)="BRK", A$(1)="ORA", and so on. Then there is a simple relationship between a peeked value of a location (say, P) and the string A$(P). A numeric array could hold the values of the locations of the start of each line on the screen; DIM L(24) could hold each value from 32768 up. Then the location of the ninth character along line fifteen is L(15)+9. A fifty-two element array might hold all the cards in a pack. As mentioned in Chapter 5, the zeroth element can be reserved for special purposes, typically for averages or totals. Other uses include the storage of values for sorting. The sorts in Chapter 5 all operate on string arrays, which could consist of a key (name, catalogue number, reference) followed by a relative-file record number. An array variable is slower to process than a simple variable, because of the processing overhead associated with its subscript. Nevertheless, access is faster than some calculations and function evaluations, so look-up tables are sometimes used to speed up programs which contain repetitive calculations on a limited range of arguments. For instance, it may be worthwhile to set up a table holding present values of money over a number of years, or of square roots from 1 to 100.

Arrays are useful in games and problems of the board-game or rectangular grid type, and we can use this topic as a bridge to multi-dimensional arrays. Ingenious applications of single-dimension arrays where more dimensions appear appropriate include the '8 queens' problem, where the object is to arrange 8 chess queens on a chessboard so that none attacks any other. An array of only 8 numbers can represent the board; each value must be different, and from 1 - 8 to denote the position of that column's queen. Diagonals are tested by a difference method which the diagram illustrates, the first example passing all tests and the second having two attacking queen pairs:

6 4 1 5 8 2 7 3		4 7 3 8 1 5 6 2

Another ingenious algorithm is that for assessing card strengths in five-card poker: the hand is sorted, and the four consecutive differences evaluated. Of these, there are only three of importance: 0,1, and any other value, corresponding respectively to pairs (or threes or fours), straights, and others. The 3^4 (=81) possible values can be assessed by an array. Chess games are usually stored as an 8 by 8 array, pieces being represented by a positive or negative number (representing colour) of value related to the importance of the piece.

Multi-dimensional arrays have more than one subscript; the maximum is 255. It is always possible, though inconvenient, to simulate such arrays by partitioning single-dimension arrays, so there are BASICs which permit only one subscript. A simple two-dimensional example shows how the contents of the array dimensioned by DIM A$(1,7) might be stored:

A$()	0	1	2	3	4	5	6	7
0	"POSITIVE"	"HOT"	"ON"	"LARGE"	"HIGH"	"WARM"	"CALM"	"WELL"
1	"NEGATIVE"	"COLD"	"OFF"	"SMALL"	"LOW"	"COOL"	"ROUGH"	"ILL"

So that INPUT A$(0,3) had taken in LARGE from the keyboard, or been assigned in a program, and PRINT A$(1,7) prints the word "ILL". Note that an array with *n* dimensions usually requires *n* nested loops to input or output all its data.

These arrays are valuable for storing data for business reports, as the example shows. The schematic BASIC routine demonstrates the logic which was used to generate the reports (which are incomplete here, for reasons of space restriction). It should be self-explanatory. The only subtle point is the use of an additional code of each type; this is an overflow or 'wastebasket', into which unrecognised items are put. In each case the contents of this extra, non-existent code should be zero. For example, if a sales code had somehow been recorded as "%", J would take the value 15 on leaving line 100.

```
FOR ALL RECORDS: READ SALES CODE S$, ORIGIN CODE O$, AGE CODE A (1-8), VALUE V
    100 FOR J=1 TO 14: IF S$<>MID$("ABCDEFGJKMPTVX",J,1) THEN NEXT
    110 SA(J,A) = SA(J,A) + V
    120 FOR J=1 TO 10: IF O$<>MID$("BCFGHOPQSU",J,1) THEN NEXT
    130 O(J) = O(J) + V: O(0)=O(0) + V
NEXT
FOR J=1 TO 15: FOR K=1 TO 9: SA(J,0)=SA(J,0) + SA(J,K): NEXT: NEXT
```

At the end of this process, array SA() holds values by sales code and age code, and O() holds the same values by origin code. Totals are held in the zero elements.

TOTALS BY SALES CODE AND AGE CODE

SALES CODE: A	10343.00	SALES CODE: B	15275.71	SALES CODE: C	38916.11	SALES CODE: D	798.42
Age Code: 1	8152.35	Age Code: 1	10720.77	Age Code: 1	28721.49	Age Code: 1	507.24
Age Code: 2	1256.08	Age Code: 2	3128.44	Age Code: 2	5296.83	Age Code: 2	152.08
Age Code: 3	337.19	Age Code: 3	541.57	Age Code: 3	3025.52	Age Code: 3	139.10
Age Code: 4	158.49	Age Code: 4	365.40	Age Code: 4	662.46	Age Code: 4	0.00
Age Code: 5	388.40	Age Code: 5	490.01	Age Code: 5	1111.06	Age Code: 5	0.00
Age Code: 6	50.49	Age Code: 6	29.52	Age Code: 6	98.75	Age Code: 6	0.00
Age Code: 7	0.00	Age Code: 7	0.00	Age Code: 7	0.00	Age Code: 7	0.00
Age Code: 8	0.00	Age Code: 8	0.00	Age Code: 8	0.00	Age Code: 8	0.00
Age Code: 9	0.00	Age Code: 9	0.00	Age Code: 9	0.00	Age Code: 9	0.00

SALES CODE: E	0.00	SALES CODE: F	20185.51	SALES CODE: G	1613.80	SALES CODE: J	18237.68
Age Code: 1	0.00	Age Code: 1	15037.31	Age Code: 1	1592.80	Age Code: 1	12303.31
Age Code: 2	0.00	Age Code: 2	3302.80	Age Code: 2	21.00	Age Code: 2	3232.12

SUMMARIES BY SALES CODE, ORIGIN CODE & AGE CODE

SALES CODE: A	10343.00	ORIGIN CODE: B	157.01	AGE CODE: 1	89536.95
SALES CODE: B	15275.71	ORIGIN CODE: C	223.76	AGE CODE: 2	19006.55
SALES CODE: C	38916.11	ORIGIN CODE: F	2527.49	AGE CODE: 3	8255.29
SALES CODE: D	798.42	ORIGIN CODE: G	0.00	AGE CODE: 4	1892.06
SALES CODE: E	0.00	ORIGIN CODE: H	0.00	AGE CODE: 5	3473.58
SALES CODE: F	20185.51	ORIGIN CODE: O	59815.17	AGE CODE: 6	317.53
SALES CODE: G	1613.80	ORIGIN CODE: P	36286.15	AGE CODE: 7	0.00
SALES CODE: J	18237.68	ORIGIN CODE: Q	288.00	AGE CODE: 8	0.00
SALES CODE: K	173.60	ORIGIN CODE: S	13666.17	AGE CODE: 9	0.00
SALES CODE: M	16313.35	ORIGIN CODE: U	9518.18	AGE CODE:	0.00
SALES CODE: P	546.59	ORIGIN CODE:	0.01		
SALES CODE: T	0.00				
SALES CODE: V	0.00				
SALES CODE: X	78.17				
SALES CODE:	0.01				

TOTAL BY SALES CODE: 122481.95   TOTAL BY ORIGIN CODE: 122481.95   TOTAL BY AGE CODE: 122481.95

Two-dimensional arrays may be used to store quite large quantities of data (about 32K less the space occupied by BASIC) very efficiently. Integer arrays, which store numbers from -32768 to 32767 in only 2 bytes, are particularly efficient. They can be saved and reloaded *en bloc* to disk, providing rapid access to a lot of data with little disk drive use. To understand the approach, read the next few paragraphs carefully.

The example we'll consider is a garment inventory system. Its volumes of data are: 50 cloth types, identified by a four-digit number.

Each cloth is available in 1 to 12 colours; the average is about 4.

Each cloth/ colour combination has 1 to 8 styles of garment; that is, a cloth in blue may be made into only one type of jacket; the same cloth in brown may be made into two other designs.

Each garment is produced in six sizes.

At each level of complexity, details about the cloth or the clothes are stored; for example the cloth width is recorded for every cloth type, and, at a more detailed level, the quantity in stock of every size of each garment is required.

We can store the data in arrays like this:

C%(50,4)

	0	1	2	3	4
0	3	0	0	0	0
1	1101	2	1	60	2
2	1149	5	12	36	7
3	2065	3	10	60	10
4	0	0	0	0	0
..					
50	0	0	0	0	0

M%(200,3)

	0	1	2	3
0	0	0	0	0
1	3	200	2	2
2	4	50	3	5
3	1	100	1	6
4	2	200	1	7
5	3	250	4	11
6	7	800	6	17
7	10	100	3	20
8	6	200	2	22
9	6	200	2	24
..				
200	0	0	0	0

S%(800,2)

	0	1	2
0	0	0	0
1	99	121	6
2	40	210	12
3	24	022	18
4	18	220	24
5	13	004	30
6	50	001	36
7	54	001	42
8	53	001	48
9	58	001	54
10	63	001	60
..			
800	0	0	0

SZ%(4800)

0	0
1	44
2	17
3	8
4	23
5	90
6	12
7	10
8	10
9	10
10	12
11	25
12	23
..	
4800	0

$C\%(0,0)$ = Total number of cloths entered
$C\%(n,0)$ = Cloth number
$C\%(n,1)$ = No. of colours
$C\%(n,2)$ = Delivery date
$C\%(n,3)$ = Cloth width
$C\%(n,4)$ = Cumulative no. of colours

$M\%(n,0)$ = Colour codes
$M\%(n,1)$ = Length stored
$M\%(n,2)$ = No. of styles
$M\%(n,3)$ = Cumulative no. of styles

$S\%(n,0)$ = Telephone orders
$S\%(n,1)$ = Style code
$S\%(n,2)$ = Cumulative no. of sizes

$SZ\%(n)$ = inventory+ some other data

Together, these arrays occupy 16160 bytes (including the array overheads. See Chapter 2). The brackets show the way in which one array is dependent on the earlier array. The details, once set up, are difficult to alter, because all the subsequent details are stored immediately after, leaving no room for manoeuvre. C%(0,0) currently holds 3, showing that only three cloths' data has been keyed in so far. When the next cloth is entered, the fourth row of C%() will fill, 1 - 12 rows of M%() will depend on this, and a maximum of 96 rows in S%() may be filled in turn. Finally, SZ%() has from 1 to 576 elements filled. The cumulative frequency pointers (which are not strictly necessary) make this scheme fairly easy to implement. However, BASIC programs which store data like this are amongst the most difficult to decipher of any BASIC, the problems increasing with the number of arrays. Whether this is undesirable depends on one's point of view. Some short extracts from programs show the type of program to expect:

```
S%((M%((C%(N-1,4) + M - 1),4) + K),0) = P   : REM TELEPHONE ORDERS
SZ%(S%(J + K - 1,1) + U) = SZ%(S%(J + K - 1 ,1) + U) + S : REM UPDATE STOCK POSN
PRINT M%(C%(N-1,4) + M ,1) : REM PRINTS AVAILABLE STOCK
```

Multi-dimensional arrays with more than two dimensions are not used much, probably because of the difficulty of visualizing the data's storage pattern within its arrays. The diagram (right) illustrates a three-dimensional array, set up by the statement DIM X(15,20,3). Since zero elements are allowed, the array's 'pigeonholes' occupy 16 by 21 by 4 locations. Assuming a conventional order or rows, then columns, then depth, leads to the diagram, in which for instance X(0,0,0) is the top-left element in the 2-dimensional array on top of the heap, and X(1,2,3) occupies row 1 and column 2 of the array at the bottom of the heap. Four-dimensional arrays can be pictured as several stacks of three-dimension arrays arranged side-by-side. After this, depiction becomes progressively more complicated. The maximum number of dimensions is 255 (see Chapter 2). In practice, shortage of RAM will make this figure, or anything like it, impossible. Section 2.3 of Chapter 2 explains the calculations necessary to determine the total number of bytes taken up in RAM by any array.
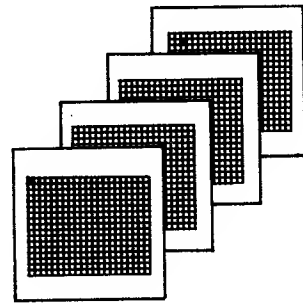
4.2 Debugging BASIC programs

This section lists common faults in BASIC programming. While such a list cannot hope to be exhaustive, it should help in pinpointing errors.

Peculiarities of BASIC. These include a few bugs.

ASC of a null character doesn't evaluate as 0, but crashes.

CLOSE doesn't properly close an IEEE file without PRINT# to the file.

DATA statements may give trouble if new DATA statements are inserted before them.

FOR ... NEXT occasionally behaves oddly: see Chapter 5 on this.

FRE may be slow in BASICs 1 and 2. See for example Chapter 2.

INPUT crashes on Return; also input from a file prints no warning message if extra data (e.g. separated by commas) has been read in.

LEFT$, RIGHT$ may crash if the numeric part of the argument is 0.

PRINT attempts to print anything it is given; a stray '.' appears as 0, for instance.

.S saves machine code, omitting the final byte; so add 1 to the end address.

STR$ introduces a leading space into positive numbers.

TAB and SPC have some quirks carried over from CBM BASIC's ancestors.

Numerals are held and formatted to a certain degree of accuracy; see Chapter 13.

Strings have a maximum length of 255; attempts to exceed this give ?string too long.

Some mathematical functions will not accept certain values without error.

CBM disks: see the end of Chapter 6 for a summary of possible bugs.

Differences between BASIC ROMs are outlined in Chapter 2 and explained elsewhere in detail. BASIC 4 disk commands, and SYS calls to ROM, are nearly always incompatible between BASICs.

Syntax errors are usually fairly self-explanatory. These cases may be difficult:

(i) Included keywords. Misprints are particularly easy with logical constructions, because these are largely alphabetic. IF A=B OF C=D reads IF A=BO=D for example.

(ii) ?OUT OF MEMORY has diverse causes:-

    i. Too many levels of brackets, especially within loops and subroutines.

    ii. Absence of POP causing RETURNs to build up on the stack. See Chapter 5. Example: IF ASC(IN$)=27 THEN POP:GOTO MENU correctly aborts input.

    iii. Insufficient RAM, especially with large arrays.

    iv. Can occur when start and end of program pointers are altered.

Incorrect processing, without Syntax error indication is often caused by one of these:-

(i) Variable name repeated by mistake. See Chapter 2's variable name list.

(ii) Variable value changed in error. Typically FOR L=1 TO 10: GOSUB 100: NEXT

(iii) Wrong meaning of a statement. Very common with logical expressions.

(iv) Subroutines may be poorly structured, so program flow drops through.

(v) Omission of 'FN' will cause a function to be read as an array. Example: PRINT FN DEEK(X) mistyped as PRINT DEEK(X) is interpreted PRINT DE(X).

Errors caused by assuming a software setup appear when a program is re-run but not preceded by a setting-up program; examples include failure to specify the screen character set, failure to change memory pointers, failure to send control commands to the printer, and sometimes the use of LOAD within a program. Operators accustomed to a rigorous input validation may not adapt to the occasional use of INPUT.

Systematic, recurrent errors are usually caused by faults in the logic of programs:

(i) The zeroth or last entries in buffers may be omitted or misplaced.

(ii) Graphics or data-storage POKEs may change strings, variables, BASIC, or machine-code.

(iii) Keyboard entries at the wrong time or of the wrong sort may corrupt data, for example where an ESCape key allows exit from any routine back to the menu.

(iv) The logic of (say) a merge may be faulty in special cases. Identifying these may be difficult, requiring a painstaking dry run through the code.

Hardware problems can be detected by test programs. But during the course of running programs, trivial hardware problems may be overlooked:

(i) Shift-lock on causes the screen appearance of inputs to be odd, and may cause apparently valid key entries to be rejected.

(ii) A printer may lack paper or ribbon, or not be online, and so fail to function. It may be wrongly set.

(iii) Disk drives may be off or disconnected.

CHAPTER 5: ALPHABETIC REFERENCE TO BASIC KEYWORDS

This chapter lists all CBM BASIC keywords with explanations, examples, notes, and details of their operation at machine-code level. It should be useful to the learner, and also provide a convenient source of reference to experienced programmers who wish to check up on programming queries of the sort which inevitably arise in the course of writing programs. I have occasionally drawn attention to differences between CBM BASIC and other dialects of BASIC. The format of the explanations is roughly consistent for each keyword, which appears in bold type at the top of the page. Normal type indicates 'keywords' which are not present in CBM BASIC, but which can be written for it or adapted from other sources, or obtained in software form or as plug-in EPROMs. BASIC 4's specifically disk oriented keywords are listed in Chapter 7.

Note on BASIC operators.

When a string expression or arithmetic expression is evaluated, the result depends on
(a) the priority assigned to each operator, and
(b) the presence of parentheses.
Parentheses, in either string or arithmetic calculations, have the effect of ensuring that the entire expression within parentheses is evaluated as a unit. In the absence of parentheses, priority is assigned to operators in this order, starting high:

↑	**Power**
+ -	**Unary plus and minus**
* /	**Multiply and divide**
+ -	**Binary plus and minus**
< = >	**Comparisons - less than, equal to, greater than**
NOT	**Logical NOT - unary operator**
AND	**Logical AND - binary operator**
OR	**Logical OR - binary operator**

The *orithmetic* operators are relatively familiar and straightforward. Note the high priority of unary plus and minus; the point of this is illustrated by expressions like:

$$2↑-4*3 \quad \text{and} \quad 6 + - 3 \quad \text{and} \quad -1234 * - 2345 ,$$

which otherwise are meaningless. CBM BASIC evaluates a 'true' statement as -1, and a 'false' statement as 0. These are not standard between computers; Apple for example has true = 1, and other differences in interpretation. CBM *comparisons* are straightforward with numerals, but less so with strings, which are compared as far as the shorter string. So "1" as a string is < "10", but also "5" is > "449". CBM BASIC's *logical* operators use a 16-bit, 2-byte system; this means that 'true', which is printed as -1, is held as #FFFF. The maximum range of arguments for logical expressions is therefore -32768 to 32767. PRINT NOT 32768, for example, gives an error. Because NOT flips the 16 bits of the argument, X plus NOT X always add to -1, so NOT 10 is -11.
    It is important to realise that the *lower* priority operators have the largest sphere of influence, as it might be called. Ordinary arithmetic illustrates this in many ways: 2x + 1 is immediately seen to be twice x, plus 1. With the less common logical and comparison operators, this is rather easier to forget. See for example note [3] to AND.

ABS

BASIC arithmetic function

PURPOSE: Computes the absolute value of the arithmetic expression in parentheses following ABS. In other words, ABS makes a negative number or expression positive. This function has some applications in programming with numbers; it is not a major feature of BASIC.

Syntax: ABS(arithmetic expression). A string expression, or incorrect arithmetic expression, will generate one of a number of errors, including syntax, type mismatch, and division by zero errors. An expression which, when evaluated, is too large, causes an overflow error. Like all functions, ABS can appear on the right of an assignment statement, within a PRINT statement, and as part of a logical expression, for example after IF.

Modes: Both direct and program modes are valid.

Examples:
```
IF ABS(QTY) > 10000 THEN PRINT "*": REM PRINT WARNING ASTERISK
X = -12.5 + .5: PRINT ABS(X)      : REM PRINTS 12
1000 IF ABS(X - X1)< 1E-6 THEN PRINT "FINISHED": END
2000 Z% = ABS(10*SIN(X)): REM Z%=INTEGRAL PART OF ABSOLUTE VALUE
10220 IF ABS(AX%-BX%)<4 AND ABS(AY%-BY%)<4 GOTO 10200: REM FETCH BETTER START POSNS
```

The first example prints an asterisk if variable QT exceeds 10000, or if QT is negative with magnitude larger than 10000, such as -25342.3.

The third example shows how to test for approximate equality; this may be very useful when allowing for rounding errors and when performing iterative calculations which converge to some correct value. In this example, the value is accepted if the maximum error is 1E-6 (.000001). Typically, the more exact the precision, the longer such a program will take to run.

Fourthly, Z% in line 2000 takes integer values 0-10 only, in a pattern resembling a rectified sine curve. The very last line is taken from a game in which each player has a 'worm' to control on the screen; this line ensures that the starting positions of player A and player B, which are generated by the RND function, are not too close together.

Abbreviated entry: aB

Token: $B6 (182)

Operation: The expression in parentheses is evaluated and checked, and if valid put into floating point accumulator #1. ABS operates only on the sign byte of this accumulator. In fact ABS does less work than any other function. The sign byte (location $63, or $B5 in BASIC1) is shifted right, so that the negative (high) bit is not set. It does this whether or not the byte was negative. As far as further calculations are concerned, the number is positive. There is no loss of accuracy in this conversion inside the accumulator, but as with all numerical expressions, there may be a loss so far as the initial evaluation process is concerned. That is, ABS(-123456789012) and ABS(123456789012) are identical, but don't retain all the figures of the original arguments.

ROM entry points:

```
BASIC 1: $DB2A (56106)
BASIC 2: $DB64 (56164)
BASIC 4: $CD8E (52622)
```

AND

BASIC binary logical operator

PURPOSE: Calculates the logical AND of two expressions which are first converted into 2-byte integers. The result is itself a 2-byte integer. If the expressions were logical, the values 0 ('false') and -1 ('true') obtain, so the truth-value of a multiple condition can be found.

Syntax: Arithmetic or logical expression AND arithmetic or logical expression. Both expressions must be integers within the range -32768 to 32767, or floating point numbers which round down to within this range. Logical expressions invariably fall within this range, since they take values of -1 or 0 only. Out of range values, string expressions, and syntax errors in either of the two expressions will cause an appropriate error message to be printed to the screen.

Modes: Direct and program modes are both valid.

Examples: PRINT 380 AND 75
100 IF D%>0 AND D%<100 THEN PRINT "WITHIN RANGE 1-99"
6260 OK = -1 AND Y>79 AND Y<90 AND M>0 AND M<13
146 IF J1=46 AND JD=0 THEN JD=1+LEN(JS$)

The first example is a straightforward 16-bit AND between two numerals. The values and their bit equivalents are 380 (=%00000001 01111100) and 75 (=%00000000 01001011), so 380 AND 75 is evaluated by CBM BASIC as %00000000 0100100 or 72.

The second example shows AND used in a composite test; both parts of the test must be true to print the message.

The third example is a simplified part of a date validation subroutine. The object is to check that the decade is the 80s and the month within the usual range. OK is set to 'true', ANDed with four separate tests, each of which must be true if OK is to remain true.

Finally, another example of a composite test: this line, from a very long input routine, accepts decimal numbers which it build into a string JS$. J1 is the ASCII value of the last key pressed; JD is the position of the decimal point, or zero if no decimal point has yet been input. The example tests for the truth of two conditions: if the decimal point (ie full stop, with ASCII value 46) has been typed at the keyboard, and also this key is an acceptable one, then the decimal point's position in JS$ is fixed.

Notes: [1] The truth table for AND is:-

AND	T	F		AND	1	0
T	T	F		1	1	0
F	F	F		0	0	0

Where 1='true' or 'bit set on', 0='false' or 'bit set off'.

Note that when stored as 2-byte signed integers, false =0 =$0000, whereas true =-1 =$FFFF. (To convert $FFFF into its positive equivalent, flip the bits and add 1. This method gives $0000+1, so $FFFF is -1). This is why AND with a false expression is always false, while AND with a true ex-pression leaves the value unaltered. It is also the reason that NOT-1 is 0 and vice versa.

[2] Hierarchy. BASIC order copies FORTRAN and ALGOL. NOT then AND then finally OR have the lowest priority of all the operators. AND is therefore processed last in many cases.

[3] Common bugs: logical expressions are quite tricky; errors are compar-atively easy to overlook. Because of this four examples of typical wrong statements follow:

[i]   DE = D + DM + 365*Y + INT(Y/4) - (INT(Y/4)*4=Y) AND M>2

This is taken from a routine to find the weekday. The day, month and year are combined mathematically into a parameter taking only the values 0-6. In the example, the final expressions are intended to subtract 1 should the

year be a leap year <u>and</u> the month be March to December. But because of the low priority given to AND, if M is 1 or 2, the entire expression evaluates as 0. Everything after '=' and before AND is calculated, but this result is then ANDed with 0. This shows the power of a low priority command which could be compared - perhaps a little fancifully - to a recessive gene. The correct version has the joint expression enclosed in another set of parentheses.

[ii] `IF INT(Y/4)*4=Y AMD M>2 THEN DE=1`

Logical operators have relatively few syntactical requirements and so, if mistyped, are difficult for the translator to distinguish from variables. The line when run will not, as might be expected, cause a ?SYNTAX ERROR message. Instead it is interpreted like this:

`IF INT(Y/4)*4=YA>2 THEN DE=1`

and its run-time behaviour will depend on whether YA exists.

[iii] `IF PEEK(C+1) AND PEEK(C+2)=0 THEN END: REM END OF PROGRAM REACHED`

Failure to fully specify all the conditions is a source of bugs; the example is supposed to find two zero bytes at the end of a BASIC program stored in RAM. What is needed is this:

`IF PEEK(C+1)=0 AND PEEK(C+2)=0 THEN END`
or:

`IF PEEK(C+1) + PEEK(C+2)=0 THEN END`

The incorrect version will stop whenever PEEK(C+2) is zero and PEEK(C+1) is non-zero.

[iv] `IF J<1 AND J>8 THEN`

Never happens!

Abbreviated entry: aN

Token: $AF (175)

Operation: Binary operators are evaluated with the first argument in floating point accumulator #1, and the second in accumulator #2. AND uses exactly the same routine as OR, except that on entry a test location is loaded with zero. (OR loads it with #$FF). This is the only difference between these routines. Each accumulator in turn is converted into a 2-byte integer, and the low and high bytes are processed separately. Using 'TEST' to refer to the byte in the test location, the routine computes this function:
TEST EOR ((TEST EOR A) AND (TEST EOR B)).
When TEST is #$00, EOR TEST has no effect, so

```
A AND B =     (      A  AND       B)
A   OR B = NOT( NOT A  AND  NOT B)
```

All ROMs process this instruction in the same way.

ROM entry points:

```
BASIC 1: $CED9 (52953)
BASIC 2: $CECB (52939)
BASIC 4: $C089 (49289)
```

APPEND

System command unavailable directly in CBM BASIC

PURPOSE: Links two programs end-to-end into a single program. This can be very helpful in adding standard subroutines or BASIC utilities such as cross-referencers onto a BASIC program.

> *NOTE: APPEND in the sense used here applies to BASIC programs only, not files of data, and may be run on any CBM machine, irrespective of whether or not it is equipped with disks.*

Versions: Appending one program onto another requires that the linenumbers do not overlap; if they do, a program with lines 10 20 30 and 50, say, which has another program with 40 50 and 60 appended to it will appear as one program linenumbered 10 20 30 50 40 50 60. If the routines aren't too long, they can be listed on the screen and incorporated into the main program by loading it, homing the cursor, and entering the lines remaining on the screen. Longer routines would require a boring, but reliable, process of repeatedly loading the routine to be appended, loading the program, adding new lines, and saving the result so far. This process gives a MERGE, not an append; a merge is often potentially more use than an append, but is harder to implement.

Amongst the versions that have been written are several for tape: Jim Russo and Henry Chow's 'Merger' (Pet User Notes, Nov-Dec '78) and Roy Busdiecker 'Universal Tape Append' (Compute! Mar '81) are two. They use the same method, namely loading the second tape to start at the end address of the first program. From the users's point of view this is fairly nice and easy; all you do is press 'play' twice. The routine to be appended must be at the start of another tape, or at a known position. Between these two versions' publishing dates, a lot has happened, and much of Busdiecker's article is concerned with variations between ROMs. Disk versions are less sophisticated usually, because the header is more difficult to get at. For example CPUCN 2#5 has a 30-line program which reads a program, writes it as data, reads the nexts program, and writes it to the same file. See Chapter 6 for details.

The version below uses a different principle, and will append programs from different sources and recorded on different machines. The program to be appended - i.e. added onto the end - is loaded first. Then a SYS command moves the entire program up memory into the high end of RAM, as indicated by the pointer; so protected machine-code is untouched. Then the main program is loaded, and a second SYS command shifts the first program back to connect with the second. The program also rechains the BASIC lines, so that the link addresses are correct. I have included an ?OUT OF MEMORY indication if the programs together are too large. The USR locations 1 and 2 store temporary pointers, so if you're using USR, these will need resetting.

Examples.

[1] Load the append program, then run it, so that cassette buffer #1 holds the machine-code. Now enter SYS 634. This moves the append program itself into high memory.

Type NEW and enter 100 PRINT "HELLO".

This short program is held in the ordinary BASIC part of RAM starting at $0401.

SYS 673 will move APPEND down again from its position higher in RAM. It will be positioned correctly and chained, so that on LISTing you'll see line 100 at the start of the program, which runs normally, apart from briefly printing "HELLO".

Don't RUN a program between the two SYS commands, as strings may corrupt the part of memory storing the program to be appended.

[2] Load any program; type SYS 634. (Both SYS commands can be used repeatedly without reloading). Now load any other program (or the same one again!) and enter SYS 673. The new composite program should be correctly linked and should run as one program. If you type SYS 634 again, the new program will move up memory and a further program can be inserted at the start.

Notes: [1] ROMs. The BASIC loader is set up for the upgrade ROM (BASIC 2). BASIC 4 shares pointers with BASIC 2, and is therefore an identical routine, except for two absolute addresses. The data statements finish with two jumps ($4C = 76 decimal); one rechains the appended programs, the other prints the out-of-memory message when an append is impossible. BASIC 4 requires JMP $B4B6 and JMP $B3CD in place of the upgrade ROM routines. So 80 DATA .... 76,182,180,76,205,179 is correct for BASIC 4.

BASIC 1 ('Old ROM') needs pointers from 40-53 decimal to be changed to 122-135.

[2] Cassette Buffers. With BASIC 4 in mind, I've written the routine to load into cassette buffer #1, which is untouched by BASIC 4's disk handling. If loading is to be done from cassette #1, this buffer will of course be overwritten, so the machine-code must be loaded elsewhere, the obvious place being buffer #2. The code relocates, so substitute 826-864 for 634-672, and 865-934 for 673-742.

BASIC 2 APPEND ROUTINE:-

```
0 POKE 59468,12: PRINT "[CLEAR]$$$$$$" :REM UNDERLINE (SHIFT-$) TIDIES TITLE
1 PRINT "[REVS]APPEND": PRINT "[DOWN]MACHINE CODE IS NOW LOADED INTO SYS 634 AND SYS
  673.
2 PRINT "[DOWN]LOAD THE PROGRAM TO BE APPENDED;  ENTER [REVS]SYS 634[RVSO] ";
3 PRINT "TO STORE IT HIGH UP IN MEMORY.
4 PRINT "[DOWN]LOAD THE MAIN PROGRAM AND ENTER [REVS]SYS 673[RVSO],TO ";
5 PRINT "MOVE THE FIRST PROGRAM DOWN AGAIN,   ONTO THE END  OF THE PRESENT ONE.
6 PRINT "[DOWN]LINES ARE AUTOMATICALLY LINKED.
10 DATA 165,53,133,2,165,52,133,1,160,0,165,1,208,2,198,2,198,1,177,42
20 DATA 145,1,165,42,208,2,198,43,198,42,208,234,165,43,201,4,208,228,96
30 FOR L = 634 TO 672: READ M: POKE L,M: NEXT: REM SYS 634 MOVES PROGRAM UP
50 DATA 160,0,56,165,1,229,42,165,2,229,43,144,54,165,42
60 DATA 208,2,198,43,198,42,177,42,208,244,56,165,42,233,1,176,2
70 DATA 198,43,133,42,177,1,145,42,230,42,208,2,230,43,230,1,208,2,230,2
80 DATA 165,53,197,2,208,234,165,52,197,1,208,228,76,66,196,76,85,195
100 FOR L = 673 TO 742: READ M: POKE L,M: NEXT: REM SYS 673 APPENDS PROGRAM
READY.
```
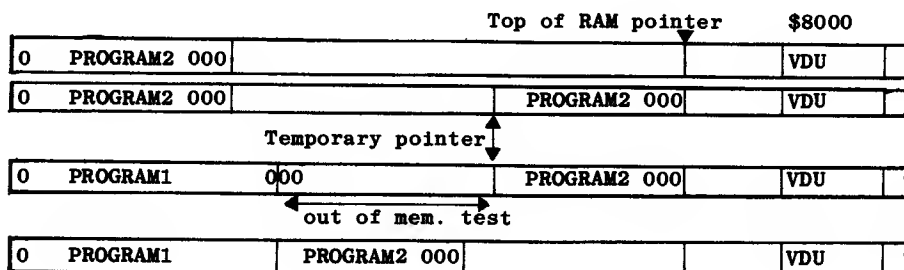
ASC

BASIC arithmetic function of string argument

PURPOSE: Computes the Commodore ASCII value of the initial character of a string expression. ASC is essential when testing individual characters, for example screen formatting characters from the keyboard, and generally whenever the numerical equivalent of an ASCII character is more easily handled than the character itself.

Syntax: ASC(string expression). The string expression can be any valid expression of literals, string functions and the '+' concatenator, with the single exception of the null character "". Any string whose length is 0 elicits an ?ILLEGAL QUANTITY ERROR message; in practice the null character as defined by "" is the only easy way to generate such a string. The CBM ASCII value as returned by ASC can take any value from 0-255; a table in the appendices shows the relationships between characters and their ASCII values. Note that ASC(X$)=0 when X$=CHR$(0); this is not the same as "" in Commodore's BASIC.

Modes: Direct and program modes are both valid.

Examples:
```
160 GET J1$: IF J1$="" GOTO 160
163 J1=ASC(J1$): IF J1=13 THEN :   REM PROCESS CARRIAGE RETURN
166 IF J1=20 THEN:                 REM PROCESS DELETE KEY
```

This incomplete program extract shows how keyboard entries can be processed; line 160 GETs a key, avoiding the ?illegal quantity trap by testing for the null character. When a key has been entered, it is converted to its ASCII value for processing. Complete validation of keyboard entries in BASIC can be carried out in this manner, with the exception of the STOP key only.

```
1340 FOR L=1 TO 6: POKE 799+L, ASC(MID$(TEST$,L)): NEXT
```

This example shows the method to move a string into RAM: the string TE$ of length 6 is POKEd into locations 800 to 805, for use in a machine-code comparison routine, from BASIC, in six separate pokes.

```
22000 IF PEEK(QQ)=ASC("*") THEN ERR$=" * SET"
PRINT ASC(MID$(S$,L)) - 192 : REM CONVERTS UPPER CASE A-Z TO 1-26
```

Finally, the third example shows how readability can be improved by using the ASCII function itself, rather than its value - 42 in the case of "*". The fourth example prints the Lth letter of string S$ as a number from 1 to 26, so if S$="HELLO" and L=2, the value 5 appears. This type of routine is useful when computing check digits, enciphering data, and so on.

Notes: [1] The converse function to ASC is CHR$. PRINT ASC(CHR$(N)) prints N. STR$ is not the converse: STR$(42) is not an asterisk, but " 42".

Abbreviated entry: aS

Token: $C6 (198)

Operation: After the function's string expression has been evaluated, it is set up in RAM with its 3 parameters (length and 2 byte pointer) on the stack. ASC recovers these parameters. It tests the length, and if this is zero exits with ?illegal quantity. This is surely a bug; there is no problem in making the value 0. However, now the accumulator is loaded from memory, using the string's pointers, so whatever the length of the string, its initial is fetched. This value is the ASCII value: there is no conversion carried out on the byte. A standard ROM routine turns it into the floating point equivalent in accumulator #1.
All ROMs process this function in this way.

ROM entry points:
```
BASIC 1: $D663 (54883)
BASIC 2: $D665 (54885)
BASIC 4: $C8C1 (51393)
```

ATN

BASIC arithmetic function

PURPOSE: Calculates in radians the principal value of the arctangent of the argument; this can be any arithmetic expression irrespective of sign. The diagram illustrates the relationship between two sides of a right-angled triangle, and the angle calculated by ATN.

NOTE: This function has no connection with ATN on the IEEE bus, which is the 'attention' line.

Syntax: ATN(arithmetic expression). The expression must be syntactically correct and within the range acceptable to the floating point logic (±1.7 E38 approx).

Modes: Direct and program modes are both valid.

Examples: `1100 ALPHA = -ATN (YV/ZV9: BETA = -ATN(XV/ZV):`
                 `2130 LET R=ATN( (E2-E1)/(N2-N1) ):REM COMPUTE BEARING AND DISTANCE`

Both examples, as might be expected, are related to trigonometry; one is from a perspective plotting program, the other from a two dimensional program for surveyors in which coordinates easting and northing are input. In each case the assigned variable, ALPHA, BETA, and R, takes the value of an angle in radians, which therefore is in the range -pi/2 to +pi/2.

Notes: [1] The diagram shows the connection between X and ATN(X),for those who are unused to geometry; a right angled triangle is a convenient standard to demonstrate geometrical ratios, but has no particular significance beyond its ease of use.
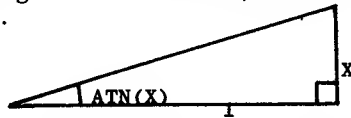
[2] See the appendix on trig. functions for general solutions.
[3] To convert radians to degrees, multiply by 180/pi. This changes the range of values of ATN from -pi/2 - pi/2 to -90° - 90°.
[4] In some cases, ATN(X) is a useful transformation to apply, since it condenses almost the entire number range into a finite set from about -1.57 to +1.57.

Abbreviated entry: aT

Token: $C1 (193)

Operation: The actual evaluation uses a 12-constant series summation. The argument (after validation) is converted into the range 0-1: if negative, the sign is stored for later recovery, but the calculation is carried out on the absolute value. And if the argument is greater than 1, the reciprocal is used in the series, and the result subtracted from pi/2 (90°).

All ROMs process this instruction in the same way. That is to say, the logic is identical, even though the entry points, absolute addresses, and (with BASIC 1) zero page locations vary.

ROM entry points:

     BASIC 1: $E048 (57416)
     BASIC 2: $E08C (57484)
     BASIC 4: $D32C (54060)

AUTO

BASIC system command not available directly in CBM BASIC

PURPOSE: Utility to generate linenumbers when entering BASIC program lines.

Versions: Typically these generate linenumbers starting at 100 and incrementing in steps of 10. The usual implementation is a BASIC routine to print numbers and to input an entire line when return is pressed, using the keyboard buffer to accept two carriage return characters. One of these causes the line to be incorporated into the program; the next runs the program again. This is also a favourite machine-code command on EPROMs from 'Toolkit' through to 'Power'.

The following routine has these features:
[1] Optional flashing cursor; omit the POKE in 60010 if this is not needed.
[2] Check for premature return, so that a linenumber is not wasted,
[3] Lines up to length 80 are accepted
[4] Press STOP to stop.

```
60000 INPUT "AUTO: ENTER START,INCREMENT"; S,I
60010 PRINT "[CLR][DOWN][DOWN][DOWN]"; S;: POKE 167,0
60020 GET A$: IF A$="" GOTO 60020
60030 PRINT A$;: IF ASC(A$)<>13 THEN 60020
60040 P = PEEK(32889 + LEN(STR$(S))): IF P=32 OR P=160 GOTO 60010
60050 PRINT "S=" S+I ":I=" I ":GOTO 60010[HOME]"
60060 POKE 158,2: POKE 623,13: POKE 624,13
60070 END
```

Note that line 60040 checks the location just after the linenumber; if it finds either a space or a shift-space, clearly nothing has been entered in the line so far. The routine therefore prints the same linenumber again. The value 32889 is 32768 + 121, which is appropriate to 40-column screens. With the 8032 this must be replaced by 32768+241 = 33009.

BASIC 1 PETs have the keyboard buffer (and much more) differently arranged. Line 60010 requires POKE 548,0 and line 60060 becomes
60060 POKE 525,2: POKE 527,13: POKE 528,13

CHR$

BASIC string function of numeric argument

PURPOSE: Converts any numeric expression in the range 0-255 into a string with length 1 consisting of the CBM ASCII equivalent character. This is the only convenient method to print and manipulate special characters like carriage return and ", which are CHR$(13) and CHR$(34) respectively.

Syntax: CHR$(numeric expression). The expression in parentheses must evaluate to 0-255. If the number is non-integral, it will be rounded down, and this rounded value must be in the correct range. So CHR$(-.01), CHR$(500) and CHR$(X$) cause error messages.

Modes: Direct and program modes are both valid.

Examples: A$ = CHR$(34) + CHR$(18) + "NAME" + CHR$(146) + CHR$(34)
                 NS$ = CHR$(160) + NS$
                 PRINT CHR$(7)
                 3300 PRINT#4,CHR$(27)"E08"CHR$(27)"L06"

The four examples above illustrate the use of this function to construct individual characters which are otherwise difficult to deal with. The first puts a string within quotation marks, and adds the [RVS] and [RVSOFF] characters. The second adds a leading shifted-space to a string; this is more readable than the alternative NS$ = " " + NS$. CHR$(7) is the 'bell', and this command will make appropriately equipped CBM's tinkle and printers beep. The final example shows a command typical of non-IEEE, non-Commodore printers; CHR$(27) is 'Escape' and the string sets horizontal and vertical spacing on a Qume daisywheel printer.

             PRINT CHR$(34);: FOR J = 1025 TO 1100: PRINT CHR$(PEEK(J));: NEXT
             C$="": FOR J = 1 TO 6: C$ = C$ + CHR$(PEEK(KT + J)): NEXT

Conversions of the contents of RAM into strings can be performed in BASIC by combining CHR$ with PEEK. The first example, in direct mode, prints a line or two of BASIC as it is stored in RAM.(This is not the best method). The second recovers a string which has been poked into RAM; C$ is built up one character at a time until a 6-character long string is formed.

Notes: [1] CHR$ is the converse function to ASC. A particular application of these functions is conversion from one character set to another, for instance screen dumping to a printer, where the PEEKed value needs a fairly elaborate routine to ensure that it PRINTs the way it looks on the VDU. See DUMP.
[2] CHR$(0) represents a null character, but has length 1. This may result in some anomalies; X$=X$+CHR$(0) adds a trailing null character to X$, the length of which is also incremented by 1, but the nulls do not print; so X$'s length appears to be longer than X$. Embedded null characters can be inserted into strings: Y$="123" + CHR$(0) + "45" prints 12345 but returns VAL of 123 and LEN of 6. If sorted, Y$ precedes 123*5, 12344, and so on. Note that ""<CHR$(0) is 'true', rather oddly.

Abbreviated entry: cH (includes the $)

Token: $C7 (199)

Operation: First, the contents of the parentheses are found and checked for range 0-255. Provided this is correct, a string of length 1 is set up at the current string pointer position, and the single byte value stored in this location. If the string is assigned - X$=CHR$(123) say - this string is permanent; if the string is used as an intermediate only, as in PRINT CHR$(123), the pointers are not reset and the next string will overlay the character.
All ROMs process this function in this way.

ROM entry points:

         BASIC 1: $D5C4 (54724)
         BASIC 2: $D5C6 (54726)
         BASIC 4: $C822 (51234)

CLOSE

BASIC input/output command

PURPOSE: Completes the processing of a file and deletes the file and its details from the **three file tables**. Files opened to the keyboard or the screen are deleted from the tables with no other action. Cassette files opened for reading are dealt with in the same way. But cassette files which write data also write a zero byte to denote end-of-file; and if the secondary address was 2, a tape 'header' is also written holding the end-of-tape value of #5. IEEE files with secondary address zero - usually, non-CBM hardware - again are simply removed from the tables; other IEEE files are sent commands to close files, and this function is carried out by the receiving hardware. In the case of CBM disks, an end-of-file: indicator is put into the last sector of the file, so that the chaining sequence of tracks and sectors for that file is complete and up-to-date and terminates correctly.

Syntax: The syntax is identical to that of CLOSE; however, any parameters following the logical file number are subsequently over:written by CLOSE, so for practical purposes CLOSE arith. expr. is the correct syntax, where the expression must evaluate, after rounding down, to 1-255. If the file does not exist, no error message results.

Modes: Direct and program modes are both valid.

Examples: `OPEN 4,4: PRINT#4,"HELLO!": CLOSE 4: REM MESSAGE TO PRINTER`
                `OPEN 1,1,1,"FILE": PRINT#1,"HELLO!": CLOSE 1: REM MESSAGE TO TAPE`
                `100 CLOSE 1,2,3,"4": REM SAME EFFECT AS CLOSE 1`
                `1000 PRINT#8,CHR$(13);: CLOSE 8: REM BASIC<4 DISK FILE CLOSE`
                `1100 PRINT#4: CLOSE 4: REM CLOSE PRINTER, WHEN CMD HAS BEEN USED`

CLOSE is a straightforward command, made more complicated than need be the case by the behaviour of CMD and PRINT#. The former leaves output devices still listening, and needs a final PRINT# to unlisten the bus; the latter, on CBM disk drives using BASIC<4, prints extra linefeed characters (ASCII character 10) after the carriage returns which mark the end of adjacent records. BASIC 4 also has the DCLOSE command.

Notes: [1] <u>RAM Tables</u>. CLOSE deletes three entries from these tables (see OPEN for illustrations) unless the entry happens to be the last of the files, by overwriting its three parameters by those of the last entry, then reducing the number of files open by 1. This of course is designed so that the ten files maximum may be efficiently used. Sometimes, notably after editing a program, the number-of-open-files parameter is set to 0, leaving the tables in RAM. If a file has not been closed, due to Stop or perhaps a syntax error, it may still be possible to close it by poking in the number of open files (or 10) and closing the file in direct mode. The location is 174 (610 in BASIC 1). Alternatively, `OPEN15,8,15: CLOSE 15` is suggested in a manual.

[2] <u>Disk Files</u>. Files opened for read need not be closed except to make space for more files. *CBM disk files opened for write must always be CLOSEd correctly. Otherwise, the track/sector pointer in the final sector will point to a usable area on the disk; sooner or later two files will become interlocked and the data on one corrupted.* See COLLECT for more on this subject.

Abbreviated entry: clO

Token: $A0 (160)

Operation: Parameters are fetched by the identical routine used by OPEN. The logical file is looked for, and, if found, its parameters are taken from the tables are overwrite any other values. The device number determines which branch is now taken: cassettes, screen, and keyboard are processed as described above; IEEE devices also call a 'Clear Channel' ROM routine.

ROM entry points: CLOSE is a 'kernel' command. Its address is $FFC3. It calls:

BASIC 1:	$F2C8 (62152)	LDA file no. then: $F2CD (62157)	CLOSEs.
BASIC 2:	$F2A9 (62121)	" $F2AE (62126)	"
BASIC 4:	$F2DD (62173)	" $F2E2 (62178)	"

CLR

BASIC command

PURPOSE: Appears to erase all BASIC variables currently in memory, leaving the BASIC program, if there is one, unchanged. Any machine code routines in RAM are left unaltered.

Syntax: CLR. CLR has no parameters. It may be followed by spaces, but must be followed by a colon or and end-of-line zero byte. (Some versions of BASIC use a parameter with CLEAR to allocate specially reserved RAM: this cannot be done directly with Commodore's CLR).

Modes: Direct and program modes are both valid.

Examples: CLR
```
50000 CLR: ?"VARIABLES ALL ERASED": REM ALL RESULTS SO FAR ARE LOST.
10 POKE 52,0: POKE 53,48: CLR: REM TOP OF MEMORY IS NOW $3000
10 POKE 134,128: POKE 135,48: CLR: REM OLD ROM: TOP OF RAM=$3800
```

This command operates by moving pointers about; it does not erase variables in the sense of, say, putting null characters in all the locations which previously held data. The first two examples are straightforward; in direct mode, if X perhaps was 1.414 and S$ was "J. Smith", then after CLR both variables will return 0 or null, as appropriate to the variable type. And in program mode the same effect obtains. Program running is not changed; so the program carries on as before, except that its variables, which presumably aren't wanted, are cleared. Also references to subroutines and loops are lost. For a complete description of this command, read the detailed explanations which follow. However, it is not necessary to fully understand its operation. The final two examples, which are alternative program lines, one for BASIC 1, show how CLR can be exploited for useful purposes, given an understanding of its *modus operandi*. A pair of zero-page pointers hold the location of top of RAM; this is not set by hardware, but by the machine itself on switchon. If new, low values are poked in the machine acts as though its RAM storage had been reduced; strings which normally fill RAM to its limit now limit themselves to the new value. In this way, free RAM is made available to the programmer for machine code routines and general storage. CLR ensures consistency between all the pointers.

Notes: [1] Simple variables (integers, strings, floating-point variables and function definitions) and arrays (integer, string, and floating-point) are deleted. In addition the DATA pointer is RESTOREd and the stack pointer reset, losing all FOR .. NEXT and GOSUB .. RETURN references. $FFE7 in ROM is called to abort input/output activity: files are aborted and the screen and keyboard are restored to primacy.

[2] There is no easy way to erase strings only, for example, or just integers. It is possible to erase arrays; their pointers are held differently, as is necessary to avoid ambiguity. After CLR, variable and array pointers are not distinguishable, so recovering the lost values is difficult.

[3] As with NEW, CLR generates anomalous error messages if a machine-code program has been loaded or the BASIC pointers are abnormally set for some other reason. Poking values for the start and end of BASIC, then CLRing, is one possible cure.

Abbreviated entry: cL   Token: $9C (156)

Operation: The 'limit of RAM' pointer, as we've seen, is stored in the 'bottom of strings' pointer; this means that new strings will be stored in the top of memory, overwriting the old ones. The 'end of BASIC' pointer is stored in the 'end of variables' and 'end of arrays' pointers. This loses both variables and string pointers. When the stack is reset, the top two values are retained, so RTS continues the program running at the same place. In addition to the changes listed in note [1] a few flags are reset.

ROM entry points: BASIC1:$C770 (51056) BASIC2:$C577 (50551) BASIC4:$B5EE (46574)

CMD

BASIC output command

PURPOSE: CMD combines two entirely distinct functions. (i) It prepares an output device, typically a printer, to receive subsequent PRINTed data until the device is unlistened. (ii) It then prints whatever string follows CMD to the printer or other device. In essence it allows a program with many PRINT statements, which would normally appear on the screen, to be diverted to some other output device.

Syntax: CMD arithmetic expression:
CMD arithmetic expression, printable expression including , and/or ;
The arithmetic expression must evaluate to 1-255. A logical file number of zero is disallowed. The comma separator, for example in CMD5,"HELLO", appears with INPUT# too, but not with PRINT. This is because PRINT 25 is syntactically correct, but CMD 5 25 is ambiguous.

Modes: Direct and program modes are both valid.

Examples: Assume OPEN 5,4 has opened a file to a printer. (OPEN 4,4 may well be used in practice: I've put 5 purely to make clear which parameter is which.

```
CMD 5            switches further output to printer. Then prints crlf.
CMD 5,;                         "              Without crlf.
CMD 5,"HELLO"                   "              & prints "HELLO"
PRNT=5: CMD PRNT  is syntactically valid.
```

Notes: [1] If we compare PRINT#5,"HELLO" with CMD5,"HELLO" it is clear that these instructions are rather similar; however, the puzzling feature of the commands is that PRINT#5,; which unlistens the device does exactly the opposite of CMD5,; which causes it to listen. This confusing aspect of CMD is the result of its combining two disparate instructions.

[2] Problems: CMD often gives rise to minor bugs.

```
[i]  OPEN 4,4: CMD 4: INPUT "NAME";N$ :REM "NAME" IS PRINTED
[ii] GET turns off CMD; only one line appears on the printer:
     10 OPEN 4,4: CMD 4,;
     20 PRINT "LINE"              :REM PRINT LINE REPEATEDLY...
     30 GET X$: IF X$="" GOTO 20 :REM IF NO KEY IS PRESSED?
     40 PRINT#4,;: CLOSE 4: END
[iii] Commodore printers (not others) somehow tend to make CMD fail
      to operate. GOSUB for example has this effect.
```

[3] To summarise, CMD seems to be, in the US phrase, a kludge to enable a program full of print statements to be easily diverted from the screen to some other device. It is easier than replacing all PRINTs with PRINT#. When developing a new program, PRINT# is likely to be a better choice: it lends itself better to CLOSE and will not lose its effect erratically. Also Commodore (cf. their printers) seem to support PRINT# in preference.

Abbreviated entry: cM

Token: $9D (157)

Operation: The parameter following the CMD token is checked. It must evaluate to 1-255. The device number corresponding to this file number is looked up in a table of up to 10 values, and the output device set. ?FILE NOT OPEN or ?DEVICE NOT PRESENT errors may greet the user while this is being attempted. The syntax is checked after CMD's parameter. Either an end-of-statement (colon or new line) or comma followed by printable expression is accepted. Finallt, the PRINT routine in ROM is entered.

ROM entry points:

```
BASIC 1: $C985 (51989)
BASIC 2: $C991 (51601)
BASIC 4: $BA8E (47758)
```

CONT

BASIC command

PURPOSE: Resumes BASIC program running after encountering STOP or END in the program, or after the STOP key had been pressed, or after a null input crash on INPUT. In this way not only can breakpoints be put into BASIC, but a program can be stopped and restarted at any point. (Well ... nearly any point. The STOP key will abort files, so that its message and READY will appear on the VDU; in some cases therefore CONT does not completely resume operation).

Syntax: CONT. No other parameters; may be followed by spaces, but must be followed by a statement terminator - a colon or end of line.

Modes: Direct mode only. (In program mode CONT goes into an infinite loop which continually jumps to itself).

Notes: [1] As a BASIC program runs, a record is kept of current and previous linenumbers, and a pointer is kept which indicates where the next statement is. All this is part of the overhead which helps to make translators slower than compilers. It also makes useful commands like CONT possible. The HELP command, implemented on some toolkits to point to the error in a line which has caused a syntax error, uses the linenumber and pointer; the routine cannot be in BASIC, which would change the pointer, but must LIST a single line in machine-code and then calculate where in the LISTed line the error was located.

[2] While the program is stopped, any of its variables may be examined by PRINTing; their values can also be changed in direct mode. With CBM BASIC new lines can't be added if CONT is to work. A ?CAN'T CONTINUE ERROR is also caused after CLR or NEW or if exit from the program was by way of a syntax error. In such cases, GOTO a convenient linenumber may serve the same purpose.

[3] The principal locations are: ($3A) holds 'previous linenumber', ($38) holds the pointer into BASIC. The high byte of ($38) is made zero if exit was by syntax error; by POKE-ing these locations, CONT can be made to work, and jump to anywhere in BASIC, although there's little practical value in doing this.

Abbreviated entry: cO

Token: $9A (154)

Operation: First the syntax is checked. Then the pointer into BASIC used by CONT (not the same as CHRGET) is tested for high byte zero, which is a standard test for a syntax error exit. Obviously a valid pointer into BASIC must be $0400 or greater, so the zero byte never leads to ambiguity. If a zero byte is found, therefore, the routine branches to print the can't continue message. Otherwise, and let us hope usually, the routine puts the stored previous linenumber into the 'present linenumber' slot, sets GETCHR to the pointer to the next statement, and runs.

ROM entry points:

BASIC 1:	$C745 (51013)	Unvalidated:	$C747
BASIC 2:	$C76B (51051)	"	$C76D
BASIC 4:	$B7EE (47086)	"	$B7F0

COS

BASIC arithmetic function

PURPOSE: Evaluates the cosine of the argument, which is assumed to be in
radians. The cosine is a ratio which is constant for an angle; the diagram
illustrates this.

Syntax: COS(arithmetic expression). The expression must be syntactically correct
and within the range acceptable to the floating-point logic (± 1.7 E38 approx).

Modes: Direct and program modes are both valid.

Examples:
```
PRINT COS(1)              prints cosine of 1 radian = .54 approx.
PRINT COS(45 * [PI]/180)  prints cosine of 45° = .707 approx.
1000 Y=EXP(-K*T) * (A*SIN(W*T) + B*COS(W*T))
2000 X=ALPHA+SIN(ALPHA): Y=1-N*COS(ALPHA)
```

The first examples show COS used in direct mode (sometimes called 'calcul-
ator mode'!) performing direct calculations. The conversion between degrees
and radians has to be performed by the user. The second examples are
typical formulas using trigonometrical functions; the first is the equation
of a damped sine curve. The second calculates two coordinates, X and Y,
on a cycloid.

Notes: [1] The diagrams show the cosine's ratio in terms of a right angled triangle,
and the concept of a radian. 'A' and 'H' conventionally represent sides ad-
jacent to X and hypotenuse (diagonal), respectively.
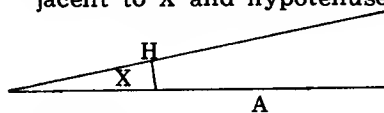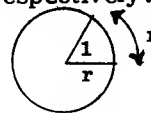
COS(X) = A/H   Angle = 1 radian

[2] Accuracy is not greatly affected by the size of the angle: this function
operates by dividing the argument by 2*pi and taking the remainder, so
there is no series approximation error related to the size of the argument,
only the error caused by the limited precision to which the argument can
be held.

[3] See the appendices for the inverse function ARCCOS.

Abbreviated entry: None

Token: $BE (190)

Operation: The argument is evaluated, and the result put into floating-point accum-
ulator #1. Pi/2 is added and the routine then drops into SIN, so COS(X) is
avaluated as SIN(X + pi/2).

ROM entry points:

```
BASIC 1: $DF9E (57246)
BASIC 2: $DFD8 (57304)
BASIC 4: $D282 (53890)
```

CRUNCH

BASIC system command unavailable directly in CBM BASIC

PURPOSE: Improves the speed of BASIC execution by deleting as much of the program as is considered redundant.

Versions: Quite a number have been issued; some, in BASIC, are only suitable for preparation of a 'fast' version of the program; some machine code versions may be used at run-time. The routine is also called 'compactor'. Uncrunch programs, which present each instruction spaced out on its own individual line, are possible too.

The rationale is that REM statements, spaces, short lines and so on, while helpful to an investigator into a program, slow the translator by wasting time jumping past spaces, switching to new lines and so on, and indirectly by slowing up GOTOs and GOSUBs, lengthening the program and thus causing more garbage collection, and so on. Unfortunately, it must be said that such mechanical ways of speeding up program execution do not have a great effect, even with specially constructed programs; their appeal is really of the 'every little bit helps' type.

Various points of attack are:-

[1] Elimination of all REM statements and lines. If they are referenced by GOTO or GOSUB or THEN the REM statement only may be retained, or, better, deleted but with its reference changed to the next line.

[2] Elimination of all spaces which are not within quotes. (Some BASICs, e.g. Apple's, do this anyway). A program modified in this way sometimes gives problems; X=T AND U will think it contains the function TAN.

[3] Elimination of lines by conflating as many together as possible. Lines spanning more than 255 bytes are unreliable, however, since pointers for DATA for example are single-byte only. Also the program won't LIST. So the maximum linelength is usually limited to 250 BASIC characters. Also, of course, a line may be referenced, say by GOTO, and therefore not be conflatable with the previous line(s).

[4] Renumbering the program with lines starting at 0 and increment of 1 makes line references as short as possible: processing 'GOTO 53' is faster than 'GOTO 12000'

[5] Systematic changes of variable names to 1 character names only, where possible, speeds up variable processing.

[6] Spare semi-colons can be removed from PRINT statements.

[7] Since the program has no spaces, the CHRGET routine may be modified to exclude the check for spaces.

[8] A trace or shadow routine *might* be able to count the frequencies with which variables are used during an actual program run; an initialisation routine could be added to the program to assign the variables in their optimum order.

For further discussion on these points, see Chapter 2.

[9] Where a 'wedge' is in use, which intercepts the GETCHR routine, considerable timesaving is often possible by deleting it with a short 6502 routine, if it is not required at run-time.

See Chapter 14 for details.

[10] Finally, the interrupt sequence can be shortened. Since the keyboard buffer will not work if this is done, its use is limited to programs which perform prolonged processing without intervention by an operator.

See Chapter 13 for details on this point.

The BASIC routines on this page illustrate the sort of methods by which BASIC programs may be compressed. They are far slower than the machine-code equivalents but nevertheless have some interest. The first, longer subroutine, to be appended on or near the end of a BASIC program, deletes all spaces not within quotes from the program, and deletes all REM statements from lines unless the entire line is a REM statement. In this case, only REM is left in place of the original REM line; it is not completely erased, since it may be the destination of a GOTO or GOSUB. Note that abbreviated forms of keywords appear on the screen; this prevents over-long lines from overrunning the standard 80 character linelength. (Because of the way the listing has been printed, the abbreviation of 'END' has appeared 'En'. This means unshifted E followed by shifted N. The same sort of thing is true for the other abbreviations, which are, of course, identical to those printed in the BASIC keywords reference section).

```
63000 POKE59458,62:A=1025:B=256:GOSUB 63100:GOTO 63003:REM *** AS STARTER
63002 B=256:A=B*PEEK(826)+PEEK(827):A=PEEK(A)+B*PEEK(A+1):GOSUB 63100
63003 L=PEEK(A+2)+B*PEEK(A+3): IF L>62999 THEN PRINT"FINISHED":END
63004 PRINT"[CLEAR][DOWN][DOWN][DOWN]"L"[LEFT]";:Q=0:REM PRINT LINENUMBER,SET QUOTES
63006 FOR K=A+4 TO A+93: P=PEEK(K): REM NOW LOOP THROUGH LINE
63008 IF P=0 THEN 63050:REM END OF LINE
63010 IF P=143 AND K<>A+4 THEN PRINT"[LEFT] ";:GOTO63050:REM DEL 'REM' UNLESS AT S
63012 IF P=143 THEN PRINT"REM";:GOTO63050:REM LEAVE 'REM' IF AT START
63014 IF P=34 AND Q THEN Q=0:PRINTCHR$(34);:NEXT:REM END OF QUOTES
63016 IF P=34 AND NOT Q THEN Q=-1:PRINTCHR$(34);:NEXT:REM START OF QUOTES
63018 IFNOTQANDP>127ANDP<203 THEN PRINTT$(P-127);:NEXT:REM PRINT EXPANDED TOKEN
63020 IF P=32 AND NOT Q THEN NEXT:REM IGNORE SPACE
63022 PRINTCHR$(P);:NEXT:REM PRINT VARIABLES,INTEGER,$,% ETC
63050 PRINT:PRINT"GOTO63002":REM PREPARE FOR NEXT LINE
63052 POKE 826,A/B: POKE 827,A-INT(A/B)*B:POKE 158,2:POKE623,13:POKE624,13
63054 PRINT"[UP][UP][UP][UP][UP][UP][UP]":END
63100 DATA***,"En","Fo","Ne","Da","In","INPUT","Di","Re",LET,"Go","Ru","IF","REs"
63101 DATA"GOs","REt",REM,"St",ON,"Wa","Lo","Sa","Ve","De","Po","Pr",?,"Co","Li"
63102 DATA"Cl","Cm","Sy","Op","CLo","Ge",NEW,"Ta",TO,FN,"Sp","Th","No","STe",+
63103 DATA-,*,/,ˊ,"An",OR,>,=,<,"Sg",INT,"Ab","Us","Fr","Po","Sq","Rn",LOG,"Ex"
63104 DATACOS,"Si",TAN,"At","Pe",LEN,"STr","Va","As","Ch","LEf","Ri","Mi"
63108 FOR K=1 TO 1E5: READ X$:IF X$<>"***" THEN NEXT: REM READ DATA UP TO *
63110 DIM T$(75):REM ARRAY FOR TOKENS
63112 FOR K=1 TO 75:READ T$(K):NEXT:RETURN:REM FILLS ARRAY WITH EXPANDED TOKENS
```

This second subroutine belongs at the start of BASIC and has the function of combining several lines into one. The composite line consists of the original lines separated by colons. The maximum linelength resulting must not exceed 251 characters, since the ROM rechaining routine (amongst others) cannot then operate properly.

```
0 INPUT "COMBINE LINES";L,U: C=1025: B=256: E=PEEK(42)+B*PEEK(43)-4
1 LT=PEEK(C+2)+B*PEEK(C+3): PRINT LT;: REM PRINTS LINENUMBERS
2 IF LT<L THEN C=PEEK(C)+B*PEEK(C+1): GOTO 1: REM FIND LOWER LINE
3 IF LT>L THEN PRINT "LINE NOT FOUND": END
4 C=C+4: REM START EXAMINING BYTES IN THE PROGRAM LINE
5 Q=PEEK(C): IF Q<>0 THEN C=C+1: GOTO 5: REM FIND END OF LINE ZERO
6 LT=PEEK(C+3)+B*PEEK(C+4): PRINT LT;: REM PRINT LINENUMBER
7 IF LT>U THEN SYS 46262: END: REM RECHAIN. (NOTE** BASIC 4 VERSION)
8 POKE C,ASC(":"): FOR J=C+1 TO E: POKE J,PEEK(J+4): NEXT: E=E-4:GOTO5
```

** BASIC 2: Line 7 contains SYS 50242, but is otherwise identical.
** BASIC 1: Line 0 has E=PEEK(124)+B*PEEK(125)-4. Line 7 uses SYS 50227.
And line 8 must be spread over 2 lines, 8 & 9, because POKE of PEEK fails.

Chapter 2 explains the working of these routines and others like them.

DATA

BASIC data marker

PURPOSE: Enables data of any type, alphabetic, numeric, or ASCII to be stored within a program, without being read from disk or tape or being keyed in. The data is retrieved by the READ statement which assigns each item of data to a variable in the same order that the data is stored. Originally, BASIC accepted data from punched cards, not from keyboards, so READ statements appeared throughout programs in the way INPUT and GET do now.

Syntax: DATA is followed by ASCII characters interpreted like this:-
" delimits a literal, which is READ as a single string
, outside of quotes separates one DATA item from the next
: outside quotes, or a new line, ends the DATA statement.
Other characters are treated as data. Note that the position within a program of DATA statements is irrelevant, but the order is important.

Mode: Program mode only is valid. (The data pointer starts at BASIC, and cannot reference data in the input buffer).

Examples: 100 DATA "Al,Aluminum,24.6","Cu,Copper,136.2","Fe,Iron,35.1"
12000 DATA MACHINECODE,120,169,46,133,96: PRINT "STARTING.."
50000 DATA 27,14,27,9,22,9,22,9: REM HORIZONTAL
50010 DATA 3,4,5,8,8,9,9,10   : REM VERTICAL
50020 DATA 1,20,2,6,6,6,6,6   : REM LENGTHS OF INPUTS

The first example shows three strings held as data; READ X$ takes in the entire string within quotes, so READ X$: PRINT X$ repeated three times prints each string. The second example shows data with a special marker; a block of DATA beginning in this way can be made relocatable, using a loop to read all the data until, in this example, X$ say = "MACHINECODE". Finally, three lines show how data can be structured. Three sets of eight parameters hold details relevant to a screen input format.

Notes: [1] DATA is used for repetitive work: sometimes there is no need for DATA e.g. PU$="EachPackUnitTubeReelSet Pair" holds information as a string. The command is processed by the same routines that INPUT and GET use, which explains the punctuation by " and , and :. Also the variables must be of the same type as the data. Read X$ is always safe, but READ Y may not be. See READ for full explanations of these points. Note also that RESTORE sets the pointer to DATA back to start, so data is always rereadable.

[2] DATA statements can be forced into a program using the keyboard buffer to simulate keyboard entry of a line.

[3] Bugs: (i) DATA uses INPUT's routines, so some peculiarities of INPUT affect READ. Unshifted leading spaces and some graphics are lost.
(ii) Syntax error reported in a valid DATA line in fact means that there is an error in the READ statement. You'll have to search to find which one.
(iii) Unnoticed commas can introduce baffing bugs. The statement
DATA 31,28,31,30, has 5 data items, including a null string.
(iv) Take care when introducing more DATA into a program which has some already. READ will impartially treat information in the wrong sequence as though it were correct. This can create problems, especially with 6502 code.
(v) A variable cannot be input: DATA 1,2,3,X treats X as a string.

Abbreviated entry: dA   Token: $83 (131)

Operation: When a data statement is found, it is ignored, just like a remark statement except that the next statement, not the next line, is jumped to. The routine hunts for a : or zero byte, the Y register holding the offset; this is added to CHRGET's address so the effect is to skip the data.

ROM entry points:

BASIC 1: $C7F0 (51184)   BASIC 2: $C800 (51200)   BASIC 4: $B883 (47235)

DBL

Command unavailable directly in CBM BASIC

PURPOSE: increases the accuracy of calculations by increasing the storage space of floating-point numbers.

Versions: Some BASICs (IBM, Tandy) have commands of this type, in which space allocated for the storage of floating-point numbers is, for example, doubled. Longer numerals are slower to process, but more accurate. Commodore (and Apple, which has nearly identical number processing routines) are designed around their standard five byte storage system, and it is impossible to extend the processing capability of the current routines. (There are rumours that BASIC 5 will include BCD arithmetic, enabling great accuracy to be obtained). It is certainly possible to reach the point at which numerals are no longer processed accurately. Thus 999 999 999.1 is printed as 999999999, and any values much larger are converted so they appear in scientific notation. There is of course an element of spurious 'accuracy' in many figures of this magnitude. Not many measurements are correct to one part in a thousand million. There are few routines available, as a result of this, to process long numerals. Osborne/Donahue has 25 pages on the subject.* The best approach is to use fixed-point numbers; in this way all the difficulties associated with floating-point accumulators are abolished. A usable format might be 15 figures before and after the decimal point, plus extra space to allow the output to be grouped in sets of three digits separated by commas or spaces. Fifteen figures after the decimal may seem excessive; but some calculations, for instance overnight interest on bank deposits, need considerable precision. The BASIC translater could be programmed to intercept and process (say) A\$=A1\$*A2\$. But this would be ambiguous in the case A\$=A1\$+A2\$. So the best routine is likely to use syntax like this: !A\$+B\$ or this: SYS 700: A\$+B\$, and, to avoid having to peek the answer byte by byte, to assign the result to another string.

---

*This book has BASIC programs which add, subtract, and multiply (not divide) integers only. The relevant chapter is 'Making the most of CBM features' which appeared in the earlier edition as 'Overcoming the limitations of PET BASIC'. The multiply routine has bugs: the first item must have an even number of characters, and embedded zeros may crash the program. To remedy this, add:

```
1160 GOSUB 3000
2150 RETURN
```

DEF FN
BASIC command

PURPOSE: Assigns a numerical function, which can be called by FN. The function definition has a name (of the usual BASIC type) and a dependent variable.

Syntax: DEF FN real variable (real variable) = arithmetic expression. The variable in brackets is the dependent variable. If the arithmetic expression does not include it, it's called a 'dummy variable'. The definition has to fit into one line of BASIC. After the function has been defined, it can perform calculations on its argument: PRINT FN name (arith. expr.) for instance prints the value taken by the function. There may be run time errors if the function cannot be evaluated, typically ?DIVISION BY ZERO ERROR.

Mode: Program mode only; direct mode produces an ?ILLEGAL DIRECT ERROR.

Examples:
```
10 DEF FN DEEK(X)=PEEK(X) + 256*PEEK(X+1):REM SETS UP FN DE(X)
100 DEF FN MIN(X)= -(A>B)*B - (B>=A)*A    :REM RETURNS SMALLER
1500 DEF FN Y(X)=A*X*X + B*X + C :REM CALCULATE AX²+BX+C
527 DEF FN L(QQ)=QQ*(B=10):REM ALWAYS 0; OR -QQ WHEN B=10
```

Line 10 defines DEEK(X) as a double-byte peek. The result is much easier to read than a subroutine; PRINT FN DEEK(1) prints the current USR address which is stored in bytes 1 and 2. If X is negative, or exceeds 65535, the program will of course crash, with an error. The second example uses X as a dummy variable. In the same way that FRE(0) and FRE(99) return the same value, FN MIN(1) and FN MIN(9) take the identical value, which is A if A is smaller, B if B is smaller. Line 1500 is a mathematical function: the example is a quadratic expression; it could be a financial calculation, a scientific formula, a commercial cost expression. Note that line 1500 includes three variables, A, B, and C, which are included in the evaluation of the quadratic. The function can of course contain constants:

```
10 DEF FN Z(X)=5*(1+TAN(X)),
```
and it can include a function def-

inition:
```
15000 DEF FN P(P) = 1 + 2*(1-P) + 3*(1-P)^2 + ... +FN PP(P)
15005 DEF FN PP(P)= 6*(1-P)^7 + 7*(1-P)^8: REM 2 LINES FOR DEF
```

Notes: [1] DEF works by storing a pointer to the expression among the simple variables. FN causes the dependent variable to be assigned the value in brackets, and then the BASIC code in the program itself is used to evaluate FN. A function can be redefined freely, like any other variable: DEF FN Y(X)=X: DEF FNY(X)=22: is OK. The definition is stored like this:

NAME ASCII+128	NAME ASCII	PTR. TO LOW	EXPR. HIGH	PTR. TO LOW	VAR. HIGH	Not used

The high bits in the name, which are on and off respectively, ensure there will not be confusion with other variable types, so DEF FNX(X)=X%+X$ is valid.
[2] Bugs. i. FN called before the equivalent DEF FN gives ?UNDEF'D FUNCTION ERROR, because it is unable to find, and can't set up, the function.
ii. An error in the function definition causes ?SYNTAX ERROR in the line using FN, even if the line is valid. (READ does the same thing).
iii. If a new program is loaded from within an old one, unless it has an identical definition in the identical place in RAM, any function definitions which existed will no longer work correctly, and should be redefined.
[3] Note that the dependent variable does not change when a function definition is used. So, in the very first example above, X=100: PRINT FN DEEK(1000) leave X unchanged, although FN DEEK uses X. The value of X is in fact temporarily stored in the area reserved for the function definition itself.

Abbreviated entry: dE (fn has no short form) Tokens: DEF $96 (150), FN $A5(165)

Operation: DEF checks FN token, mode, variable types, brackets, and '=', but not the expression, then sets the name and pointers. FN has no action address; it is searched for during expression evaluation and has its own ROM routine.

ROM entry points:
```
DEF:   BASIC 1:$D295 (53909)   BASIC 2:$D28D (53901)   BASIC 4:$C4DC (50396)
FN:    BASIC 1:$D2D6 (53974)   BASIC 2:$D2CE (53966)   BASIC 4:$C51D (50461)
```

DEL

BASIC command not available in CBM BASIC.

PURPOSE: DEL deletes BASIC program lines; typical syntax is DEL a - b where a and b are linenumbers. This command removes test routines and driver routines to clean up a program when testing is over, or removes particular features of a program to leave a core of reusable standard routines.

NB: DELETE is sometimes a disk command to remove a file; 'SCRATCH' is Commodore's version.

Versions: In view of the simplicity of programming and usefulness of this command it is remarkable how few versions exist. In Microsoft BASIC DEL can only be supported in direct mode, because the program shrinks, and the storage of variables has to be revised. Validation of DEL a - b is similar to LIST, and the operation of the routine would be to search for the two lines, then memory move the upper part of the program to the end of the lower part, and rechain the result.

The version below is in BASIC, in the form of a subroutine which sits at the end of the program. RUN 61000 inputs the linenumber limits, and the routine proceeds to print on the screen all the linenumbers which the program has between (and including) the limits. It relies on the well-known keyboard buffer trick of putting in carriage returns from the program.

```
61000 A=1025: B=256: INPUT "DELETE FROM,TO";L,U
61010 IF PEEK(A+2)+B*PEEK(A+3) < L THEN A=PEEK(A)+B*PEEK(A+1): GOTO 61010
61020 POKE 828,U-INT(U/B)*B: POKE 829,U/B: GOTO 61040
61030 B=256: A=PEEK(826)+B*PEEK(827): U=PEEK(828)+B*PEEK(829)
61040 IF PEEK(A+2)+B*PEEK(A+3) > U OR PEEK(A)+B*PEEK(A+1)=0 THEN END
61050 PRINT "[CLR][DOWN][DOWN][DOWN]" PEEK(A+2)+B*PEEK(A+3):
      PRINT "GOTO 61030": PRINT "[UP][UP][UP][UP][UP][UP][UP]
61060 POKE 826,A-INT(A/B)*B: POKE 827,A/B: POKE 158,2:POKE 623,13:POKE 624,13
61070 END
```

Comments:

A=first byte of link address; so its initial value is 1025, and the pointer to the next line, and the current linenumber, are stored in locations A/A+1, and A+2/A+3.
B=256 is a convenient constant.

61000 inputs L,U = lower and upper linenumbers
61010 scans the line numbers until one is found which is not less than L
61020 stores the upper linenumber in cassette buffer #2
61030 Loop to print linenumbers: A recovers link, U recovers upper line
      61040 ends if upper linenumber exceeded, or program's end reached.
      61050 clears screen, prints linenumber, prints GOTO 61030, moves up
      61060 saves link address and puts two returns into the buffer.
61070 END causes the loop to delete one line.

DIM

BASIC command

PURPOSE: Allocates space in memory for an array of specified name, type, and
dimensions. The name has two significant characters, the type may be real,
integer or string, and multiple dimensions are accepted. Array elements are
numbered from zero. On setting up, every element of any array is made 0
if numeric or null if string.

*NOTE: Strings do not need to be individually dimensioned for length; the
system tokes care of this. So X$(20) is o string orroy holding 21 strings;
not o single string of length 20.*

Syntax: DIM name(arith. exp.1, arith.exp. 2, ..., arith.exp. n1) [, name 2(arith.exp.
, ...)] where the square brackets indicate optional repetitions. Each
arithmetic expression is evaluated and rounded down if non-integral. The
permitted range of values is 0-32767. High values will generate ?OUT OF
MEMORY ERROR. See note [3] for information about BASIC 1's peculiarities.
The syntax is not checked thoroughly. DIM T for example does not give
any error indication.

Modes: Direct and program modes are both accepted.

Examples: 12000 DIM P%(18),L%(8),A1(18),SG$(2)
            540 DIMS(B*N + 20):   REM B= 2 TO 4.
            50 DIM A(10,10,10),T(24),POSN(X,Y,Z): DIM LOCATE (2*Y),Q(X,10)

            FOR J=0TO10: X$(J)=STR$(J): NEXT: FOR J=10 TO 0 STEP -1: ?X$(J): NEXT

DIM is a straightforward command: the problems associated with it mainly
derive from the difficulties associated with processing large amounts of data.
Arrays can be 'dynamically' dimensioned with Microsoft BASIC. This means
lines like 540 are valid, where an arithmetic expression has been used to
compute the array subscript size, as well as lines like 12000, in which
absolute values are used. Line 540 assigns an array S() a dimension which
is B times as large as another array of dimension N, and adds another 20
spare elements. In this way, arrays can be assigned by soft-coding to be
a suitable size for the work in hand. Line 50 dimensions three multi-dimension
arrays. Note that DIM must be repeated at the start of each new DIM
statement. Finally, the direct mode example shows an implicit dimensioning of
the array X$(). Although DIM X$(10) is not included in the line of coding,
the first time it is met during running the translator searches for X$(0), and
when it doesn't find it, sets up the array. The default value of DIM is always
10; larger arrays therefore must be dimensioned to avoid ?BAD SUBSCRIPT
ERRORs.

Notes: [1] <u>Some general notes on arrays</u>. These notes are long and comprehensive;
don't be put off DIM and arrays because of this detail and apparent complex-
ity. The basic idea of giving a whole batch of data just one name is simple,
and the method of numbering the separate items isn't too hard either.
i. Since computers start counting at zero, it is not surprising that Microsoft
have allowed zeroth elements in their arrays. Some people *consider that
these elements should not be used, because of possible compatibility problems
between other versions of BASIC. In any program developed for subsequent
mini or mainframe use, or with portability in mind, this is likely to be true.
On the other hand, this may be unimportant; certainly there are plenty of
other potential conversion problems. The zero element, because of its
uniqueness, may hold averages, totals, comments, or any other summary item
about the array. This example line shows how a total might be built up:

            DIM A(20): FOR X=1TO20: INPUT N: A(X)=N: A(0)=A(0)+N: NEXT

ii. ?REDIM'D ARRAY ERROR will occur if DIM is inadvertently included within
a loop. Move it to an earlier part of the program.

---

*see for instance Donald Alcock's 'Illustrating BASIC'.

iii. DIM X(5,0) is syntactically correct but adds nothing extra to the array X(5) except the additional effort of incorporating ',0' to the subscripts. DIM C(7,2) sets up a three-column array, with a choice of C(M,1) or C(M,2) for M = 1 to 7. (And the zero elements may also be used). Two-dimensional arrays, like this one, are usually visualised as rows followed by columns: A(R,C).

iv. CLR seems to delete all variables and all arrays, an effect produced by the shifting of several pointers. (See diagram). Because of the way variables are partitioned into simple and subscripted types, it is easy to erase all the arrays from memory, whilst leaving all the simple variables untouched. We can achieve this in BASIC>1 with: POKE 46, PEEK(44): POKE 47, PEEK(45) and in BASIC 1 with: P=PEEK(126): POKE 128,P: P=PEEK(127): POKE 129,P. Large arrays consume a lot of RAM; this manoeuvre may therefore usefully eliminate a redundant array from memory. The 'Scatter Sort' (q.v.) provides an example.

[2] Array dimensioning by default doesn't only occur when an assignment statement refers to a non-existent array value. It happens also when such an array is present in an expression. This program: 0 X=Y=Z: END when run, sets up variable X and assigns it the value -1, because Y and Z are both zero. Although X is present after the program, neither Y or Z is. But this: 0 X=Y=Z(3): END not only sets up X, but array Z(), which is given the default dimension of 10. This may lead to unexpectedly small reserves of RAM.

[3] BASIC 1. This ROM has a serious bug, causing an array to remain empty from its 255th element on. Items out of this range are written wrongly (the 260th as the fourth) and read back wrongly. This error applies to multi-dimension arrays, and causes bugs which can be hard to detect. For example, remembering to allow for the zeroth element, X(4,50) has 5*51 = 255 elements, and Y(9,24) has 10*25 = 250 elements. Both of these will process successfully. But Z(16,16) has 17*17 = 289 elements and will not be reliable.

[4] String Arrays. Any previously undefined variable will cause all the arrays held in RAM to memory-move up, to create the necessary space. This is time-consuming and especially so with BASIC 4 string arrays. This is because BASIC 4 strings each have their own pointer, and these all need updating. To see this effect, try DIM X$(1000):A=1:B=2:C=3:D=4:PRINT A. *If* this once-only delay is important – often it won't be – set up most or all variables *before* large arrays are dimensioned. For the connection between string array dimension and memory-freeing time, see Chapter 2 and the section on FRE. Finally, note that DIM can be absurdly high with strings, because all the pointers can point to the same string: FOR J=0 TO 1000: X$(J)="ELEPHANTINE": NEXT uses 3K bytes.

POINTERS: End of BASIC /End of vars /End of arrays

BASIC<4:	PROGRAM	Vars/String pointers	String array		Strings

BASIC 4:	PROGRAM	Vars/String pointers	String array		Strings
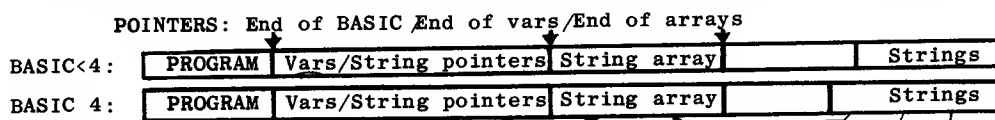
[5] Storage Space. Space taken up in RAM can be found with the aid of FRE. F=FRE(0): DIM Z%(500): PRINT F-FRE(0) shows the method. It can be found by the formula, for any n-dimensional array:
Bytes=5 + 2*n + (dim1 + 1)*(dim2 + 1)* ... *(dimn + 1)*2,3 or 5 for integer, string, or real number arrays respectively. Example: PQ(100,4,2) occupies
5 + 6 + 101*5*3*5 bytes = 7586 bytes.

Abbreviated entry: dI (this is why DIRECTORY needs diR!)

Token: $86

Operation: The first character of the array name is stored in X. Most of the work is done by the next routine, which searches for the variable, and by another routine which it calls, and which is extremely long, 'find or create array'.
After this, if the statement hasn't ended, DIM loops back to check for a comma and repeat the operation with the next array to be dimensioned.

ROM entry points: BASIC1:$CF71 (53105) BASIC2:$CF63 (53091) BASIC4:$C121 (49441)
Find/create array:BASIC1:$D0B9 (53433) BASIC2:$D0AC (53420) bASIC4:$C2FC (49916)

DUMP

Utilities unavailable directly in CBM BASIC

PURPOSES: (1) **A screen dump prints a duplicate of the screen onto paper.**
A printer may, of course, be unable to reproduce the full range of Commodore characters. Routines of this sort are valuable for record-keeping purposes. If the screen is built up with POKEs or machine-code a special routine is necessary. With output which is simply PRINTed to the screen it is usually quicker to direct the output to the printer.
(2) **A dump of variables prints out current variable names and values.**
This is of some use when debugging BASIC.

Versions: (1) <u>Screen Dumps.</u> Many versions, both BASIC and machine-code, exist. Before looking at these, let's consider the problems that can arise. Firstly, some characters may be unprintable. Secondly, a printer may not use CBM'S version of ASCII. Thirdly, the upper and lower case alternate character sets have to be allowed for. Fourthly, some screens have 80 columns, others 40. None of these is a real problem. (If however some non-standard screen display is used, for example a high-resolution graphics hardware unit, completely new routines will be needed to dump their screen output).

Early versions, in BASIC,*were concerned with non-CBM printers, which did not exist. They convert the screen memory characters into outputtable equivalents. (See appendix for screen memory and ASCII). Graphics were ignored or printed as (say) *. This program, including allowance for either graphics mode, shows the type of thing necessary:

```
40000 REM *** 40 COLUMN SCREEN DUMP ***
40010 OPEN 4,4: CMD 4: IF PEEK(59468)=14 GOTO 40200
40100 FOR J=0 TO 24: FOR K=0 TO 39: X=PEEK(32768 + J*40 + K)
40110 IF X<32        THEN PRINT CHR$(X+64) ;: GOTO 40160
40120 IF X>31  AND X<65  THEN PRINT CHR$(X)     ;: GOTO 40160
40130 IF X>128 AND X<160 THEN PRINT CHR$(X-64) ;: GOTO 40160
40140 IF X>159 AND X<193 THEN PRINT CHR$(X-128);: GOTO 40160
40150 PRINT "*";
40160 NEXT: PRINT: NEXT: PRINT#4: CLOSE4: RETURN
40200 FOR J=0 TO 24: FOR K=0 TO 39: X=PEEK(32768 + J*40 + K)
40210 IF X<32        THEN PRINT CHR$(X+96) ;: GOTO 40160
40220 IF X>31  AND X<91  THEN PRINT CHR$(X)     ;: GOTO 40160
40230 IF X>128 AND X<160 THEN PRINT CHR$(X-32) ;: GOTO 40160
40240 IF X>159 AND X<219 THEN PRINT CHR$(X-128);: GOTO 40160
40250 PRINT "*";: GOTO 40160
```

This routine separates lower-case mode (40200 ff.) from upper-case, and is therefore a general purpose routine. Changing the range of K from 0-39 into 0-79 makes this usable for an 80-column machine. Note, though, that BASlC 1 in lower case has its upper and lower cases reversed, so programs written in BASIC 1 tend to yield odd displays, and odd dumps, when run on other ROM machines. Note that the BASIC subroutine above can be compressed, with loss of clarity. Lines 40110 to 40140 can be replaced by:

```
40110 IF X<65 OR (X>128 AND X<193) THEN PRINT CHR$(X-(X AND 128)
         -2*(X AND 32)+64);: GOTO 40160
```

and lines 40210 to 40240 by:

```
40120 IF X<91 OR (X>128 AND X<219) THEN PRINT CHR$(X-(X AND 128)
         -96*((X AND 96)=0));: GOTO 40160.
```

---

*CPUCN nos. 6 & 7 for example have a routine (lower-case mode only) by J Allason and M. Bennet.

Screen dumps in machine code are newer. K Finn, ('Micro', Aug '80) has a CBM printer version for BASICs 1 and 2. C Brannon ('Compute!' Nov/Dec '80) and E Brannon ('Compute! Mar '81) have versions for BASIC 2 and 1 respectively. The first uses SYS to print the screen; a variable number of lines may be selected. The second changes the interrupt, so a simple key-stroke will print the screen. This is valuable if for some reason (e.g. a program in machine-code) a SYS command can't be issued, or if, in direct mode, the SYS command spoils the screen's appearance.

My routine below (see elsewhere for rationale) is a short relocatable dump for any printer; graphics characters*are treated as '#'. Upper and lower case settings are allowed. It saves temporary values in RND's workspace. It is often helpful to have such a routine available, perhaps in a cassette buffer. To use it, open the printer: OPEN 4,4: CMD 4: SYS 826: PRINT#4: CLOSE4 is an example, when the routine starts at $033A (=826). Three locations are marked; these are all user-modifiable.

This routine calls one ROM address, which prints carriage return and line feed. It is written for BASIC2. There is no serious difficulty in writing it to run on either machine ... however, as it stands, BASIC4 requires the following substitute lines:

```
.:  6020 A9 01 85 89 20 DF BA A2
.:  6060 8C D0 AD 4C DF BA.
```

B*   <u>RELOCATABLE BASIC 2 SCREEN DUMP</u>.

```
    PC  IRQ  SR AC XR YR SP
.;  0401 E62E 32 04 5E 00 F8

.
.:  6000 A9 00 85 89 85 8A 85 8B
.:  6008 A9 80 85 8C A9 40 85 88
.:  6010 E6 89 A5 89 C9 29 D0 0F   $29 = Max.Cols. + 1
.:  6018 E6 8A A5 8A C9 19 F0 43   $19 = Max.Lines to be output
.:  6020 A9 01 85 89 20 E2 C9 A2
.:  6028 00 A1 8B 29 7F 24 88 D0
.:  6030 06 24 7E F0 13 D0 21 24
.:  6038 7E D0 09 48 A9 02 2C 4C
.:  6040 E8 D0 0D 68 A9 23 D0 10   $23 = Default character (#)
.:  6048 48 A9 02 2C 4C E8 D0 05
.:  6050 68 09 40 D0 03 68 09 60
.:  6058 20 D2 FF E6 8B D0 B1 E6
.:  6060 8C D0 AD 4C E2 C9
```

(2) Variable <u>Dumps</u>. The best known implementation is the Toolkit's version; it and related routines dump ordinary (string, integer, and floating-point) variables, but not arrays, which are thought to be too difficult. There is no difficulty writing such routines in BASIC; and in any case values can simply be printed. Generally dumps are designed to print to the screen; diverting output to a printer may produce oddities. There is a published example of this type of dump in Compute! (3#1, Jan '81) by F Levinson. This works by putting 12 bytes into the output buffer:

    " A =" A ;0   (the zero is intended to represent a null character). The variable names are changed in cyclical sequence, through A,A0-A9, AA-AZ,B,B0-B9,BA-BZ,...A%,A0%-A9%,AA%-AZ%,..., and at each loop the variable is sought, using the ROM routine for the purpose. When a variable is found, the buffer is printed; the print routine determines the value of the variable, and the name in quotes is printed verbatim.

    An alternative type prints the variables in the order in which they are stored in RAM, in other words in their order of first use by the program.

---

*Including the shifted space character.

END

BASIC command

PURPOSE: Causes a program to exit to immediate mode. The Ready message is printed. This command may be used to set breakpoints in BASIC programs. CONT causes a program to continue at the next instruction after END.

Syntax: END has no parameters. It may be followed by spaces, and must be followed by an end of statement byte - either a colon or a zero byte at the end of the line.

Modes: Direct and program modes are both valid.

Examples:
```
20000 PRINT#4,CHR$(12): CLOSE 4: SYS 45056: SYS 739: END
855 IF PEEK(32766)<>TR THEN PRINT "*** TRACK READ ERROR": END
5000 GOSUB 51000: END: GOSUB 58000:END: GOSUB 15000: END
59999 END
```

Several related facets of the END command are shown here. The first program line is part of an exit routine, which tidies up the program before ending; a control character resets the printer which is then closed, and the disk unit is reconnected and a RAM routine called. (None of this is standard to Commodore). The second example shows an error-trapping line of BASIC which stops the program if a condition is not met: in the actual example, a test location which holds an incorrect track number causes execution to end. The third example is not from a finished program, but illustrates a way to use breakpoints. Each subroutine performs some initialisation function: lowering the top of memory, allocating variables in memory in an efficient order, poking machine code. In the final version no ENDs will be present here, but during testing each routine can be separately checked, using CONT to continue with the next. The last example uses END to ensure that subroutines - located at 60000 and after- are not inadvertently entered.

Notes: [1] Some BASICs require an END at the physical end of a program, even if it ends invariably somewhere else. (The last line might be GOTO 1, say). This is carried over from the days when programs were held as stacks of cards, and it was important to separate the programs in a box of cards.

[2] END leaves the program in memory: other exits, such as calling ROM routines to clear RAM, can be employed, if for example it is feared that lines from the program might be accidentally deleted in direct mode.

Abbreviated entry: eN

Token: $80 (128)

Operation: This routine is shared with STOP; the only difference is that the carry bit is set on entry to the routine by STOP or by the stop key, but is cleared for END. This flag (the carry bit) determines whether the message "BREAK IN" plus linenumber is printed. With END, of course, it isn't. After the usual syntax check, the routine tests the mode: if it is direct mode it skips past several instructions which save two parameters for CONT, the linennumber and the current CHRGET address pointer. The routine now throws away two bytes from the stack, since it wishes to enter direct mode, and does not need the return address. In the case of END it prints "READY." after loading pointers to the BREAK IN .. text stored in memory, which of course are unused. Early BASICs set the I/O device to 0, or keyboard, but BASIC 4 does not, so presumably CONT may be entered from non-CBM devices.

All the ROMs process this instruction in similar ways; the test used in BASIC 1 for direct mode is different, though, because the input buffer is in its zero page.

ROM entry points:BASIC 1: $C71E (50974) 2: $C741 (51009) 4: $B7C8 (47048)

EXP

BASIC arithmetic function

PURPOSE: Calculates e (2.718281828...) raised to any power within the range
-88 to +88 approximately. The result is always positive, approaching zero for
large negative powers, and increasing indefinitely for large positive powers.
EXP(0) is 1.

Syntax: EXP(arithmetic expression). If the expression evaluates to a value larger
than about 88, ?OVERFLOW ERROR will result and the program will end. If the
expression is a large negative number on evaluation, there is no equivalent
underflow error message; the value is simply set to zero.

Modes: Direct and program modes are both valid.

Examples:

```
PRINT EXP(10):        REM PRINTS 22026.44 ...
Y=EXP(1):             REM ASSIGNS Y VALUE OF E = 2.7182818 ...
PRINT EXP(LOG(N)):    REM PRINTS N (POSSIBLY WITH ROUNDING ERROR)
100 FOR N=0 TO 20: P(N)=(M N)*EXP(-M)/FACT(N): NEXT: REM POISSON
200 NT = NE * EXP(-B*EXP(-K*T)):              REM GOMPERTZ
```

Like SQR, this function is a special case of the power function, and therefore
is strictly speaking unnecessary. EXP(Q) can be replaced by $2.7182818 \wedge Q$.
But like SQR it is more easily recognisable in its familiar form EXP; familiar,
that is, to the mathematically-minded.

The two first examples are straightforward evaluations. The third reveals
or underlines the fact that EXP is the converse function to LOG, which is
calculated to base e. Whenever a logarithmic transformation has been used,
perhaps to reduce the magnitude of the numbers being dealt with, EXP can
reconstruct the solution, provided that it is within the limits accepted by
the PET's floating-point logic.

The final examples are both formulas; EXP invariably is used in scientific or
statistical calculations. The first such example is a statistical one; the
Poisson probability distribution deals with randomly occurring, rare events.
Given the mean number M of such events (misprints per page, say) line 100
computes the probabilities of 0,1,2, ...,20 such events happening. It uses a
function definition FACT(N) which is N! or N*(N-1)*(N-2)* ... *1. Chapter
16 has more on this topic. Finally, we have a growth curve of the so-called
'logistic' or 'ogive' shape. This sort of thing turns up in population models.

Notes: [1] The number e has a large number of special properties. The rate of
growth of EXP(X), for example, equals EXP(X), so for small DX,
(EXP(X+DX)-EXP(X))/DX is about equal to EXP(X). Malthus' population
theory gives a result involving e, which accounts for the popular meaning of
'exponential growth'. The infinite series $1+x+x^2/2+x^3/6+...$ converges to
EXP(x). It (e) is irrational; only the first few terms appear to recur. *

Abbreviated entry: eX

Token: $BD (189)

Operation: Rather unexpectedly, this function does not call the power routine in
ROM, but uses its own series evaluation method. This involves the following
steps: (i) The argument is multiplied by $1/\log_e 2$. (ii) The result is tested for
range. (iii) The result is normalised into the range 0-1, saving the exponent
on the stack. (iv) The accumulators are interchanged. (v) The series routine
is called; this computes $2 \wedge (x/\log_e 2)$. (vi) The power of 2 is added back; the
result is now e$\wedge$(argument). All ROMs process this instruction similarly.

ROM entry points:

```
BASIC 1: $DEA0 (56992)
BASIC 2: $DEDA (57050)
BASIC 4: $D184 (53636)
```

---

*'Irrational' means, mathematically, 'not expressible as a ratio'.

FOR.. TO.. [STEP]

BASIC loop command

PURPOSE: Permits repetitive processing of all BASIC between a FOR variable ...
TO... [STEP] statement and the corresponding NEXT. When NEXT is
encountered, the loop variable is checked and, if it matches NEXT, added
to the value originally assigned to STEP. If the result falls within the
limits specified by FOR and TO, the loop continues with the statement
following the FOR statement. Otherwise, BASIC continues linearly with
the statement following NEXT.

The loop variable may be used as a counter, pointer, or subscript, and
may be changed within the loop. Step size defaults to 1.

Syntax: The full syntax is: FOR real variable = arithmetic expression TO arith-
metic expression [STEP arithmetic expression] . Constructions such as
FOR / UNTIL and DO / WHILE are not obtainable directly in BASIC, but
can be simulated by programming. Many FOR loops can coexist while the
program runs, and they are called 'nested' loops, unless NEXT doesn't
match FOR, in which case either a loop variable or variables will be lost,
or ?NEXT WITHOUT FOR ERROR appears.

NEXT, the end of the loop, has syntax: NEXT [real variable [,real var-
iable][,real variable] ...]. Square brackets denote optional variables.

Modes: Direct and program modes are both valid.

Examples:
```
FOR J=1 TO 1000: PRINT "*";: NEXT: REM J USED TO COUNT TO 1000
FOR J=1 TO 1000: PRINT J;: NEXT  : REM ACTUAL VALUE OF J USED
FOR J=1 TO 1000: NEXT:            REM DELAY LOOP; ABOUT 1 SEC
```

These three simple loops illustrate loop processing with about the minimum
possible code. In each case J is the loop variable, and in neither case is
it modified within the loop. Therefore, unless the Stop key is pressed,
each loop continues 1000 times. Whenever NEXT is met, J is incremented by
1, since 1 is the default value of STEP. On leaving the loop, J equals 1001.
Loops are often used in benchmarks, which provide *some* indication of the
speed of execution of a computer language. The third example takes about
a second; the same BASIC operating with the 6502 at a different clock speed
will take a proportionally longer or shorter time.

```
100 K=0: FOR J=32768 TO 32768+255: POKE J,K: K=K+1: NEXT
200 FOR J=33768 TO 32768 STEP -1: POKE J+1, PEEK(J): NEXT
```

Screen peeks and pokes are the subject of the next couple of loops; the
first puts 0 to 255 directly into screen memory, starting at the top of the
screen, so all 256 ROM characters appear. They appear differently in upper
and lower case modes, of course. The inclusion of K within the loop shows
one method by which variables can be made to change in step with each
other. This principle is quite useful. Line 100 can in fact be simplified,
eliminating K, by writing 100 FOR J=0 TO 256: POKE 32768+J,J: NEXT .
Line 200 is a memory-move routine, which shifts 1000 bytes of the screen
forward by one location. To do this successfully, it is essential to begin at
the top end and work back, since otherwise each byte will be obliterated by
the previous byte. This is the reason for the negative STEP parameter.
Try the routine omitting the negative step if you don't yet see this.

```
1000 FOR J=1 TO LEN ("ABCX£$"): IF IN$<>MID$("ABCX£$",J,1) THEN NEXT:
     IN$="!"
1010 REM J NOW EQUALS 1-6, THE POSITION OF I$ WITHIN THE STRING OF
     CHARACTERS WE'RE TESTING, OR J EQUALS 7 AND IN$ = "!"

2000 FOR Y1 = Y1 TO 9E9: IF Y1-Y>1 THEN PRINT#5,SOUTH$: NEXT
```

Two program extracts show how IF statements within loops can be dealt
with. The first tests input IN$ against the contents of a string. If IN$
is not found in the string, it's reset to a warning value. Otherwise, J now
equals IN$'s position within the test string; this may be useful in extracting
other substrings. Line 2000 is part of a graph plotting program: steps are

drawn southward, from Y to Y1, incrementing Y1 until the condition fails. Finally, we have an example of nested loops, in which J controls the step size between a 2-dimensional plot on the screen. I have assumed (see SET) that a function to draw single 'points' exists:

```
FOR J=60 TO 2 STEP -1: FOR X=0 TO 79 STEP J: FOR Y=0 TO 49 STEP J:
      POKE 0,X: POKE 1,Y: SYS 826: NEXT Y,X,J
```

Notes: [1] Loops in practice are quite easy to use; don't let the rather long list of notes efface this fact from your mind.

[2] Syntax. (i) A loop variable must be a simple real variable: FOR X% = 1 TO 9 and FOR X(5)=0 TO 10 both cause ?SYNTAX ERROR. (ii) A loop is always executed at least once, even though strictly, in standard BASIC, a loop like FOR V=10 TO 1:...:NEXT should be ignored. Apart from taking time to test, and thus slowing down benchmarks, the corresponding NEXT has to be found, and, in unstructured BASIC, this is impossible. So the example sets V to 1, then executes the contents of the loop once. (iii)Inclusive limits apply, so that: FOR J=0 TO 9: causes J to take values 0,1,2,...,9 and execute the loop ten times. (iv) for j=1 to 1E4: in lower-case mode is treated as 1 to 14.

[3] Accuracy. If the loop variable and the step size are each stored exactly, there will be a rounding error only with extreme values, so a loop will execute precisely under these conditions. Generally, integers and binary decimals are stored exactly, including the default step value of 1. For this reason, both FOR Q=1 TO 1000000: and FOR J=.5 TO 1000000 STEP .0625 execute perfectly, but FOR M=1 TO 1000: STEP 1/3 doesn't, as can be seen by including PRINT M in the loop. FOR M=1 TO 1000.1 will ensure the count is correct.

[4] Speed. When fine-tuning a program to run with as little delay as is possible, the contents of loops are an obvious candidate for examination. Firstly, the variables: the loop variable itself is held by the stack as a pointer, so if it is used merely as a counter there is no point in putting it early on into the RAM variables. The rule should be to order variables in RAM according to their presence *inside* the loop. When loops are nested, the innermost variables obviously should have priority over those within fewer loops. The more variables a program has, the more difference this will make. Time-saving can be more spectacular with the second approach, rewriting the loop(s) to use fewer instructions, or fewer redundant operations such as assignments, calculations, or conditions. It is easy to compose examples showing many faults, and a large speed increase when these are removed, but again, in practice, factors of the order of five or six times the original speed are not very likely to occur. Let's consider an example incorporating both these factors:

```
7600 REM DATA IS STORED IN RAM IN BATCHES OF 116 BYTES, STARTING AT $6C00.
7610 REM   SO RECORD NO. R% STARTS AT 27648 + 116*(R%-1).
7650 O$(1)="":O$(2)="":O$(3)="":... :REM O$() HOLDS OUTPUTS
7660 FOR J=0 TO 27: O$(1)=O$(1)+CHR$(PEEK(27648)+ 116*(R%-1) + J)): NEXT
7670 FOR J=28 TO 47:O$(2)=O$(2)+CHR$(PEEK(27648)+ 116*(R%-1) + J)): NEXT
7680 ...
```

This program extract is perfectly good and workable, but, owing to BASIC's restrictions, the decision to rewrite it to run faster may be worthwhile. If so, we see that each loop holds a considerable amount of calculation, which can be moved out of the loop, and performed once only. We can use a temporary string in place of the arrays, which will process faster; and we can ensure that the variables are arranged in the optimum order. We get:-

```
7650 RS=27648 + 116*(R%-1): S$="" :REM RECORD START POSITION AND STRING
7660 S$="": FOR J=0 TO 27: S$=S$+CHR$(PEEK(RS+J)): NEXT: O$(1)=S$
7670 S$="": FOR J=28 TO 47:S$=S$+CHR$(PEEK(RS+J)): NEXT: O$(2)=S$
7680 ... :REM BEST ORDER FOR THIS CODE IS S$="":RS=0:J=0 WITH O$() 1ST ARRAY
```

S$ is the most important variable in the rejigged code, because it occurs twice as often as any other variable. R%, which was very influential in the original, now is unimportant as far as this part of the program goes.

[5] <u>Nested and structured loops</u>. A nested loop has an appearance which may be represented diagrammatically like this:
And in a program like this:

```
FOR X = X1 TO X2: ...
  FOR Y = Y1 TO Y2: ...
    FOR Z = Z1 TO Z2: ...
    ...
    NEXT Z
  NEXT Y
NEXT X
```

```
┌──First variable
│ ┌──Second variable
│ │ ┌─Third variable
│ │ │  ...
│ │ └─NEXT
│ └──NEXT
└── NEXT
```

Each depth of nesting puts 18 bytes of information on the stack, and each NEXT moves the stack pointer back 18 bytes. FOR and GOSUB share the stack; there are limits on the ways they can be used together. Every new FOR variable is checked against the current stack contents, and, should an active FOR loop exist already with that variable, the stack pointer is reset to that previous loop, erasing subsequent loops in effect. Several 'nests' can be built within a larger loop, and this is perfectly legitimate and should give no bugs:

FOR X=X1 TO X2: FOR Y=Y1 TO Y2: FOR Z=Z1 TO Z2: NEXT Z,Y: FOR A=A1TOA2: FOR B=B1 TO B2: NEXT B,A,X

Omission of the loop variables from NEXT (i.e. NEXT:NEXT and so forth) guarantees correct nesting. Structured programming has several things to say about loops; one is that there should be one exit point only, and not jumping from the middle of a loop to another part of the program. Another is the requirement for an explicit exit condition at the start of the loop, to make it more readable. The following skeletal loop shows how both of these ends can be achieved. It is a DO...UNTIL loop, starting with its loop variable set to BEGIN and with an arbitrary upper limit.

```
100 OK=-1: FOR J=BEGIN TO 9E9
110 IF NOT OK THEN J=9E9: GOTO 200
    ... PROCESSING ...
150 IF ... THEN OK=0: REM TEST
    ... PROCESSING ...
200 NEXT J
```

[6] <u>Bugs</u>. (i) Omission of a negative step: FOR J=100 TO 0: A(J)=J: NEXT
(ii) Omission of NEXT. There is no 'next omitted' error. FOR H=1 TO HRIZ: FOR V=1 TO VERT: GOSUB 1000: NEXT. Both these errors cause loops to end much more quickly than in the correct version. This may also happen with (iii) Inadvertent change in the loop variable; this is particularly liable to happen with subroutines in the loop - see GOSUB for examples.
(iv) The loop variable(s) may be omitted by mistake: FOR I=0 TO A: FOR J=0 TO B: X(A,B)=A*B: NEXT J,I needs X(I,J)=I*J in place of the expressions in A and B if the object is to fill the array with the product of row*column.
(v) An incomplete GOSUB (i.e. with RETURN not yet made) will give ?NEXT WITHOUT FOR, for example: 10 FOR J=1 TO 10: GOSUB 20: END / 20 NEXT
(vi) The upper limit of the loop is stored in the stack; therefore the attempt to vary the exit from the loop by controlling it will fail (unless the stack itself is altered). Example:

```
100 A=10: FOR XX= 1 TO A: REM i.e. 1-10
110 INPUT A: PRINT A     : REM CHANGE A...
120 NEXT: REM BUT LIMITS REMAIN 1-10.
```

(vii) Use of nonexistent loop variable will give ?NEXT WITHOUT FOR; so will NEXT without a loop variable if previous loops do not exist any longer or never existed. 0 FOR I=1 TO 10: NEXT II uses a non-existent variable; 0 NEXT has no corresponding FOR; and 0 FOR I=1 TO 10: FOR J=1 TO 10: NEXT I: NEXT eliminates J by its reference to I, so nothing is left for NEXT.

[7] <u>Logical variables</u>. DO WHILE loops can be simulated like this:

FOR J = -1 TO 0: ... : J = TEST EXPRESSION: NEXT

Where the omitted processing is performed until J becomes false.

Abbreviated entry: fO   stE. There is no short form of 'TO'.   NEXT has nE.

Tokens: FOR $81 (129)   TO $A4 (164)   STEP $A9 (169)   NEXT $82 (130)

Operation: See NEXT for operation of the stack and ROM entry points.

FRE

BASIC arithmetic function

PURPOSE: Computes the number of bytes available to BASIC between the end of the array storage and the start of strings. FRE first performs the so-called 'garbage collect' routine, which rearranges all the strings held in upper RAM into one consecutive block. This is useful when dealing with strings and string arrays, because (unlike numerals) they occupy variable space in RAM. This function measures the free memory.

Syntax: FRE(expression). FRE is a function in the sense that it returns a value. However, its expression is a dummy. Typically, PRINT FRE(0) or F=FRE(1) may be used. But FRE(X), FRE(X$), or FRE(A%(5)) are syntactically correct versions of the function.

Modes: Direct and program modes are both valid.

Examples:
```
PRINT FRE(0)
1000 PRINT FRE(0) "BYTES AVAILABLE AT PRESENT"
200 X=FRE(0): DIM Q$(75): PRINT (X-FRE(0)) "BYTES USED BY POINTERS"
```

The first two examples, in direct and program mode respectively, simply print the free memory. The third is a more elaborate piece of code which demonstrates the use of FRE to measure the differences before and after some memory-using statement(s). This example prints the amount of RAM taken up with the pointers for a string array of dimension 75.

Notes: [1] This diagram illustrates the situation. If a new string is defined which, even after garbage collection, is too long to fit into RAM, an ?OUT OF MEMORY ERROR message is printed.

Top of BASIC RAM

BASIC	variables	arrays	FRE memory	strings
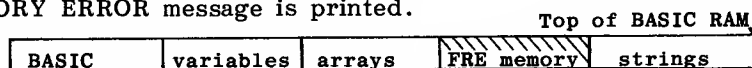
This example program shows how rapidly RAM can be used; this is part of an input routine which gets single characters in order to exercise greater control over the permitted input than is allowed by INPUT:

```
5 GET X$: IF X$="" GOTO 5
10 I$=I$+X$: GOTO 5
```

Each X$ takes one byte, and each I$ occupies one more byte than it did previously. A string of length n takes $\frac{1}{2}n(n+3)$ bytes, using a little algebra. So for example a 20-character input occupies 230 bytes.

[2] With no program in memory, FRE returns the number of bytes after the end of the program; so after Commodore's BASIC message, and (say) 31743 bytes free, PRINT FRE(0) prints 31740 or 31741 depending on the ROM. Lowering the top of memory by POKEs will reduce the number of bytes returned by FRE.

[3] Timing. This is a well-known problem associated chiefly with BASIC 2. A program using DIM X$(512) will intermittently stop to garbage collect, whenever string space is short, not only on executing FRE, and the process is slow. (BASIC 1 has the same problem; but people were cautious of large arrays, which didn't work correctly). The time taken to free memory is a function of the number of strings in upper RAM; it is a surprisingly precise relationship, and is about $.00008 * (N+11)^2$ seconds with BASIC<4. See Chapter 4 on ways of minimising this delay. One of the features of BASIC 4 is that the strings are held differently and freed more quickly. Chapters 2 and 4 give details. The following program, which can be entered in direct mode, is about the worst case with BASIC 2, and runs in 83 minutes:

```
DIM A$(7900):FORJ=0TO7900:A$(J)=CHR$(1):NEXT:T=TI:J=FRE(0):?(TI-T)/60 "SECS"
```

Operation: The function firstly frees memory by calling the garbage collect subroutine. It then subtracts the pointer to the end of the arrays from the pointer to the bottom of the strings, and converts the result into floating point in accumulator #1.

ROM entry points: BASIC1:$D264 (53860) BASIC2:$D259 (53849) BASIC4:$C4A8 (50344)

GET & GET #*

BASIC input command

PURPOSE: GET and GET# read a single byte from the keyboard and from any
device, respectively. In the case of the keyboard, if there is no character
in the keyboard buffer a null string or numeric value zero is returned. On
entering GET or GET#, the status byte ST is set to zero; the end of a
correctly CLOSEd tape file sets ST to 64, and the end of a correctly CLOSEd
CBM disk file sets ST to 64 and in addition sets the byte read by GET# to
Carriage Return.

Syntax: GET [#arith. exp.] var. name [,var. name][,var. name] ...
GET may optionally be followed by # with a logical file number which must
evaluate to 1-255. At least one variable name must follow. The processing of
GET resembles INPUT (q.v.) in its use of the input buffer, but no extra
parsing is carried out on GET's single byte, so this command may be used to
input any data, unlike INPUT which presumes certain formatting conventions.

Mode: Program mode only. Direct mode generates ?ILLEGAL DIRECT ERROR.

Examples:
```
5 GET X$: IF X$="" GOTO 5
10 PRINT "[UP]"X$"    [LEFT][LEFT][LEFT]" ASC(X$): GOTO 5
200 GET A$,B$,C$: PRINT A$B$C$: GOTO 200
```

If you're uncertain about the function of GET, these examples, when RUN, will
soon give you the idea. The first prints X$ and its ASCII value at a fairly fixed
position on the screen, where X$ is the single byte returned by GET. You will
be able to observe how GET can accept a carriage return, for instance, which
has the ASCII value 13. This is an infinite loop which Stop can terminate. Line
100 is a similar loop. The syntax is more appropriate to GET#; however, if you
are quick, more than one variable will be set from the keyboard. The method of
line 5   is necessary if a keypress is awaited. It is the starting-point for
crashproof input routines; see Chapter 4 on this topic.

       55 GET A   is valid. However, apart from 0-9 which set A=0-9 as expect-
ed, ?SYNTAX ERROR is printed, or ?EXTRA IGNORED with , and :. Also,
space,+,-, and E return 0. It's usually best to GET a string variable.

```
2000 GET#8,X$: IF ASC(X$)=13 GOTO 3000: REM STRING IN$ IS COMPLETED
2010 IN$=IN$+X$: GOTO 2000                : REM BUILD STRING IN$
```

This example shows how a string is built up from successive bytes.

Notes: [1] The Keyboard Buffer. GET (provided that an input device number is not
found by $FFE4) takes one character from the keyboard buffer. (Characters
are put there during IRQ servicing). This buffer occupies 10 bytes from $026F
(623 ff. dec.), and $9E (158 dec.) indicates how many characters are present;[2]
if 0, the null character is assigned to a string variable. BASIC 4 has a variable
length buffer: $E3 holds its greatest length. LINENO GET X$: IF X$>""GOTOLINENO
empties the buffer. So does POKE 158,0 although this is reversible: POKE 158
with some non-zero number revives characters in the buffer. In fact, poking
158 with 200 in direct mode prints the entire contents of cassette buffer #1.
Apple has a different and inferior GET which waits for a keypress. The short
routine which follows can be used to test any BASIC for keyboard buffering:

```
10 FOR J = 0 TO 3000: NEXT: FOR J = 1 TO 20: GET X$: PRINT J;X$: NEXT
```

When RUN, this delays for a few seconds, then GETs and prints out characters.
If several keys are pressed in turn during this delay, they will, with CBM
machines, be printed later, showing that a buffer exists. The buffer can hold ten
keys, and it it easy to demonstrate that BASIC<4 erases the buffer and starts
over if more keys than this number are pressed. This has a practical effect
on crashproof input routines. Note that BASIC 4 retains earlier keys, and
its buffer need not be 10. POKE 227,0 for example locks out the keyboard
altogether.

---

*Unlike INPUT# and PRINT#, GET# has no separate token, so I've treated it with GET.
[2] BASIC 4's keyboard buffer is set to 10 characters on power-up, but it can be changed
by a poke. BASIC 1's buffer begins at $020E, and contains $020D bytes at any instant.

[2] Tape. The tape reading routine is part of $FFE4. It can be recognised in ROM after the point where the input device number is compared with #3. After this point, the carry bit is clear for both tape devices, which are numbered 1 and 2 by Commodore. The character is taken from the cassette buffer (i.e. 192 bytes from $027A and $033A). When the buffer has been read, everything pauses while the next block from tape is read into the buffer, and its pointer reset to start. The end-of-file marker is a zero byte, which will cause ST to be set to 64 as the last character is read. If this is not detected, the next GET# (or any other input/output command) resets ST to zero, so the cassette will keep reading further data.

[3] Disk. Whenever ST is set non-zero, a carriage return character is sent, except with BASIC 1, which sets ST but returns the previous byte. It is not only EOI (end of file, in effect) which sets ST; time out on read has the same effect, so slow devices may send only carriage returns. The time-out feature can be disabled in BASIC 4 (by POKEing $03FC with a negative amount, e.g. POKE 1020 ,128). Typically, therefore, this type of routine is used with GET#:

```
2000 IN$=""
2005 GET#8,X$: IF ST=64 OR ASC(X$)=13 GOTO 3000: REM EXIT WITH IN$
2010 IN$=IN$+X$
2015 GOTO 2005
```

[4] Since GET# takes in colons and commas and so forth, it can be used to check a file's contents in a way impossible with INPUT#. Moreover there is no limitation to 80 characters length, although a built-up string like the one in the earlier example cannot exceed 255 characters in length. BASIC<4 include carriage returns when using GET# from the screen; each GET# which was a multiple of 40, e.g. the 40th, 80th, etc., became CHR$(13). But GET# from the screen is rarely used. Note also that the difference between GET and INPUT as regards cursor flashing is determined by the number of bytes in the keyboard buffer, but this may be overridden by POKEing the cursor flash location with the value zero.   This location is $A7 (167) in BASIC>1, and $0224 (548) in BASIC 1.

Abbreviated entry: gE & gE#

Token: $A1 (161)

Operation: GET and GET# use the input buffer, but place a zero byte into $0201 so that a single character only is taken. The 'get' part of the routine uses the kernel routine at $FFE4, which returns with a character in A and with ST possibly set. There is also an assignment part to the routine, which shares ROM with READ and INPUT. If the '#' symbol is found, the number or expression after it is worked out and this logical file number is stored for future use in $03 (BASIC 1), $0E (BASIC 2), or $10 (BASIC 4). When the byte has been fetched, normal device input/output is restored.

ROM entry points:

```
GET:BASIC 1:$CA9F (51871) KEYBOARD BASIC 1:$E2B7 (58039)
    BASIC 2:$CA7D (51837)   FETCH: BASIC 2:$E2B8 (58040)
    BASIC 4:$BB7A (47994)          BASIC 4:$E003 (57347) - NEEDS SEI.
```

GO

BASIC dummy command

PURPOSE: Sole function is to look for a matching TO, and, if found, to perform GOTO. The raison d'etre is to provide GO TO as well as GOTO in BASIC.

Syntax: GO must be followed by one or more spaces, TO, and a linenumber.

Notes: [1] BASIC 1. This token is not present in BASIC 1; this early version had the facility to eliminate spaces on tokenisation, so that GO TO converted itself to GOTO. This method of forming tokens leads to more ambiguities than the later method. Possibly for this reason, it was changed, so that a line like:

    10 IF 256=LE THEN PRINT "HIGH"

no longer appeared to contain LET. However, GO TO was also eliminated, and a patch put in, consisting of the token GO and a special routine to check that it was followed only by TO.

It follows that programs developed on later machines, using GO TO, will not LIST properly with BASIC 1; GO produces ?SYNTAX ERROR.

[2] BASIC 4. GO is no longer a patch, but processed along with other tokens. Some versions appear to be defective. An early manual for this ROM states that an extra byte or token must appear between GO and TO to compensate for a bug: GOXTO for example, or GO TO TO.

[3] GO can be intercepted by a wedge and used perhaps as a command in a computed GOSUB or GOTO routine. See Chapter 14, section 14.3.2 .

[4] GO causes problems with some renumber utilities, which haven't allowed for the existence of this token.

Abbreviated entry: None

Token: $CB (203). Not present in BASIC 1.

Operation:In BASIC 2 the routine to execute a BASIC statement is at $C700. The patch to process this command is at $C721.

In BASIC 4, the entry point for GO is $B7AC (47020).

GOSUB
BASIC command

PURPOSE: Performs a jump to any line in a program. The target line is identified by its linenumber. When RETURN is next encountered, control is transferred to the statement following GOSUB. In association with IF or ON, conditional calls to subroutines may be made.

Syntax: GOSUB linenumber. The linenumber must be ASCII numerals (e.g. 1234), and, like GOTO, the first character outside the range 0-9 marks its end. Computed GOSUBs of the type GOSUB x need to be specially written. If the line doesn't exist within the program, a run-time error will occur.

Modes: Direct and program modes are both valid. A subroutine in a BASIC program in memory can be tested in direct mode.

Examples: i FOR V=0 TO 24: FOR H=0 TO 39: GOSUB 1000: NEXT: REM HORIZ & VERT POSNS
```
          1000 POKE 245,V: POKE 226,H: SYS 58843: RETURN: REM FOR BASIC 1
```

```
     ii 2024 IF RIGHT$(JS$,1)<>CD$ THEM EM$="IN CHECKLETTER": GOSUB 12000
          ...
```

```
    iii 12000 PRINT "[HOME][23 DOWN][10 RIGHT][RVS]*** ERROR " EM$ " [RVSO]";
        12010 FOR J=0TO2000: NEXT:                : REM DELAY LOOP
        12020 FOR J=1 TO LEN(EM$)+11: PRINT "[LEFT] [LEFT]";: NEXT
        12030 GOSUB 100:  FOR J=1 TO JL: PRINT " ";: NEXT: RETURN
```

```
     iv 500 GOSUB 510
        510 REM ** SUBROUTINE TO BEEP BELL ONCE **    (Detail omitted)
```

i. This first example shows how a subroutine may be called in direct mode. Line 1000 is a subroutine which positions the cursor, using 2 parameters, H and V. The direct mode line performs an exhaustive test on it.

ii. The same piece of code may be required in many different places within a program. This use of subroutines - one of the most important - is exemplified by line 2024: on discovery of an error in a check digit, the parameter EM$ is set to a suitable value, and the subroutine called. In other parts of the program the identical subroutine is called, but EM$ takes other literal values: "IN SALES CODE", "- NOT ON FILE", and so forth.

iii. This four line routine prints an error message in reverse on the bottom of the screen, and erases it after about 2 seconds. Then, in line 12030, it calls another subroutine, which in fact moves the cursor to the position on the screen which the operator is using for input. The erroneous string, of length JL, is erased ready for reinput.

iv. This is a simple example of the use of subroutines with multiple entry points. GOSUB 510 beeps the speaker; GOSUB 500 beeps it twice.

Even when code is used by only one part of a program there are many situations in which subroutines improve the total program. Here are some examples:

v. Programs written in a structured or semi-structured fashion can have controlling routines written like this:

```
        7000 IF JS$="S" THEN GOSUB 2000: GOTO 6000: REM SKIP TO NEW ITEM
        7010 IF JS$="B" THEN GOSUB 3000: GOTO 6000: REM BOOK STOCK IN
        7020 IF JS$="E" THEN GOTO  4000: REM EXIT AND CLOSE DOWN
```

vi. Any routine which is too long for one line, or requires multiple IFs or other confusing constructions, may be easier to deal with as a subroutine.

vii. Batches of similar routines may be clearer when written as subroutines, so that a block of the program collects together in one place a set of closely related procedures.

```
        600 PRINT "(";: GOSUB 400: PRINT ")": RETURN: REM INDIRECT JUMP
        610 GOSUB 500: PRINT ",Y": RETURN:            REM ZEROPAGE,Y
        620 PRINT "(";: GOSUB 500: PRINT ",X)": RETURN: REM (ZEROPAGE,X)
```

Notes: [1] Linenumbers following GOSUB are dealt with by the scanning routine which GOTO also uses. The effect is similar to the VAL function. This incomplete validation allows ON ... GOSUB to function, since a comma has to be treated as a marker for the end of linenumber. It permits some odd anomalies, which also occur with GOTO. For example, all the following commands are interpreted GOSUB 0:-

```
GOSUB            GOSUB REM NEW            GOSUB 0xxx            GOSUB [PI] ,
and
GOSUB 1000NEXT        GOSUB 50*2
```

are interpreted GOSUB 1000 and GOSUB 50 respectively.

[2] Timing: Since subroutines can be called from any part of the program, it is desirable from the speed point of view to put the most commonly used of them at the start of the program. This minimises search time for the linenumber. (RETURN stores a pointer to the original GOSUB, so there is no search time spent in RETURNing). Program structure of this type is therefore common:

0	GOTO 5000
100+	Standard subroutines
1000+	Menu options 1,2,3,...
5000+	Initialisation
	Menu for all options
	Closedown and end
50000+	Initialisation, closedown,
	and utility subroutines.

[3] Note that GOSUB 500: RETURN has the same effect as GOTO 500.

[4] See text for computed GOSUB routines.

[5] It is sometimes useful to escape from a subroutine without returning to the previous GOSUB. See POP in this reference section for details.

[6] A program with subroutines is inevitably fragmented into discrete chunks, so subroutines may need to be isolated from the remaining program to prevent dropping-through and execution of subroutines at the wrong time. For example, with subroutines starting at 60000 the line 59999 END guards against this eventuality. Subroutines can call themselves, but an exit mechanism of some sort is necessary. 100 GOSUB 100 for example will generate an ?OUT OF MEMORY ERROR as the stack fills up with return addresses. When handled correctly, this technique is called 'recursion'. It is used widely in translaters and compilers. Incidentally, the claim that 23 levels of subroutine can be handled by CBM BASIC should be treated with caution. All intermediate results, and loops, are pushed on the stack, so a subroutine with loops and many parentheses may unexpectedly run out of memory with far fewer than 23 subroutine levels.

Abbreviated entry: goS   **Token: $8D (141)**

Operation: The stack is tested. If there is not room for 6 bytes an ?OUT OF MEMORY ERROR message appears. (Although it only uses 5). Assuming this test is passed, 5 bytes are pushed onto the stack: the current CHRGET address, the current linenumber, and a GOSUB token ($8D). After this its operation is identical to GOTO. It scans linenumbers in the same way as GOTO, either from the start of the program or from its current position, depending on the linenumber. Finally it carries out a BASIC warm start.

Stack use demonstration program:-
```
10 P=512: GOSUB 20
20 PRINT PEEK(P),: P=P-1: IF P=500 THEN END
30 GOTO 20
```

Gives:-
```
0       0       198     238
4       16      0       10
141
```
Token   Location   Linenumber
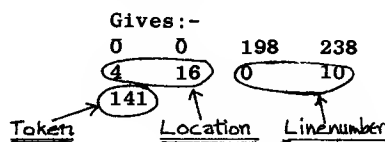
ROM entry points:

BASIC 1: $C780 (51072)   BASIC 2: $C790 (51088)   BASIC 4: $B813 (47123)

GOTO & GO TO

BASIC command

PURPOSE: Performs a jump to any line in a program. The target line is identified by its linenumber; not, for example, by a label. In association with IF or ON , conditional jumps may be made, selecting which part of the program to go to.

Syntax: GO TO or GOTO followed by a linenumber. The linenumber must be in ASCII (e.g. GOTO 1234). Computed GOTOs of the type GOTO x need to be specially written. Note that the linenumber is processed in a similar way to the VAL function; the first character not 0-9 is deemed to be the final character in the linenumber. Nonexistent lines cause ?UNDEF'D LINE ERROR.

Modes: Direct and program modes are both valid. Direct mode will cause a jump to the program in memory, and, provided the target line number exists, will execute the program from the point of entry. Since CLR isn't performed the variables set up by the program are unaltered: this command therefore resembles CONT and is usable after STOP, END, and the STOP key.

Examples: D$="022983": GOTO 12000
          100 GET X$: IF X$="" GOTO 100
          GO TO 100

The first example shows a direct mode GOTO statement. Before executing GOTO, a variable is set; in the example,with an invalid date, to test the operation of the program. Any line may be jumped to, including itself. The second example is a conditional loop which, until a key is pressed, loops indefinitely. Without the condition, line 100 will constitute an infinite loop, from which only the stop key will rescue the program. The third example illustrates that GO TO is an acceptable variant of GOTO.

Notes: [1] On the subject of the differences between GOTO and GO TO, see the reference page dealing with the GO token. Generally, GOTO is better.

[2] Some apparent anomalies result from the translator's method of dealing with the linenumber following GOTO. GOTO 1OIO, with I erroneously keyed in place of the numeral 1, does not produce a syntax error message, but is treated as GOTO 10 would be. The mistyped GOTOT10 is interpreted GOTO 0. And the solitary statement GOTO is taken to mean GOTO 0. By poking a null character into a linenumber, GOTO 200 may be made to LIST as GOTO 20 but act like GOTO 2.

[3] Timing: the time spent searching for the target linenumber is not on the whole large. (Some BASICs, notably Sharp, are far slower). However, to cut this time to a minimum, it's necessary to know how GOTO is processed. This is done as follows:
(i) The high bytes of the line numbers (and only the high bytes) are compared; (ii) if the target linenumber is larger by this test, lines <u>after</u> the current line are scanned; (iii) if the target linenumber is not larger - by the test - lines from the program's start are scanned. To take a concrete example: 25600 GOTO 0, 25600 GOTO 10000, and 25600 GOTO 25825 all have to scan BASIC from the beginning. 25600 GOTO 25856 and 25600 GOTO43000, on the other hand, scan forward from their current position.

[4] See the text for computed GOTO routines.

Abbreviated entry: gO (=GOTO)   Token: $89 (137)

Operation: The linenumber is fetched one character at a time and converted into a 2-byte integer. The process stops when a non-numeric character is found. The location of the next line is calculated. Now, in x GOTO y, the high byte of y is compared with the high byte of x: if larger, lines are sought from the next line. Otherwise, they are sought from the start of BASIC. If the line wasn't found, ?UNDEF'D LINE is branched to; otherwise, CHRGET is pointed to the zero byte just before the target line. RTS executes it.

ROM entry points:BASIC1:$C79D (51101) BASIC2:$C7AD (51117) BASIC4:$B830 (47152)

HTAB & VTAB

BASIC commands unavailable directly in CBM BASIC

PURPOSE: Moves the cursor to any position on the screen, as specified by horizontal and/or vertical parameters. This is sometimes called PRINT @.

Versions: This type of function is easy to write in CBM BASIC. All that is needed is a print statement including [HOME] and a suitable number of cursor down and cursor right characters. Machine code routines can also be written; they are faster than the BASIC equivalent, which may be important in some circumstances. For example, a formatted screen which inputs intensively validated information may well be improved by such a routine. It is quite easy to find the ROM routines responsible for handling this, since the reset routine, used at switchon, must format the screen at some stage. The drawback of machine dependence, though, has to be taken into account, because each ROM has its routines in a different place, set in silicon. BASIC 4 has two versions!

```
BASIC 1: POKE 226,H: POKE 245,V: SYS 58843: RETURN
BASIC 2: POKE 198,H: POKE 216,V: SYS 57949: RETURN
BASIC 4
(40 COL): POKE 198,H: POKE 216,V: SYS 57471: RETURN
BASIC 4
(80 COL): POKE 226,H: POKE 224,V: SYS 57439: RETURN
```

Note that the 8032 is more difficult to deal with because it has several types of screen editing. This version resets the top left corner of the scrolling window.

Demonstration: A demonstration program in BASIC follows. Line 1000 holds the machine-code subroutine, and corresponds to BASIC 2, but any of the routines listed previously can be substituted for it.

```
5 REM
6 REM **** RUN 10 USES SYS COMMAND TO POSITION CURSOR AT H,V ****
7 REM
8 REM **** RUN 20 PRINTS CURSOR CONTROL CHARACTERS ****
9 REM        NOTE THE DIFFERENCE IN SPEED
10 FOR V=0 TO 24: FOR H=0 TO 39: GOSUB 1000: PRINT "[shift &]";:
      NEXT H,V: END
20 FOR V=0 TO 24: FOR H=0 TO 39: GOSUB 1001: PRINT "[shift &]";:
      NEXT H,V: END
1000 POKE 198,H: POKE 216,V: SYS 57949: RETURN
1001 PRINT "[HOME]";: FOR J=0 TO H: PRINT "[RIGHT]";: NEXT:
      FOR J=0 TO V: PRINT "[DOWN]";: NEXT: RETURN
```

IF

BASIC conditional command

PURPOSE: Allows (i) Conditional branch to any program line,
              (ii) Conditional execution of statements following IF.

Syntax: IF arithmetic or logical expression THEN linenumber or statement(s)
                                      GOTO linenumber
THEN may be followed by a null statement: IF X=1 THEN: is valid.
On execution, if the expression evaluates to 0 it is treated as 'false' and no
further part of the line is executed; if it evaluates to any non-zero value
it is regarded as 'true'. This fact enables the conditional expression to be
arithmetic, not just logical with alternatives 0 and -1. See also note [1].

Modes: Direct and program modes are both valid.

Examples: FOR N=1 TO 1000 STEP .01: GOSUB 100: IF VAL(N$)=N THEN NEXT

This direct mode example is being used to test a rounding routine; if the
condition fails, the loop ends and PRINT N displays N's final value.

```
500 IF P=60 THEN P=0: GOSUB 30000: GOTO 600: REM PAGE THROW
700 IF X=1 THEN IF A=4 AND B=9 THEN PRINT"*";:REM SPECIAL VALUES
800 IF 7+6 GOTO 900
1000 IF 8 AND 7 THEN THIS IS NEVER REACHED!
1200 IF YN$="Y" THEN: $D,1 : REM BASIC WEDGE IN USE
```

This batch of examples illustrates most points relevant to IF. Firstly, its use
in conditional execution of BASIC: if 60 lines have been printed, the counter
is reset to 0, a subroutine to call form feed and print a new heading is run,
and the processing resumed. None of this is done if the condition was not
true. Line 700 contains a composite IF; this is entirely valid since THEN may
be followed by any statement. Note that 'IF X=1 AND A=4 AND B=9 THEN' is
exactly identical in its effect (but slightly slower). Line 800 causes an
unconditional branch to 900. This is because 7+6 evaluates to 13, which is
non zero. Line 1000 is the opposite; anything after THEN cannot be reached
by BASIC running normally. Finally, line 1200 demonstrates a point which is
sometimes important with wedges in BASIC which add extra commands. Here,
'$' signals a special instruction (disk directory with Compu/think disks) which
if intercepted by the wedge will carry out the command, even when the IF
condition is false. The colon, starting a new statement, prevents this.

Notes: [1] IF .. GOTOn is of course redundant; it can always be replaced by Ii ..
THEN n. However, it is *slightly* faster. Note that IF .. GO TOn is not valid,
while IF .. THEN GO TOn is! IF .. GOSUB n is not allowed, and must have
THEN. On the subject of syntax, note finally that GOTO doesn't validate the
linenumber fully, so that IF A=B GOTO 10XX will branch to line 10.

[2] IF X THEN... is the same as IF X<>0 THEN ... and vice versa.

[3] Rather strangely, a condition may include strings, which on 'evaluation'
may not use the floating point accumulator, so that the <u>previous</u> calculation
determines the 'truth' of the condition. Q$="":IF X$ THEN: is false, while
Q$=CHR$(I):IF X$ THEN: is true.

[4] Some BASICs, notably IBM's 8100 series, allow only IF .. GOTO, resulting
in exceptionally spaghettied programs. Apple integer BASIC skips to the next
statement, not *line,* after a false condition.

Abbreviated entry: None   Token: $8B (139)

Operation: This short routine evaluates the expression after IF, then checks for
GOTO or THEN. If one of these is found, the exponent of accumulator #1 is
examined. If zero, i.e. 'false', the next line is jumped to, using a routine in
common with REM. If non-zero, i.e. 'true', the next character is checked; if
it's a numeral, GOTO is called; if not, the next statement is executed.

ROM entry points: BASIC1:$C820 (51232) BASIC2:$C830 (51248) BASIC4:$B8B3 (47283)

INPUT

BASIC input command

PURPOSE: Provides users with an easily-programmed method to key in data from the keyboard to the CBM. INPUT accepts data from the keyboard and echoes it as output to the screen, unless the input/output devices have been changed, for example by CMD. INPUT# is an alternative form which is designed for input from tape or disk file storage. Input is terminated by the 'Return' key or by the ASCII character for 'Return'.

Syntax: The INPUT statement itself has this syntax:-

INPUT [string literal within quotes;] var. name [,var. name][,var. name]...
When RUN, this statement prints a question mark followed by a flashing cursor to prompt the user. The optional string is printed before the question mark onto the screen. Thus, INPUT X$ and INPUT "CODE";X$ are each valid. The first prints ? with the cursor, the second CODE? and the cursor. Subject to the rules which follow, the variable X$ will be assigned, on Return, the data typed after the cursor. Note that the optional string must be within quotes and is not a string expression. If X$="NAME", nevertheless INPUT X$;N$ generates ?SYNTAX ERROR, presumably to avoid confusion with INPUT X$.

The keyed-in data is processed according to these rules:-
(i) Alphanumerics are dealt with straightforwardly, but many characters are not, notably " , : Return and the screen editing characters. The quotes mark " sets a flag causing subsequent input to appear as a literal, so that Home and Delete for example appear as they would within a literal, without homing the cursor or deleting the previous entry. Other special characters, such as , and : may be incorporated into string input in this way. Carriage return, however, always terminates INPUT and turns off quotes mode. An opening quote, with or without a later closing quote, is therefore a valid entry in response to INPUT's prompt; but quotes in the middle of a string entry generate ?FILE DATA ERROR (or ?BAD DATA ERROR in BASIC 1). INPUT shares routines with GET and DATA and, like them, relies on the comma as a separator and the colon to mark the end of a statement. These are treated as separators with INPUT. ?EXTRA IGNORED will result if the separators seem to indicate that there are more strings of input than corresponding variables to assign them. The double prompt ?? is printed when INPUT has had fewer strings of input than it has variables. Leading spaces are ignored.

(ii) When the input doesn't match the type of variable to which it is assigned, ?REDO FROM START appears and the input is repeated. There are minor exceptions to this. An integer variable may be assigned a non-integer value without an error message, and a real variable may be assigned data in scientific format.

(iii) INPUT takes in all the characters following the prompt to the end of the line. Consequently it is difficult to use INPUT with a screen neatly boxed with graphics. (It can be done by editing the resulting graphics input out of the string). And the total length of the string is limited by the screen width to 39 or 79 characters, when a prompting string isn't used.

(iv) Finally, CBM's notorious input crash, which alone is sufficient to make an unmodified INPUT unsuitable for many applications. If 'Return' only is pressed in response to INPUT's ? BASIC prints 'READY.' and stops. It can be revived by CONT without loss of data. Actually, this is true only if no file appears to be open to INPUT, and like SPC(and TAB(, this feature can be changed by POKEs. See note [2]. Note: VIC has no input crash!

Mode: Program mode only. Direct mode generates ?ILLEGAL DIRECT ERROR.

Examples: 100 INPUT "NAME";N$
           110 INPUT "ADDRESS LINE 1 (NO COMMAS!)";A1$
           120 INPUT "ADDRESS LINE 2 (NO COMMAS!)";A2$

These are typical elementary input statements, easy to program but subject to serious drawbacks. 'Home' will home the cursor; 'Return' will crash the program; Shift-Stop will attempt to load a new program; the screen can be filled

with unwanted characters; commas or colons cause some of the string to be lost. See the notes for cures for these problems.

```
1000 INPUT AA$,BB,C%    :REM INPUTS MUST MATCH
2000 FOR J=0 TO 10: INPUT X$(J): NEXT :REM INPUT 10 STRINGS ...
2010 FOR J=0 TO 10: PRINT X$(J): NEXT :REM ... AND CHECK THEM
```

Line 1000 expects three inputs. This is a valid response:
```
HELLO!,-123.45,7.1
```
After Return, AA$="HELLO!", BB=-123.45, and C%=7. Integer assignments follow the normal rules as to range and rounding. -1.2 would be assigned -2. This response will produce ?EXTRA IGNORED:-
```
HELLO,12,-123.45,7
```
And this will produce ?REDO FROM START, because of the type mismatch:-
```
ABCDEF,19*12,4
```
One or two entries only will be accepted, if they're valid, and the double prompt of ?? will be printed on the next line, awaiting complete input. Lines 2000-2010 show how array variables may be used for input too.

Further examples showing use of (i) String literal, (ii) Keyboard buffer.
The following examples show some of the ways in which INPUT can be modified. The first four use screen editing characters to produce interesting variations on INPUT, including positioning on the screen, underlining, and reversed text. The fifth is a typical 'crashproofing' routine; sometimes * is used in place of upper-case (i.e. shifted) space. The sixth shows how characters may be inserted into the keyboard buffer, which is equivalent to keyboard entry *after* the prompt and cursor are printed. They offer the possibility of erasing the prompt and - as here - automatically entering " at the start of the input in order to force acceptance of strings with commas, etc. Where constructions like "LDA $8000,X" are common, this is quite useful. The extra " turns off quotes mode, so the screen editing facilities will operate.

```
10 INPUT "[CLR][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][RIGHT][RIGHT]
   [RIGHT][RIGHT][RIGHT][RIGHT][RIGHT]";X$: PRINT X$
20 INPUT "HELLO[DOWN][DOWN][DOWN][DOWN][RVS]";X$: PRINT X$
30 INPUT "[DOWN][DOWN]TEXT[UP][LEFT][LEFT][LEFT][LEFT]";X$: PRINT X$
40 INPUT " --------[LEFT][LEFT][LEFT][LEFT][LEFT][LEFT][LEFT][LEFT]
   [LEFT][LEFT][LEFT]";X$: PRINT X$
50 INPUT "CRASHPROOF NAME[USPC][USPC][USPC][LEFT][LEFT][LEFT]";X$:
   PRINT X$
60 POKE 158,3: POKE 623,34: POKE 624,34: POKE 625,20: REM 3 ITEMS IN
   KEYBOARD QUEUE, WHICH ARE 2 QUOTES AND A DELETE.(BASIC1:525 & 527ff)
70 INPUT X$: PRINT X$: REM X$ MAY INCLUDE , AND/OR :.
```

Notes: [1] See Chapter 4 for methods of foolproofing input using GET. Because INPUT can occur with screen scroll, if for instance many wrong entries cause the bottom of the screen to be reached, it's worth checking the result of an overflow: use, say: 10 INPUT "   VERY LONG MESSAGE   ";X$/15 PRINT X$/ 20 GOTO 10. BASIC 4 is different from BASIC<4.

[2] When CMD is in force, INPUT "MESSAGE";M$ will print the string to the device, so that MESSAGE may appear on a printer. ?FILE DATA ERROR means that INPUT is attempting to get data from a listener, such as a printer. When a file is open like this, the 'Return' crash won't happen: OPEN 1,0:INPUT#1,X$ for example inputs from a file to the keyboard. POKEing the device number location with a pseudo-file number has similar effects: try POKE 3,1 with BASIC 1, POKE 14,1 with BASIC 2, or POKE 16,1 with BASIC 4.

[3] Direct mode is prohibited because the buffer which holds the direct-mode commands is the same as that in which input characters are stored. You can however try direct mode: use SYS 51956 X$, SYS 51925 X$, or SYS48080 X$ with BASIC 1/2/4. This will attempt to assign X$ to your input. It misses the test for direct mode. SYS of the ROM addresses below works exactly like INPUT, except that it will not print a string; try e.g. SYS48062X$,Y%,Z.

Abbreviated entry: None   Token: $85 (133)

Operation: See INPUT#

ROM entry points: BASIC1:$CAE0 (51936) BASIC2:$CAC1 (51905) BASIC4:$BBBE (48062)

INPUT#

BASIC input command

PURPOSE: Provides users with an easy method to read back data from a storage device, normally tape or disk. The input which is read by the CBM is processed in the same way as INPUT processes it; this means that data sent to the storage device by PRINT# will be recovered intact. The format is consistent with that of PRINT# for strings and numerals, which are written as individual ASCII characters with carriage returns as separators. However some characters aren't recognized by INPUT# and are ignored; these include the screen editing characters, unless the quote character, CHR$(34), was written at the start of the string. Note also that 80 characters is the maximum length of a record recoverable with INPUT#.

Syntax: INPUT#arith.expr. ,var. name [,var. name][,var. name] ...
As with PRINT#, a space between the two parts of the keyword's name causes the interpreter to see two tokens instead of one (except in BASIC 1!). The arithmetic expression is the logical file number of the input file, and must evaluate to 1-255 with rounding down. The comma and at least one subsequent variable are also compulsory. No optional string exists, as it does with INPUT since a prompt is out of place when reading from tape, say.

The data which is read in is processed according to these rules:-
(i) Alphanumerics are dealt with straightforwardly, and Return, when it is read, i.e. as CHR$(13) or $0D, terminates a record, in exactly the same way that the Return key sends data from the keyboard. In an analogous manner commas or colons, if they were not preceded by a quote mark, are treated as separators, and the subdivisions of data which they separate are all assigned their own variables. There is no equivalent to ?EXTRA IGNORED, but nonetheless data will be lost if an INPUT# statement takes in data to the buffer which is subdivided into more parts than there are variables; this can only happen if commas and/or colons are used carelessly, e.g. with PRINT#1,CHR$(44) or PRINT#8,CHR$(58).

(ii) Most other errors cause the program to crash with ?FILE DATA ERROR or ?BAD DATA ERROR in BASIC 1. For example, this occurs with INPUT#1,X$ when X$=CHR$(32), because leading spaces are ignored. Similarly, when the data doesn't match its assigned variable, this error occurs.

(iii) The maximum string which may be input is constrained by the input buffer to 79 bytes (89 in VIC). BASIC 4 signals this condition with ?STRING TOO LONG ERROR and crashes the program; earlier BASICs hang.

Mode: Program mode only. Direct mode generates ?ILLEGAL DIRECT ERROR.

Examples:
```
10 OPEN 10,2,1,"TEN NAMES": REM OPEN TAPE FILE FOR WRITING TO CASS.#2
20 FOR J=1 TO 10: INPUT X$: PRINT#10,X$: NEXT: REM WRITE TAPE FILE
30 CLOSE 10: REM CLOSE FILE, I.E. WRITE FINAL BUFFER OF DATA.
100 OPEN 5,2,0,"TEN NAMES": FOR J=1 TO 10: INPUT#5,X$: PRINT X$: NEXT
110 CLOSE 5: REM INPUT# HAS EFFECT LIKE PRINT#, SO CLOSE IS NO PROBLEM
```

The above example shows a simple write-then-readback program, omitting tape rewind details and ST checks on INPUT#. The logical file numbers are arbitrary and different from each other to make things clearer. INPUT# is far less trouble than INPUT to use, because its data is already formatted in a known way.
```
10 OPEN 1,0:  REM OPEN FILE #1 TO THE KEYBOARD
20 OPEN 3,3:  REM OPEN FILE #3 TO THE SCREEN
30 INPUT#1,X$:REM INPUT FROM KEYBOARD IS SIMILAR, NOT IDENTICAL,TO INPUT
40 PRINT#3,X$:REM PRINT TO SCREEN FILE
50 GOTO 30
```

This next example shows how files can be opened to the keyboard and the screen. Because an input file (logical file #1) is open, the input crash on pressing 'Return' alone doesn't happen. A screen file is useful sometimes if CMD is being used; PRINT#3 sends output to the screen only. Input from the screen is similar to normal input, but the string may wrap round to the next

line, depending on the entry in the screen line table, with 40-column screens. The string length may be 39 or 79. If this interests you, replace line 30 with 30 INPUT#3,X$ and add 45 PRINT "LENGTH="; LEN(X$).

```
 5 SCRATCH "SEQ FILE",D1: DOPEN#2,"SEQ FILE",D1,W: REM OPEN FOR WRITE
10 FOR J=1 TO 10: X$="RECORD NUMBER" + STR$(J)   : REM MAKE UP DATA
15 PRINT#2,X$: PRINT DS$ ST: REM WRITE DISK + SHOW BOTH STATUSES
20 NEXT: DCLOSE            : REM 10 RECORDS WRITTEN

1000 DOPEN#1,"SEQ FILE",D1 : REM OPEN SAME FILE ON DRIVE 1 FOR READ
1005 FOR J=1 TO 10: INPUT#1,X$ :REM READ BACK WITH INPUT# COMMAND
1010 PRINT DS$ ST; X$       : REM PRINT RESULT + STATUSES
1015 NEXT: DCLOSE
```

This pair of programs is the BASIC 4 disk equivalent of the earlier tape program. Again, ten records are written with PRINT# and read back with INPUT#.

Notes: [1] How INPUT and INPUT# work. The buffer used by INPUT in CBM computers starts at $0200, immediately above the stack, and extends 81 bytes to $0250. This short routine enables you to see the buffer:

      FOR J=511 TO 592: PRINT CHR$(PEEK(J));: NEXT *

and typically there will be many fragments of lines, new and short lines overlaying earlier long ones. Each input chunk is terminated by a zero byte, which the little routine above won't show. When INPUT or INPUT# is running, each successive byte is put into this buffer. Eventually, carriage return is input, whereupon a zero terminating byte is put in and the buffer parsed by INPUT for commas and colons separating the buffer: each chunk is assigned a variable and numerical variables are processed in accumulator #1 before being stored further up in RAM. The 81st byte therefore may contain a zero. BASICs prior to 4 could write into RAM above $0250. This region holds the three tables of logical files, devices and secondary addresses, so overwriting them (unless by coincidence the data were identical) removed the record of live files and so crashed the program. Location $1FF holds a comma: this is to ease the task of the parser by making each chunk start in the same way, as BASIC is started with a zero 'end-of-line' byte. Taking an example from INPUT, we may have something like this: *

$01FF	$0200	-01	-02	-03	-04	-05	-06	-07	-08	-09	-0A	-0B	-0C	-0D	-0E	-0F	-10
,	H	E	L	L	O	,	1	2	3	.	4	5	,	7	.	1	null

From which AA$, BB and C% are assigned.

[2] Disk files may sometimes have data stored with a leading linefeed character; this is typical of pre-BASIC 4 files written without the precautionary PRINT#N,X$;CHR$(13); but with PRINT#N,X$: which sends Carriage return with the line feed. This is not a great problem; if records sometimes print a line below their expected place, put in a test-with-correction like this:

      1005 IF ASC(X$)=10 THEN X$=MID$(X$,2)

Line-feed is ASCII 10; when found, X$ is stripped of its initial.

Abbreviated entry: iN   **Token: $84 (132)**

Operation: This is similar to INPUT, except that the input device as specified by logical file number is first set, then unset, on either side of INPUT. The actual INPUT is complex: a flag, $0B^2 holds 0 to signify INPUT (#$98 means READ, #$40 GET); another flag, $03^2 holds the quotes mode on-off byte; two more flags, $07 and $08^2 are used in type matching, holding respectively #$FF or #0 for string/numeral, and #$80 or #0 for integer/real. With BIT and branch, the routines are elaborately negotiated.

ROM entry points:

      BASIC 1: $CAC6 (51910)
      BASIC 2: $CAA7 (51879)
      BASIC 4: $BBA4 (48036)

---

*BASIC 1's buffer extends from $0A-$59. $5A is used for working storage. $09 holds the initial comma. So J=9 TO 89 is the correct PEEK loop with BASIC 1.

^{2} In BASIC 1, these are, in order, $62,$5A,$5E, and $5F.

INSTRING$

BASIC string function unavailable directly in CBM BASIC

PURPOSE: This version of INSTRING$ inserts one string within another, without, however, overrunning the end of the reipient string. It is modified from a routine by W Maclean, quoted by Jim Butterfield in CPUCN v2#8, who says that this type of routine is useful for 'manipulating data records in disk files and setting up formatted printer or screen outputs'. The routine is relocatable; location 0 holds length, ($01) pointer, to the string, so USR (if any) will have to be repoked. The demonstration program sets up a few adjacent strings in memory; this is a check to ensure that overlap from INSTRING$ doesn't corrupt the next string.

Machine code: This version is BASIC 2; see appendices for other ROMs.

1	826	20 F8 CD	$033A	JSR $CDF8	;	Check for comma. Error message if absent.
2	829	20 9F CC	$033D	JSR $CC9F	;	Input expression. Error message if absent.
3	832	20 90 CC	$0340	JSR $CC90	;	Check it's a string. Error message if not.
4	835	A0 02	$0343	LDY #$02	;	Loop inputs LEN of string into location $00,
5	837	B1 44	$0345	LDA ($44),Y	;	and stores pointer to the start of string
6	839	99 00 00	$0347	STA $0000,Y	;	into indirect locallon ($01).
7	842	88	$034A	DEY		
8	843	10 F8	$034B	BPL $0345		
9	845	20 F8 CD	$034D	JSR $CDF8	;	Check for comma. Error message if absent.
10	848	20 9F CC	$0350	JSR $CC9F	;	Input expression. Error message if absent.
11	851	20 90 CC	$0353	JSR $CC90	;	Check it's a string. Error message if not.
12	854	A0 02	$0356	LDY #$02	;	Loop inputs LEN of second string into loca-
13	856	B1 44	$0358	LDA ($44),Y	;	tion $88, and stores pointer to start of
14	858	99 88 00	$035A	STA $0088,Y	;	second string in indirect location ($89).
15	861	88	$035D	DEY	;	[This is in the RND work area].
16	862	10 F8	$035E	BPL $0358		
17	864	20 F8 CD	$0360	JSR $CDF8	;	Check for comma. Error message if absent.
18	867	20 9F CC	$0363	JSR $CC9F	;	Input expression. Error message if absent.
19	870	20 8E CC	$0366	JSR $CC8E	;	Check it's numeric. Error message if not.
20	873	20 D2 D6	$0369	JSR $D6D2	;	Convert Fl Pt Acc#1 into integer.
21	876	18	$036C	CLC	;	Check that the numeric value does not exceed
22	877	A5 11	$036D	LDA $11	;	the end of the stting into which it is to
23	879	C5 88	$036F	CMP $88	;	be inserted. Exit if it is.
24	881	B0 1B	$0371	BCS $038E		
25	883	65 89	$0373	ADC $89	;	Increment the second string pointer by the
26	885	85 89	$0375	STA $89	;	low byte of the numeric value, so the
27	887	90 02	$0377	BCC $037B	;	pointer points inside the string.
28	889	E6 8A	$0379	INC $8A		
29	891	A0 00	$037B	LDY #$00	;	Load a byte from the first string ...
30	893	B1 01	$037D	LDA ($01),Y	;	...
31	895	91 89	$037F	STA ($89),Y	;	... & put it into the second.
32	897	C8	$0381	INY	;	Increment the position counter Y.
33	898	98	$0382	TYA	;	Now check that the new value of Y doesn't
34	899	18	$0383	CLC	;	point outside the second string; if it
35	900	65 11	$0384	ADC $11	;	does, some other string will be corrupted.
36	902	C5 88	$0386	CMP $88		
37	904	B0 04	$0388	BCS $038E		
38	906	C4 00	$038A	CPY $00	;	Check whether we've now moved every byte
39	908	D0 EF	$038C	BNE $037D	;	of the first string – if so, exit.
40	910	60	$038E	RTS		

BASIC demonstration:

```
0 INPUT "NUMBER (N)"; N
1 A$="AAAAAA": X$="123": B$="BBBBBB": Y$="ABCDE": C$="CC": D$="DDDDDDDDDD"
2 A$=A$+"": X$=X$+"": B$=B$+"": Y$=Y$+"": C$=C$+"": D$=D$+"": REM IN HIMEM
3 SYS 826,X$,Y$,N: PRINT A$" "X$" "B$" "Y$" "C$" "D$: REM DISPLAY VALUES
4 GO TO 0
```

(NB: line 2 by calculating the strings, leaves them unchanged, but in high RAM. If this isn't done – try deleting line 2 to see the effect – the actual strings in line 1 are changed, so the loop in line 4 won't work as might be expected)

The program run as it stands gives:

NUMBER (N)? 0

AAAAAA 123 BBBBBB 123DE CC DDDDDDDDDD

NUMBER (N)? 1

AAAAAA 123 BBBBBB A123E CC DDDDDDDDDD   etc.

INT

BASIC arithmetic function

PURPOSE: Converts the argument into the nearest integer which is less than (or equal to) the argument.

Syntax: INT(arithmetic expression). The argument must be a valid arithmetic expression; the limit is *not* the range for integers, but *is* the range for floating point numbers, approximately ± 1.7 E 38. For this reason the statement L=INT(1234567.8) is valid. However, L%=INT(1234567.8) generates an error, since the result is too large for an integer variable.

Modes: Direct and program modes are both valid.

Examples: 100 PRINT INT(X+.5) : REM ROUNDS TO NEAREST WHOLE NUMBER (+ve <u>and</u> -ve)
            PRINT INT (1234567.8)   : REM 1234567
            PRINT INT (-123.4)   : REM -124
            1000 PR= INT(.01+ P*(1+MU/100)): REM PRICE (PENNIES) AND MARKUP
            50 IF D<> INT(D) GOTO 40 : REM GO BACK FOR RE-INPUT

Most rounding routines in BASIC use INT. The first and fourth examples illustrate simple rounding; for commercial use such routines must be more elaborate, so that 1 appears as 1.00 and so on. The principle on which the first example relies is that the entire range from X.000 to X.999 is converted to X by INT. Obviously to round to the nearest number, and not just drop the decimal portion , .5 must be added, shifting the range up to X.500 to X+1.499, so the lower half are rounded down by INT, but the upper half of the range are rounded up. Line 1000 is a similar example, where P is a price and MU a markup percentage. The value PR is rounded down. However, there is a small item (.01) also included. This is often useful with INT, because this function occasionally will round down a value when this appears unnecessary. In the illustration, P may be 1000 and MU 25; if the outcome of the calculation is held in floating point as 1249.9999, PR takes the 'wrong' value of 1249, and this may be noticeable.

The second and third examples are straightforward examples; the fifth is a simple test for integer input.

Notes: [1] Integer expressions in brackets may need to be kept there. Zeller's congruence for finding the weekday uses INT(Y/4) + INT(C/4) + OTHERS which is easily 'simplified' into the incorrect INT(Y/4 + C/4) + OTHERS.

[2] 'INT' is the same function as 'ENTIER' in ALGOL. 'FIX' is an alternative which rounds negative numbers <u>up</u>. This is equivalent to SGN(X)*INT(ABS(X)), which separates out the sign.

Abbreviated entry: None

Token: $B5 (181)

Operation: The argumented is evaluated and validated and put into floating-point accumulator #1. The function's objective is to leave the accumulator with the rounded down equivalent of the same number, again in floating-point form. It accomplishes this by converting the entire number into its 4-byte integer equivalent, then converting this back into floating-point format. There is also a test on entry of the exponent; if this exceeds or equals 160 no conversion is carried out. The number ($>= 2^{51}$) is too large to have any decimals.

ROM entry points:

       BASIC 1: $DB9E (56222)
       BASIC 2: $DBD8 (56280)
       BASIC 4: $CE02 (52738)

LEFT$

BASIC string function

PURPOSE: **Extracts a substring from a string, consisting of the leftmost characters from the string.** This function, in association with MID$ and RIGHT$ and the string concatenation operator +, is used in text and string processing in BASIC.

Syntax: LEFT$(string expression, arithmetic expression). The string expression must be valid, i.e. made up from string functions and/or literals and/or string variables only. Its length cannot exceed 255. The maximum value of the arithmetic expression is 255. Its minimum depends on the ROM: BASIC<4 will not accept a value of zero, corresponding to a null character, but BASIC 4 will.

Modes: Direct and program modes are both valid.

Examples:
```
PRINT LEFT$("HELLO"+"THERE!",3) : REM RESULT IS HEL
PRINT LEFT$("HELLO"+"THERE!",50): REM RESULT IS HELLO THERE!
10 PRINT LEFT$(X$+"          ",10);:  REM A FORM OF TAB(
3010 PRINT LEFT$(STR$(L)+"          ",10);: REM ANOTHER TAB( ...
3010 PRINT L; LEFT$(SP$,10-LEN(STR$(L)));:   REM ... AND ANOTHER!
```

LEFT$ is closely related to RIGHT$. Further examples of string manipulations are given there. These five lines of code demonstrate some rather basic points. The first two direct mode statements show how the function works; its parameter is simply applied to the string, which may be any expression, and measures off a length from it. Rather than print an error message if the original string is not long enough (see the second example) the length parameter is not allowed to exceed the length of the string.

The final three program lines demonstrate methods of formatting strings for output to a printer; this can be a problem when TAB(doesn't work. Line 10 shows how X$, a string assumed shorter than 10, can be printed and also leave the output pointer waiting at a constant position in spite of differences in individual X$s. The first line 3010 uses exactly the same construction, but applied to a number. The alternative line 3010 achieves the same effect, with SP$ defined to be a string of spaces, but it is a less elegant construction.

Notes: [1] This diagram should make the operation of this function clear:

X$="	O	R	I	G	I	N	A	L	sp	S	T	R	I	N	G	"
String position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

PRINT LEFT$(X$,6) prints ORIGIN. These are the six leftmost characters.
PRINT LEFT$(X$,3)+"GAMI" prints ORIGAMI.

[2] LEFT$(X$,N) can be replaced by MID$(X$,1,N). With BASIC 4 ROMs this has no effect, but earlier BASICs reject LEFT$(X$,N) when N is zero. The MID$ version is preferable therefore when older ROMs are used and when a null character may be legitimately returned.

Abbreviated entry: leF (includes $)   **Token:** $C8 (200)

Operation: The pointer to the string, and its parameter, say N in LEFT$(X$,N), are recovered from the stack, where they are put by normal string function processing. The length of the string is found - the pointer points at it - and compared with N; the smaller of the two is taken.   #0 is pushed onto the stack, followed by the smaller parameter (in the process, X, Y, and A are swapped around in a byte-saving but confusing way). Finally, the routine which LEFT$, RIGHT$ and MID$ all share is dropped into, and the new string is set up for printing or assignment.

ROM entry points:

```
BASIC 1: $D5D8 (54744)
BASIC 2: $D5DA (54746)
BASIC 4: $C836 (51254)
```

LEN

BASIC arithmetic function

PURPOSE: Determines the length of a string or string expression.

Syntax: LEN(string expression). This is an arithmetic function of a string argument. The string expression must be valid; it can consist only of literals, string variables, and string functions concatenated by +. Its maximum permissible length is 255 characters. If spaces are included when using BASIC>1, for instance LE N("D"), an array LE() will be assumed, and a ?TYPE MISMATCH ERROR generated whenever the code is encountered.

Modes: Direct and program modes are both valid.

Examples:
```
        PRINT LEN("HELLO")                      : REM RESULT IS 5
        X$=" SAILOR": PRINT LEN ("HELLO"+X$)+3 : REM RESULT IS 15

        330 FOR J=1 TO 10
        340  PRINT SPC(19 - LEN(MSG$(J))/2); MSG$(J)
        350 NEXT

        IF LEN(IN$)<>L THEN PRINT "*** MUST BE" L "DIGITS"

        250 X$=" *&#" : REM LIST OF SPECIAL CODES TO BE CHECKED ...
        260 FOR J=1 TO LEN(X$) : REM NOTE THAT THE LIST IN LINE 250 CAN BE
        270  IF G$=MID$(X$,J,1) THEN RETURN: REM CHANGED; THIS ROUTINE WILL
        280 NEXT: PRINT "NOT RECOGNISED": REM STILL WORK CORRECTLY.
```

The first two examples in our illustrative batch are straightforward direct mode arithmetic calculations. The first simply measures the number of characters in the string; the answer is obviously five. The second is more complex and shows how LEN, being an arithmetic function, can be used as part of an arithmetic expression. Again the answer is obvious – the combined string "HELLO SAILOR" is 12 characters long; 12+3 is 15.

The short routine in lines 330 – 350 is a formatting routine, which prints the ten strings held as MS$(1) to MS$(9) one after the other, *centred* on the screen (change the parameter to 39 for an 80-column screen). It does this by printing sufficient spaces to print half the string before the midpoint of the screenline. The other half of the line is therefore printed symmetrically.

The next line of code is a simplified fragment of an input validating routine, which tests the length of an input string against its correct value.

Finally, lines 250 & 260 show between them how LEN can assist in soft-coding and make a program more easily modifiable. Had the loop variable in line 260 been 4, program maintenance would have been a little harder.

Notes: [1] LEN cannot return a value outside the range 0-255 (see diagram). The length isn't actually measured; only the parameter is taken, and anomalies can result from this, e.g. when CHR$(0)s are concatenated onto a string, or the parameters are altered by direct poking.

Abbreviated entry: None   Token: $C3 (195)

Operation: The ROM has only one subroutine followed by a jump. The subroutine (which is shared by ASC and VAL) sets pointers to the string and also loads its length into both A and Y. This part has been slightly rewritten in BASIC 4. The mode flag is changed from string to numeric; this is necessary to avoid ?TYPE MISMATCH ERRORs. Now a fixed-to-floating point conversion routine is jumped to; this one is in POS, which puts zero into A, and in effect converts the length in Y only into floating-point.

NAME	NAME	LENGTH	POINTER	0	0

ROM entry points:

```
    BASIC 1: $D654 (54868)
    BASIC 2: $D656 (54870)
    BASIC 4: $C8B2 (51378)
```

LET
BASIC command

PURPOSE: Assigns a value or a string to a variable. The variable's name causes an integer, real number, or string to be allowed by the assignment.
If the types don't match - if a variable with a string name is assigned a number or a numeric variable assigned a string - then ?TYPE MISMATCH ERROR is printed. Interconversions between integers and reals are allowed, subject to the condition that integers be within the range -32768 to +32767.

Syntax: LET is never needed with CBM BASIC. If the first item in any statement is not a token, LET is assumed by default; the parsing process looks for a name, the '=' sign, and a matching arithmetic or string expression. With square brackets representing the optional command, the syntax is:-
[LET] Real variable name = arithmetic or integer expression, or
[LET] Integer variable name = arithmetic or integer expression, or
[LET] String variable name = string expression.
Variables can be either simple variables (e.g. X,C%,A1$) or subscripted variables like A(7),JK%(100),M$(Z). If the variable doesn't yet exist, it is set up in one of the two RAM areas used for the purpose. Subscripted variables are put into the second of these areas, with dimension(s) set to the default value of 10, if a prior DIM statement hasn't been used. An integer variable is assigned the rounded-down value of the arithmetic expression on the right of '=', but if the value is too extreme ?ILLEGAL QUANTITY ERROR results.

Modes: Direct and program modes are both valid.

Examples: LET B=45056: LET RQ=.005:   REM SAME AS B=45056:RQ=.005
           LET Q%=Q/256: LET A1%=12.3: LET B%=10000: REM SAME AS Q%=Q/256 ETC.
           LET S$="BCFGHPQSU": LET DO$="WRITE":REM OR S$="BCFGHPQSU":DO$="WRITE"

           100 FOR J=1 TO 50: READ X$: LET Y$(J)=X$: NEXT

           142 IF JD>LEN(JD$) THEN LET JD=0

The three direct-mode examples show real, integer, and string assignments. Note that the expressions assigned to integer variables need not themselves be integral, but will be rounded down. A1% takes the value 12, and Q% is set equal to the higher byte of Q, assuming Q is in the range 0-65535. The fourth line is a composite LET statement which assigns fifty subscripted variables with strings read from data statements. As with all the other examples, LET may be omitted. Finally, we have a conditional assignment (taken from a decimal point processing routine). Note [5] enlarges upon this topic.

Notes: [1] Some BASICs require LET in their assignment statements.

[2] The assignment routine can be called in machine-code, and used to set up special user-defined variables. See VARPTR for an explicit example.

[3] Variables can be assigned and re-assigned with complete freedom. This can cause problems: a variable may be changed or reused without its previous use being remembered. This is particularly a hazard with subroutines, and is the reason that tables of variables ought to be kept with large programs. There are computer languages which possess both 'local' and 'global' variables: FOR-TRAN and PASCAL do; COBOL doesn't. As an illustration of the type of trap which may occur, consider this subroutine, which prints the value of L as a hexadecimal number, so that L=52000:GOSUB 600 prints $CB20:

600 L=L/4096:FORJ=1TO4:L%=L:PRINTCHR$(48+L%-(L%>9)*7);:L=16*(L-L%):NEXT:
     RETURN

The subroutine uses, in addition to L, variables J and L%. The values of each of these are changed by the subroutine. Suppose a table of hexadecimal values is wanted corresponding to 52000 - 52020. This loop: FOR K=52000 TO 52020:L=K: GOSUB 600: NEXT will work correctly. This one: FOR L=52000 TO 52020: GOSUB600: NEXT will not, since L is changed by the subroutine.

[4] <u>String Assignments</u>. String variables hold their strings in two distinct ways and this peculiarity of Microsoft BASIC needs to be borne in mind in several circumstances, the most common being the situation when a program is loaded from within another program, but uses the first program's variables. (The LOAD command of course is specially designed to permit this in CBM's BASIC).

```
10 REM *** DEMONSTRATION PROGRAM TO SHOW VARIABLE SHARING ***
20 A$="HELLO": B$="STRING EXPRESSION"+""
30 LOAD "2ND PROGRAM"
```

```
10 REM 2ND PROGRAM
20 PRINT "A$=" A$
30 PRINT "B$=" B$  :REM ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

These demonstration programs (written for tape – the disk version is similar) show the problem. SAVE program 1 on tape, then SAVE "2ND PROGRAM". Rewind and LOAD the run program 1. This puts two string variables after the program, A$ and B$. But (see diagram) the pointers to A$ point within the area which program 1 occupied; in fact they point to the position in memory where "HELLO" originally started. The second program therefore prints A$ as a string of the correct length but starting somewhere in the REM statement in line 30. The exact position depends on the number of spaces inserted into the programs. Variable B$, on the other hand, appears correctly as "STRING EXPRESSION". This is so because all *evaluated* strings need to be processed, and have to be stored in the next available space in RAM. (Again, see the diagram). Chapter 2 has a longer explanation of this and similar phenomena.
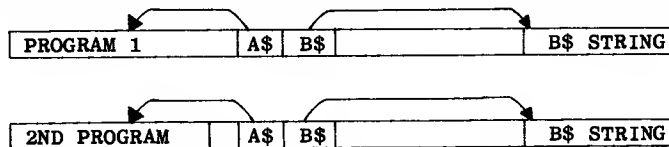
[5] When LET is not used, it becomes easy to forget the distinction between '=' as an assignment, and '=' as a comparison operator. The statement:

       IF X=0 THEN X=12345   or   IF X=0 THEN LET X=12345

uses '=' in both senses; the first use does not, obviously, set X=0. When LET is compulsory, the distinction is retained. The language 'C' uses '==' as its assignment operator. One practical effect of this occurs where dummy variables are set up at the start of a program. This statement: A=B=C=D=E looks as if it will initialise these five variables in the correct order; in fact, the statement is parsed A=(B=C=D=E), and the bracketed expression evaluated. Only A is set up, so the hoped-for speed improvement may not materialise. Rather confusingly this isn't true of arrays. A=B=C=D(5) sets up A and D with the default dimension of 10 if it doesn't exist already.

Abbreviated entry: lE (or nothing)

Token: $88 (136)

Operation: Variables are assigned like this: first, the variable is sought in RAM and set up if it does not yet exist. (This can be a longish operation if arrays have to be moved to accommodate simple variables). Its location is saved. Now, the token for '=' ($B2) is checked; if something else is present, ?SYNTAX ERROR is printed. Two variable-type flags are saved on the stack; these were set by the original search routine. Location 7 holds #0 if the variable was numeric, #FF if it was a string, and 8 holds #80 for an integer, #0 for a real variable. Now the expression is evaluated; a general-purpose routine exists for this purpose. The result is checked for type match with the variable name, giving ?TYPE MISMATCH ERROR when appropriate. Finally, the routine branches to three places, to process integers, reals, and strings respectively. A special check for TI$ is included in the string processing. All the ROMs are similar, but BASIC 4 has extra string processing to handle its complement of pointers.

ROM entry points:

```
BASIC 1: $C89D (51357)
BASIC 2: $C8AD (51373)
BASIC 4: $B930 (47408)
```

LIST

BASIC system command

PURPOSE: Displays part or all of a BASIC program in memory in a readable form.

Syntax: LIST   LIST linenumber   LIST linenumber-linenumber
   LIST - linenumber   and   LIST linenumber -   are all accepted. LIST 0 is interpreted in the same way as LIST. The actual linenumbers need not exist in the program to be specified as parameters.

Modes: Direct and program modes are both valid. In program mode, however, this command will stop the program when the lines have been listed.

Examples: LIST   :REM LISTS ENTIRE PROGRAM
      LIST 1000   :REM LISTS LINE 1000 (IF IT EXISTS) ONLY
      LIST 60000-   :REM LISTS EVERYTHING ON AND AFTER 60000
      LIST 70-200   :REM LISTS ALL LINES FROM 70 TO 200 INCLUSIVE

      555 PRINT "▨▨ INSTRUCTIONS "
      560 PRINT "▨▨RVS▨   PUTS TRACE ON/OFF/ON
      4390 PRINTL9R"DISK TYPE2 = "DT$: PRINT "C2HECK HISTORY"
      57055 print "+@@@@@@@@@@@@@@@@1@@@@@@@@@@@@@@@@1@@@@@@@@.";

      10000 LIST 400: REM LINE 400 HOLDS DATA RELEVANT TO THE PROGRAM

The first four examples illustrate, with comments, various permutations of this command. Often the output will appear on the screen. When this is the case, screen scrolling may be slowed with the RVS key, or, with the 8032's revised keyboard, ←− . BASIC 4 also has a pause feature, activated by either : or * , and cancelled by any of a number of keys.

LIST can be made to print to other peripherals. If it is directed to a printer, typically by OPEN 4,4: CMD 4: LIST a hardcopy will be generated on paper. It can also print to a cassette or disk file; in this case the file contains the program as listed, with PRINT for instance stored in 5 bytes instead of the usual tokenised single byte. The three examples of hardcopy program listings show the output produced by a Commodore printer, which is very similar to the way the screen displays it, although some of the dot patterns are not identical. Non-Commodore printers don't usually have the special characters of CBM's set, and in some cases, for instance daisywheel printers, can't have. The two lines 4390 and 57055 are typical examples of this sort of thing. Most of the listing is intelligible, but strings within quotes may produce anomalies. Line 4390 includes some RVSOFF characters which obviously are there to help with the screen appearance. Line 57055 holds a string of graphics characters which in fact are the top line of a box used in inputting data. Although tiresome, this is not usually much of a problem. Commodore tend to view all this as a reason for buying only CBM printers.

The last example shows LIST in program mode. It acts rather like STOP, except that CONT won't work, but it also lists the lines requested. There is an application of this in the relocatable loaders for LIST (q.v.).

Notes: [1] <u>REM, quotes, and POKEs.</u> Shifted characters after REM, unless enclosed in quotes, are interpreted as tokens, and printed out in their expanded forms. See the notes under REM on this subject. REM is also capable of holding carriage returns, form feeds, screen clears and so forth, and these are sometimes used to improve the hardcopy appearance or provide a rudimentary UNLIST. LIST does not provide a one-to-one conversion of program information into listing. By POKEing, lines can be generated which LIST apparently perfectly but produce ?SYNTAX ERROR on running. An appendix on internal storage of BASIC gives details. Strings can be made to list oddly by inserting unusual characters: For instance, DEL keys can make parts of a listing invisible, on the screen at least: 10 ?"GO AWAY" can be edited by moving the cursor back over the second quote, inserting eight spaces with the insert key, putting in eight deletes (which appear as reverse T) and erasing the final quote. This lists as 10 PRINT.

[2] LIST is upward compatible, but not downward compatible, between CBM
BASICs. BASIC 4 disk commands (CONCAT, DOPEN, etc) don't exist as
keywords, and therefore can't be listed, on earlier ROMs. And BASIC 1
cannot list GO TO (with a space) since it lacks the GO token.

[3] BASIC 1 list has a bug, corrected in later ROMs, which causes a line
of apparent length > 255 to list in an infinite loop. If a link address is
faulty for some reason - perhaps a bad load - there is no way to stop the
loop apart from switching off or using some hardware reset. Lines like:

```
49087 SIN SIN SIN SIN SIN SIN SIN SIN SIN SIN SIN SIN ....
43690 ++++++++++++++++++++++++++++++++++++++++++++++++++ ....
```

manifest an analogous bug; perhaps the end of program bytes have been
changed, so LIST continues past the end. The line of SINs is an attempt
to interpret a collection of $BFs in memory. ($BF=191, and 191 + 191*256=
49087). The line of plusses is a similar effect, but this time is caused by
$AA in memory, probably left from switchon. ($AA=170, and 170 + 170*256=
43690).

[4] Curiosity seekers might like to note the following:
    i. 0 LIST is the shortest self-replicating program (unless, as once
suggested in a letter to'Byte', the 'null program' is permitted).
    ii. The longest listable valid line is a five digit linenumber followed
by 251 RESTOREs or CATALOGs, depending on the ROM version.

[5] LIST happens to be a relatively compact command in ROM, and is quite
easy to move into RAM and modify. The TRACE routine printed elsewhere
and relocating loaders for user-defined LIST show this. Other modifications
include list routines which scroll down the screen (e.g. in 'Disk-o-Pro'),
lower-case listings for CBM printers which print a cursor down after each
new line, and routines to convert single characters into more readable
forms. Cursor control characters, pi, and tokens corresponding to those
of upgraded ROMs are likely to be useful, so that [HOME], [RVS], [PI]
and DOPEN replace nonsense characters, blanks, ?SYNTAX ERROR. Yet
another possibility is the substitution of graphics characters by their
keyboard equivalent; programs using graphics are difficult to enter from
hardcopy by the keyboard, because they are printed in a run-together
form which is painful to read. The only other programs on these
lines that I'm aware of are by Gregory Yob; see e.g. Printout, April '81,
for a routine, with comments. The article is a reprint from 'Creative Comp-
uting'.

Abbreviated entry: lI

Token: $9B (155)

Operation: This routine uses many zero page locations; this is one reason why
a program can't CONT if LIST is used from within it. Another is that the
return address to BASIC is pulled from the stack. The first thing to happen
is the validation: ASCII numbers, -, or end-of-statement / end-of-line are
permitted. If the syntax was correct, the BASIC addresses are pulled by
PLA/ PLA and the linenumber limits set, defaults being $0000 and $FFFF at
the ends of the range. Now there is the start of a loop to print a new
line: it tests the STOP key, prints carriage return-line feed, checks the
current line against the upper limit, and (if it's still in the loop!) prints
the linenumber. Now a second loop starts: this one processes individual
characters . If it has a quote character ($22), it reverses its quotes
flag. If it finds a zero, it uses the link address to loop to the next line
or to exit, when the link is 0. It prints the character, unless it is a
token, and the quotes flag is off, and it isn't pi; in this case, yet another
loop is entered and the Nth token is turned into the Nth reserved word by
looping until N-1 high bytes of the reserved words table have passed.

ROM entry points:

```
BASIC 1: $C5A8 (50600)
BASIC 2: $C5B5 (50613)
BASIC 4: $B630 (46640)
```

```
63499 REM *** LISTING ROUTINE INCLUDING CURSOR CONTROL CHARACTERS ***
63500 A=1025: B=256: GOSUB 63600: INPUT "LIST FROM,TO";F,T: INPUT "TITLE";T$
63501 INPUT "LINES PER PAGE";LP: INPUT "MAX.CHRS.PER PRINTED LINE";CM: OPEN4,4:
      CMD4,;
63502 PP=PP+1: IF PP>LP THEN PP=1: PRINT CHR$(12): REM FORM FEED
63503 L=PEEK(A+2)+B*PEEK(A+3): X=PEEK(A)+B*PEEK(A+1): Q=0: IF X=0 OR L>T THEN
      PRINT#4,;: CLOSE4: END
63504 IF L<F THEN A=X: GOTO 63503: REM LOOP FINDS LOWER LINE NUMBER, L
63505 IF PP=1 THEN N=N+1: PRINT T$ "              PAGE" N: REM TITLE & PAGE
63506 PRINT RIGHT$("    "+STR$(L),5)" ";: CC=6: REM CHARACTER COUNT=6 SO FAR
63507 FOR K=A+4 TO A+93: P=PEEK(K)           : REM LOOP TO PROCESS CHARACTERS
63508 IF CC>CM-7 THEN PRINT: PRINT "      ";:  REM CHARACTER COUNT=6 AGAIN
63509 IF P=0 THEN PRINT: A=X: GOTO 63502:     REM END OF LINE ENCOUNTERED
63510 IF P=34 THEN Q=NOT Q:                   REM REVERSE QUOTE FLAG
63520 IF Q THEN GOSUB 63700: NEXT:            REM INSIDE QUOTES
63530 IF NOT Q AND P>127 THEN PRINT T$(P-127);: CC=CC+CC%(P-127): NEXT
63540 PRINT CHR$(P);: CC=CC+1: NEXT:          REM PRINT ORDINARY CHARACTER
63600 DATA ***,END,FOR,NEXT,DATA,INPUT#,INPUT,DIM,READ,LET,GOTO,RUN,IF,RESTORE
63601 DATA GOSUB,RETURN,REM,STOP,ON,WAIT,LOAD,SAVE,VERIFY,DEF,POKE,PRINT#,PRINT
63602 DATA CONT,LIST,CLR,CMD,SYS,OPEN,CLOSE,GET,NEW,TAB(,TO,FN,SPC(,THEN,NOT
63603 DATA STEP,+,-,*,/,^,AND,OR,>,=,<,SGN,INT,ABS,USR,FRE,POS,SQR,RND,LOG
63604 DATA EXP,COS,SIN,TAN,ATN,PEEK,LEN,STR$,VAL,ASC,CHR$,LEFT$,RIGHT$,MID$
63605 DATA GO,CONCAT,DOPEN,DCLOSE,RECORD,HEADER,COLLECT,BACKUP,COPY,APPEND
63606 DATA DSAVE,DLOAD,CATALOG,RENAME,SCRATCH,DIRECTORY
63608 FOR K=1 TO 9E9: READ X$: IF X$<>"***" THEN NEXT: REM MAKES RELOCATABLE
63610 DIM T$(128): FOR K=1 TO 91: READ T$(K): NEXT
63620 DATA 3,3,4,4,6,5,3,4,3,4,3,2,7,5,6,3,4,2,4,4,4,6,3,4,6,5,4,4,3,3,3,4
63630 DATA 5,3,3,4,2,2,4,4,3,4,1,1,1,1,1,1,3,2,1,1,1,3,3,3,3,3,3,3,3,3,3,3,3,3,3
63640 DATA 4,3,4,3,3,4,5,6,4,2,6,5,6,6,6,7,6,4,6,5,5,7,6,7,9:REM KEYWORD LENGTHS
63650 DIM CC%(128): FOR K=1 TO 91: READ CC%(K): NEXT: RETURN
63700 IF P=17 THEN PRINT "[DOWN]";: CC=CC+6: RETURN
63702 IF P=18 THEN PRINT "[RVS]";:  CC=CC+5: RETURN
63704 IF P=19 THEN PRINT "[HOME]";: CC=CC+6: RETURN
63706 IF P=29 THEN PRINT "[RIGHT]";:CC=CC+7: RETURN
63708 IF P=145THEN PRINT "[UP]";:   CC=CC+4: RETURN
63710 IF P=146THEN PRINT "[RVOFF]";:CC=CC+7: RETURN
63712 IF P=147THEN PRINT "[CLEAR]";:CC=CC+7: RETURN
63714 IF P=157THEN PRINT "[LEFT]";: CC=CC+6: RETURN
63716 IF P=255THEN PRINT "[PI];:    CC=CC+4: RETURN
63750 RETURN
```

This list routine is written as an appendable subroutine. It searches only for those characters within quotes, although this feature can be rewritten if this is felt important. Any BASIC program can be listed with any ROM using this. The comments make it, I hope, *fairly* self-explanatory.

RUN 63500 will ask for the linenumbers between which to list, a title, the number of lines per page, and the maximum line length on printing. I have assumed the printer will move to the correct postion on receiving form-feed. Some printers don't automatically line feed when the end of line is printed; this is the rationale behind the process of keeping count (with CC) of the characters on the line so far. Machine-code routines run much faster than BASIC. Details of these are presented elsewhere in this text; see section 13.4.2 in Chapter 13.

Variables: A=current link address; X=link address of next line, and if zero denotes the end of the program. L is the current linenumber, K a loop variable, and P the ASCII value of the character being processed, or simply the value, in the case of a token. Q is the quotes flag; with each new line it is reset to 0. CC is the count of characters printed on the current line; CMAX the largest permitted number. When a line overflows to a new line, it is inset by 6 spaces in line 63508.

LOAD

BASIC system command

PURPOSE: Enables a stored memory dump to be reloaded into RAM from external
tape or disk storage. Usually this will be a program in BASIC; it might
alternatively be machine code, a dump of the VDU screen, BASIC with its
stored variables, or any other set of contiguous RAM address contents.
Any of these less common forms of LOAD may require special techniques to
prevent the CBM attempting to process the data as if it were BASIC.

Syntax: Tape has this syntax: LOAD [string exp. [,arith.exp. [,arith.exp.]]].
All the parameters are optional, because there is no ambiguity with tape in
deciding on the next program. The first is the program name, the second
the device number, and the third the secondary address. The secondary
address has no effect whatever on LOAD; it is only possible to include it
because SAVE shares the same validation routine. In all ROMs these para-
meters default to the null string (of length 0) and device #1, so the first
program on cassette #1 will load.
Disk has slightly different syntax: the string expression holding the name
of the routine and its drive number is compulsory.
Note that the string expressions are processed differently: in tape loading,
only the characters specified need match those on tape, so LOAD "HE" loads
HELLO or HEX – whichever is first – but rejects HIGHRES and "".
CBM disks have a more sophisticated matching system in which every char-
acter of the name must be given, *unless* an asterisk is present, in which
case any subsequent characters are permitted, as with the tape system; *or*
one or more question marks appear in the string; these require a character
to be present, but don't care what it is. Thus, LOAD "HE*",8 has the same
effect as the tape command above; it searches both disk drives for a prog-
ram with a name that fits its description. LOAD "0:HE???*",8 will load
HELLO but not HEX.

Note that BASIC 4, and BASIC 2 with certain 'toolkit'-type ROMs, has the
DLOAD command (q.v.) for disk loads. Also, CBM's monitor has a load
command: .L "NAME",01 and .L "0:NAME",08 for tape loading from cassette
#1 and from disk drive 0 respectively. After loading, control returns to the
monitor: these routines are not treated as BASIC but as machine code.

Modes: Direct and program modes are both valid. Their effects, however, are
different. Early CBM manuals include a flowchart which explains how they
differ.
Direct Mode: messages are printed to the screen; when the LOAD is com-
plete the program is ready to RUN, LIST, and so on; it displaces any
previous program. Anyone using a CBM is familiar with this. The sequence
of screen messages appears like this, where square brackets indicate the
optional program name allowed by tape load syntax:

```
Tape: LOAD ["PROGRAM" [,1 or 2]]   Disk: LOAD "0 or 1:PROGRAM",8
      PRESS PLAY ON TAPE #1 or 2          SEARCHING FOR PROGRAM
      OK                                  LOADING PROGRAM
      SEARCHING [FOR PROGRAM]             READY.
      FOUND OTHER PROGRAM ...
      FOUND [PROGRAM]
      LOADING [PROGRAM]
      READY.
```

In each case I've assumed that the named program file does actually exist;
if not, ?FILE NOT FOUND ERROR will appear, or, with tape, the recorder
may continue right to the end of the tape (when no end-of-tape header has
been written to tape).

Program Mode: LOAD within a program-line prints no screen messages, leaving
the screen appearance intact, but loads *and runs* the new program, retaining
the values of the earlier program's variables, subject to some qualifications.
Chaining many short programs is a relic of the old 8K PETs.

Examples:
```
LOAD              :REM LOADS FIRST PROGRAM FOUND ON TAPE #1
LOAD "CALCS"      :REM SEARCHES CASSETTE #1 FOR CALCS, CALCS COPY, ...
LOAD "PROG",2     :REM SEARCHES CASSETTE #2 FOR PROG, PROGA, PROGRAM,...

LOAD "*",8        :REM LOADS FIRST PROGRAM ON DEFAULT DRIVE 0
LOAD "1:*",8      :REM LOADS FIRST PROGRAM FROM DRIVE 1
LOAD "0:ASSEM*",8:REM LOADS ASSEM, ASSEMBLER,... FROM DRIVE 0
LOAD X$,8         :REM X$ INTERPRETED AS STRING WITH PARAMETER & NAME
LOAD "HEL?*",8    :REM LOADS HELLO, HELP, OR WHICHEVER IS FIRST

10000 PRINT "PLEASE WAIT...": LOAD "NEXT": REM PROGRAM LOADS "NEXT"
1235 LOAD "0:ANALYSE",8:REM "ANALYSE" IS NOW LOADED AND RUN.
```

The above examples are, I hope, self-explanatory. With any input/output operation, there is a chance of error; ?FILE NOT FOUND and ?DEVICE NOT PRESENT are two 'fatal' errors which will stop BASIC. Other possible load errors include (with disk) ?FILE TYPE MISMATCH and (with tape) ?LOAD ERROR. Checking DS$ (disk error message) and ST respectively will show up errors. In the case of tape, ST is tested for only one bit after a LOAD, so a checksum error may show up in ST, but not be reported by LOAD.

Notes: [1] Loading from BASIC. This is perfectly successful *provided that:*
(i) The newly loaded program is *not longer* than the older one, and
(ii) The new program doesn't use function definitions or non-computed strings from the old program. All of its other variables may be taken over with unchanged values; these need to be redefined. This pair of short programs demonstrates the use of chaining programs with LOAD; this is a tape version:
```
10 REM THIS (LONGER) PROGRAM SETS VALUES, LOADS PRINT PROG.
20 A=1: B%=2:C$="3": D$="4"+"": DEF FN E(X)=5: F(0)=6
30 LOAD "NEXT PROGRAM" : REM 'END' IS AUTOMATIC
```
Save this first, then save, as "NEXT PROGRAM", this:
```
10 PRINT A,B%,C$,D$,FN E(0),F(0)
```
Rewind,LOAD and RUN. The earlier program sets up variables with their values, then loads the second. This is automatically run, without resetting the variables, in effect performing GOTO the earliest linenumber. You will see that all the variables still exist, except C$ and FN E. See Chapter 2 for the reasons: they are in fact fairly straightforward.

[2] OLD at the start of a newly-loaded program will enable it to run correctly, irrespective of the length of the loading program, but variables' values are lost.

[3] A cassette cannot detect if 'Record' is pressed with 'Play'. If it is, the tape will not LOAD, but be erased as long as the machine continues.

[4] Automatic RUN routines on load can be written for both tape and disk; see Chapter 14. LOAD can be relocated into RAM, so that non-standard loaders can be written. One very useful ROM routine is the load routine which is used by both LOAD and the monitor's .L and omits all the resetting of BASIC. In this way, machine-code or a screen dump or whatever can be loaded from within BASIC, leaving BASIC running. Compu/think disks have this available as an option, with syntax $L;Drive,"Name". With CBM equipment the following are the relevant locations:
$D4 holds device number; $D1 holds length of string parameter; if this is non-zero, ($DA) points to its start. (BASIC 1: $F1, $EE, and($F9)). Also the load/verify flag must be set to 0, for load: this is $9D (BASIC 1: $020B). Finally, call the second LOAD routine listed below. **Example: In BASIC 4, POKE 157,0: POKE 209,0: POKE 212,1: SYS 62294 Loads next program on tape #1.**

Abbreviated entry: lO   **Token: $93 (147)**

Operation: The principal load routine has two parts, one for devices 1 and 2 (tapes), the other for IEEE devices. The IEEE routine takes in 2 characters which it presumes to be the start address; subsequent bytes are stored there and at ● subsequent locations. A separate end address is not stored. LOAD itself sets the load flag, checks the parameters, and saves the present pointers before it calls the load routine; afterwards it checks ST then cold or warm starts.

ROM entry points: LOAD is a 'kernel' command; its jump address is $FFD5.
BASIC1: $F346 /$F369   BASIC2: $F3C2 /$F322   BASIC4: $F401 /$F356

LOG

BASIC arithmetic function

PURPOSE: Computes the logarithm to base e of any positive arithmetic expression. It may be positive, zero, or negative. This function is the converse of EXP.

Syntax: LOG(arithmetic expression). Negative or zero arguments will cause an ?ILLEGAL QUANTITY ERROR. There is no upper limit on the argument except that imposed by the floating-point evaluation of the expression.

Modes: Direct and program modes are both valid.

Examples:
```
PRINT LOG(10)                  : REM ABOUT 2.3026
PRINT LOG(2.7182818)           : REM ABOUT 1
PRINT LOG(X)*.434294482        : REM LOG OF X TO BASE 10
PRINT LOG(X)*1.44269504        : REM LOG OF X TO BASE 2
PRINT LOG(EXP(N))              : REM PRINTS N (POSSIBLY WITH ROUND ERROR)
PRINT EXP( LOG(A)+LOG(B) )     : REM PRINTS PRODUCT A*B
DEF FN P(A)=TEN-INT(LOG(INT(ABS(A))+(INT(ABS(A))=0))*LT)
```

The two first examples show straightforward calculations using this function. A logarithm is a transformation that converts multiplicative relations (and division) into additive relations (and subtraction). The logarithm of ratios is constant; the logarithm of 1 is zero, since multiplying or dividing by 1 has no effect on a number. Slide-rules have their sides marked out logarithmically. These facts are illustrated in various ways in the examples. The last-but-one shows the transformation from a multiplication into an addition, and the use of EXP to find the antilogarithm. Generally, this function is used in statistical and scientific work, either analytically, because its algebraic properties are known, or simply to perform calculations in which very large numbers are combined to give a reasonably-sized result; this sort of thing can happen in statistics.

The final example shows a less desirable application of LOG; the function definition is part of a rounding routine, to be used in a business program. The rationale is that (for example) the logarithm to base 10 of numbers from 100-999.99 starts with 2; the logarithm of 1000-9999.99 starts with 3; and so on, suggesting that a decimal point can be positioned after taking the logarithm of a computed value. Unfortunately, this routine itself is subject to a rounding error; it is possible that 999.9999 may emerge as 100.00, a rather large error.

Abbreviated entry: None

Token: $BC (188)

Operation: Negative and zero arguments are tested for, and if found, the routine exits with ?ILLEGAL QUANTITY ERROR. The series evaluation routine in ROM is used; this calculates log x to the base 2. It is a remarkably short series, of 4 terms only. The argument goes through a series of conversions:* it is put into the range .5-.99999, the remaining exponent being saved on the stack. Then 1/SQR(2) is added; the result is divided into SQR(2); the result is subtracted from 1. These transformations turn x into:
$(1.414x - 1)/(1.414x + 1)$, and $\log_2 x$ of this quantity is found. Then the result is subtracted from .5, renormalised and multiplied by log 2 to base e. The routine appears to be based on the standard expression for logarithms, the series $\frac{1}{2}\log(x) = (x-1)/(x+1) + 1/3*(x-1)^3/(x+1)^3 + \ldots$

ROM entry points:

BASIC 1: $D8BF (55487)
BASIC 2: $D8F6 (55542)
BASIC 4: $CB20 (52000)

---

*I believe the following details are correct. However, there may be errors.

LOMEM & HIMEM

BASIC system command unavailable directly in CBM BASIC

PURPOSE: Reserves a part only of the normal BASIC RAM for BASIC and its variables, freeing RAM for other purposes; these include storing machine code and storing graphics pages to be moved into screen RAM.

Versions: Some micro BASICs have this instruction (Apple, ITT; Tandy has CLEAR n). In general no larger machines have this sort of command. So far as I'm aware, no one has written explicit routines to perform this sort of memory allocation with BASIC; usually, ad hoc pokes do the job. There are several possibilities which we can distinguish with the aid of diagrams:

This is the normal memory map of BASIC:

```
$0400                                    Top of RAM (eg $8000 with 32K)
      |-----------------|----|------|-------|---------|
      |     BASIC       |Vars|Arrays|       | Strings |
      |-----------------|----|------|-------|---------|
```
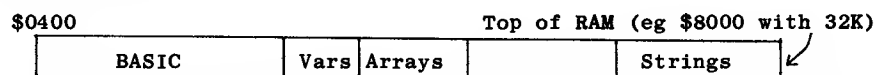
(i) We can lower the pointers to top of memory, creating a spare block of RAM at the high end, where strings would otherwise be formed.

```
$0400                                                      Top
      |-----------------|----|------|-----|--------|--------|
      |     BASIC       |Vars|Arrays|     | Strings| *SPARE*|
      |-----------------|----|------|-----|--------|--------|
```
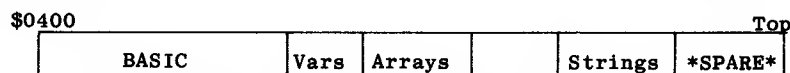
This is easily done, in either direct or program modes. The resulting block will be completely secure from BASIC, unless the locations are poked or corrupted.

     POKE 52,0: POKE 53,48: CLR: REM SETS TOP OF MEM=$3000 FOR BASIC>1
     POKE48,0:POKE49,48:POKE50,0:POKE51,48:POKE52,0:POKE53,48

Both these versions have similar effects. Note that $30=48 in decimal, so $3000 has high byte 48 and low byte 0 when using decimal pokes.

(ii) We can raise the end-of-program pointers, generating space after the BASIC program in memory. This happens automatically when one program loads another from disk or tape. Some BASIC loaders of long machine code routines like Extramon use this method.

```
$0400                                                      Top
      |-----------------|--------|----|------|-----|---------|
      |     BASIC       | *SPARE*|Vars|Arrays|     | Strings |
      |-----------------|--------|----|------|-----|---------|
```
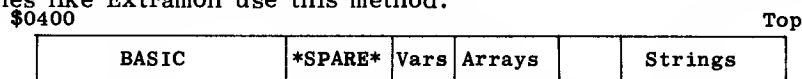
     POKE 43, PEEK(43)+4: CLR: REM RAISES VARIABLES BY 4*256 = 1024 BYTES

This simple routine adds 4 to the high byte of the pointer to the end of program. The program will still stop running when it encounters three null bytes; the unusual positioning of its variables is not relevant to its running.

(iii) We can generate space *before* BASIC by adjusting the start of BASIC pointers. Some proprietary software has a 'Memfix' routine which does this.

```
$0400                                                      Top
      |--------|-----------------|----|------|-----|---------|
      | *SPARE*|     BASIC       |Vars|Arrays|     | Strings |
      |--------|-----------------|----|------|-----|---------|
```
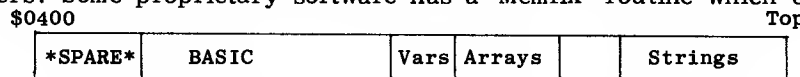
This technique is trickier than the others; it cannot for example be performed from within BASIC. A program modified in this way will SAVE in a non-standard way and LOAD again from its modified starting address.

     POKE 40,LO: POKE 41,HI: POKE 256*HI+LO-1,0: NEW

In direct mode only prepares memory for the keying in of BASIC, where LO and HI (default values 1 and 4) can be user-selected).

(iv) There are other possibilities. Chapter 2 has demonstration programs in which variables (or variables plus their program) are confined to the screen RAM. **POKE 41,128: POKE 53,131: NEW: REM DISPLAYS 768 BYTES BASIC>1**

(v) Corresponding locations for BASIC 1, the oldest ROMs, are listed in the appendix of ROM and RAM addresses.

MERGE
System command unavailable directly in CBM BASIC

PURPOSE: merges two BASIC programs together into a single program.
Unlike APPEND the ranges of linenumbers within the two programs need not be mutually exclusive. In this way, standard subroutines may be inserted into programs without the need for keying-in.

Versions: The usual method involves storing the subroutine(s) on disk or tape as sequential files - not as tokenised programs - then reading them back by a routine similar to that used when adding lines from the keyboard. In this fashion individual lines, one at a time, are merged into the initial program in memory. Keyboard buffer poking keeps the routine working until an invalid piece of BASIC is found. Normally, this is'READY.', written conveniently by LIST at the end of the sequential file. As I say, this is the usual method. It is possible, as with APPEND, to merge entirely in memory, but here we will look at two well-known methods, for tape and for CBM disk respectively.*

[1] Tape Merge.

Use this routine to save the subroutine on tape as a sequential file:

    `OPEN 1,1,1,"NAME OF SUBROUTINE": CMD 1: LIST [LOW - HIGH]`

Where the square brackets denote optional linenumbers,when only a subset of a program is wanted for future merges. The program lists onto tape.

    `PRINT#1: CLOSE 1`

Close the file with these instructions when the cursor returns. This completes the first part of the operation; the named subroutine is stored.

Merging can be carried out now whenever you have a suitable program in memory; the result is a fully merged program, as if the lines had been separately typed at the keyboard.[2] Follow these instructions fairly closely (i.e. get the pokes and cursor movements right!):-

Starting with a program in memory and the tape in cassette #1,

```
BASIC 1:   POKE   3,1:  OPEN 1,1,0,"NAME OF SUBROUTINE"
BASIC 2:   POKE  14,1:              ditto
BASIC 4:   POKE  16,1:              ditto
```

This will read the tape until the correct header has been found; now it will wait for the tape to be read.

```
[CLEAR] and type [DOWN][DOWN][DOWN] then:
BASIC 1:   POKE 611,1: POKE 525,1: POKE 527,13: PRINT"[HOME]"[RETURN]
BASIC 2:   POKE 175,1: POKE 158,1: POKE 623,13: PRINT"[HOME]"[RETURN]
BASIC 4:                           ditto
```

The tape file is now read and merged correctly, subject to the provisos in footnote 2. Eventually, ?SYNTAX ERROR or ?OUT OF DATA ERROR appears depending on whether the program or the merged subroutine had the highest linenumber. This means the merge is finished.

---

*The tape routine is the work of Brad Templeton and Jim Butterfield. Various versions exist of which this is the best. Several disk versions exist; this one is based on Mike Todd's (see IPUG newsletter, May '80). Brad Templeton's 'Power' EPROM uses an analogous technique to construct files like Apple EXEC files, enabling stored commands to control the machine as though from the keyboard. The merging process can be routinised: see e.g. 'PET's Librarian' by D J David, kb-Microcomputing, April '80.

[2] Because the input buffer is 80 characters long, lines with abbreviated input (e.g. ? for PRINT) may not merge correctly if they LIST with overlength lines; if this happens the relevant lines must be separated into shorter lines.

[2] Disk Merge.

```
LDA  #$08
STA  DEVICE NO   ; SET DEVICE to 8 (I.E. DISK)
JSR  IEEE TALK   ; SET IEEE UP FOR TALK
LDA  $63         ; 3 OR'D WITH #$60
STA  SEC ADD     ; SECONDARY ADDRESS 3
JSR  OUTPUT IT   ; OUTPUT TO IEEE
LDX  #$00        ; COUNTS BUFFER CHARACTERS
JSR  INPUT IEEE  ; GET CHARACTER FROM IEEE
CMP  #$0A        ;   LINEFEED?
BEQ  -7          ;   IF SO, IGNORE IT
CMP  #$0D        ;   CARRIAGE RETURN?
BEQ  +10         ;   IF SO, WHOLE LINE INPUT
STA  $0200,X     ; STORE PROGRAM LINE CHR.
INX              ; INCREMENT COUNTER
CPX  #$51        ;   80 CHARACTERS?
BEQ  ERROR       ;   IF SO, LINE IS TOO LONG
BNE  -21         ; INPUT NEXT CHARACTER
STA  KEYBD BUFF  ; STORE CARRIAGE RETURN
JSR  PROC        ; PROCESS LINE IN BUFFER
LDA  #$13        ; [HOME]
JSR  $FFD2       ; OUTPUT IT TO SCREEN
LDA  #$01        ; 1 CHAR IN KEYBOARD BUFFER
STA  NO CHRS     ;
JMP  TOKENISE    ; TOKENISE AND INCORPORATE LINE
JMP  ERROR       ; LINE TOO LONG
```

This rather schematic machine code illustrates the procedure by which disk merging can be made to take place. Characters are read into the buffer, just as though keyboard entry was being used, and the line is processed and tokenised in the same way. After each line, [HOME] is forced so the routine is called again. The concept is similar to the tape merge. The test for lines of length 80+ protects the tables of file numbers, device numbers and secondary addresses if these are in use; if, as is likely, they aren't, a larger number than 81 may be used.

The routine is relocatable, but not transferable between BASICs. The versions below start at $027A (cassette #1 buffer) for compatibility with BASIC 4.

Instructions.

Where F is Logical file number, S is Secondary address, D is drive number,

Save a subroutine with:

        OPEN F,8,S,"D:NAME OF SUBROUTINE,SEQ,WRITE":CMD F: LIST [LOW - HIGH]

Where the linenumbers are optional. When the file is written, close it with

        PRINT#F: CLOSE F

Merge this subroutine with a program in memory by:

        OPEN F,8,3,"D:NAME OF SUBROUTINE,SEQ,READ"   then
        enter [CLEAR]SYS 634 [RETURN]

BASIC 2							
.: 027A	A9	08	85	D4	20	B6 F0	A9
.: 0282	63	85	D3	20	28	F1 A2	00
.: 028A	20	8C	F1	C9	0A	F0 F9	C9
.: 0292	0D	F0	0A	9D	00	02 E8	E0
.: 029A	51	F0	14	D0	EB	8D 6F	02
.: 02A2	20	D5	C9	A9	13	20 D2	FF
.: 02AA	A9	01	85	9E	4C	95 C3	4C
.: 02B2	23	D1					

BASIC 4							
.:027A	A9	08	85	D4	20	D2 F0	A9
.:0282	63	85	D3	20	43	F1 A2	00
.:028A	20	C0	F1	C9	0A	F0 F9	C9
.:0292	0D	F0	0A	9D	00	02 E8	E0
.:029A	51	F0	14	D0	EB	8D 6F	02
.:02A2	20	D2	BA	A9	13	20 D2	FF
.:02AA	A9	01	85	9E	4C	09 B4	4C
.:02B2	73	C3					

MID$

BASIC string function

> PURPOSE: Extracts a substring from a string expression. The substring consists of consecutive characters from the original string expression, and may contain zero characters, all the characters, or (usually) some intermediate number of characters from the string. BASIC 1 will not permit a substring of length zero to be taken.

Syntax: MID$(string expression, arithmetic expression[, optional arithmetic expression]). Neither parameter may take a value greater than 255. If the second parameter is omitted, the substring continues by default to the end of the string expression, like RIGHT$. The first parameter determines the starting point of the substring. See the diagram.

Modes: Direct and program modes are both valid.

Examples: `200 N$=MID$(STR$(N),2) : REM REMOVES LEADING SPACE FROM +VE NUMERAL`

```
620 ni$="EachPackUnitTubeReelSet PairRollMtr "
623 for j=1 to len(ni$) step 4: if js$=mid$(ni$,j,4) then return
626 next: ok=0: em$="in Price Unit": gosub 800: return
```

`1530 MO$=MID$("JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC",3*M-2,3)`

`A$="ABRACADABRA": FOR J=1 TO 6:?SPC(J)MID$(A$,J,12-J): NEXT`

These examples illustrate typical uses to which this substring function may be put. Firstly, line 200 uses the default option in which the second parameter is omitted. This means that the string function, STR$(N), is converted into N$, starting at the second character of STR$(N) and continuing to the end. So if N=23, STR$(N)=" 23" and N$="23".

The program extract (lines 620-626, listed in lower case) is one of a set of input validation subroutines. It checks that the string js$ which has been typed into the machine is one of the four-letter substrings held by ni$. If it is not, an error message routine is called.

Line 1530 converts month number M (1-12) into a 3-letter equivalent.

Lastly, a loop prints symmetrical portions of the string A$.

Notes: [1] This diagram should make the operation of this function clear:

X$="S	A	M	P	L	E	sp	S	T	R	I	N	G"
Position in string: 1	2	3	4	5	6	7	8	9	10	11	12	13

PRINT MID$(X$,3,6) prints MPLE S which starts at 3 and has length 6.
PRINT MID$(X$,5) prints LE STRING which starts at 5 and ends at 13.

[2] The three functions MID$, LEFT$ and RIGHT$ resemble SIN, COS and TAN in that they are closely related. LEFT$ contains the main processing for all three functions. In BASIC, both LEFT$ and RIGHT$ can be put in terms of MID$, although the result is not very readable:

> `LEFT$(X$,N) is the same as MID$(X$,1,N)`
> `RIGHT$(X$,N) is the same as MID$(X$,LEN(X$)-N+1).`

Abbreviated entry: mI (includes $)   Token: $CA (202)

Operation: Sets the default for the second parameter to 255. Then, if there is not a right-hand bracket, checks and inputs the comma and the second parameter (overwriting 255). The string parameters corresponding to the first two parameters (string and starting position) are pulled from the stack. If the string has length zero, ?ILLEGAL QUANTITY ERROR appears. From this data, the true start position and length of the substring are calculated; and LEFT$ is entered to set the new string up. BASIC 2 is logically identical to BASIC 4, but the old ROM differs in several respects, mostly connected with validation.

ROM entry points: BASIC1:$D60F (54799)   BASIC2:$D611 (54801)   BASIC4:$C86D(51309)

MOD

BASIC arithmetic function unavailable directly in CBM BASIC

PURPOSE: Calculates the remainder when an integer is divided by another. Whenever a fixed numerical cycle occurs, a function equivalent to this is likely to be needed; examples include 12-hour clocks, date processing where weekday is represented by 0-6, conversions between number bases, and check digit s and check letters. The word 'mod' is an abbreviation of 'modulo'; this is a mathematical term, used in sentences like '4 = 19 modulo 5'.

Examples:
```
          DEF FN MOD(N) = N - INT(N/D)*D  : REM D=DIVISOR; RESULT IS MODULO D

          D=12: H=FN MOD(16):       REM CONVERTS 16 HOURS TO 4 O'CLOCK
          D=4 : L=FN MOD(Y) :       REM RETURNS 0 FOR LEAP YEAR Y.
          D=7 : WD=FN MOD(2173):    REM RETURNS 3; EG DAY IS WEDNESDAY
          D=256: PRINT FN MOD(50000):  REM LOW BYTE OF 50000 IS 80.

          100 ISBN$="095076500"
          110 T=0:  FOR J=1 TO 9
          120  T = T + VAL(MID$(ISBN$,J,1))*(11-J): REM CALCULATE CHECKTOTAL
          130 NEXT J
          140 D=11: T = FN MOD (T):  REM FIND REMAINDER AFTER DIVN BY 11
          150 T=11-T:               REM SUBTRACT RESULT FROM 11
          160 PRINT T:              REM NOW RESULT IS 1 - 10.
```

The function definition is, I hope, fairly clear: it subtracts the nearest multiple of the divisor from the original number N which leaves a positive answer. It returns 0 if the number is an exact multiple of the divisor. Positive numbers are assumed throughout. Four examples follow, all of which use this function. The fourth must be a familiar one to any programmer using an eight bit microprocessor.

I've included a demonstration program which uses mod to calculate the checkletter of an International Standard Book Number. Checkletters and checkdigits are an interesting aspect of computerology which hardly existed before computers; see Chapter 17 for more on the subject. Briefly, an ISBN has 9 numerals followed by a checkdigit of 0-9 or X. The value of the digit is computed as follows: weights of 10,9,8,...,2 are assigned to the numerals in the ISBN. Each numeral is multiplied by its weight, and the results added. Finally, this number is forced into the range 0-11 by taking the remainder when divided by 11. (Then it is subtracted from 11, an extra, unnecessary step). Try the program with other ISBNs. You will find that the final digit agrees with the printed value of T, or is X when T is 10.

Note: [1] Generally, integers are held exactly by the machine. All the routines on this page produce exact values. If there is a possibility of rounding errors, when using for instance expressions like 3 * .33333333, the evaluation can be foolproofed by adding in a small value:

```
          DEF FN MOD(N) = INT (.1 + N - INT(.1 + N/D)*D) ,
```

where both .1s are necessary to ensure accuracy at every stage.

NEW

BASIC system command

PURPOSE: Appears to erase any BASIC program currently in memory, together
with all its variables, so that a completely new program may be entered
from the keyboard. The effect is similar to turning on the machine anew;
LIST shows nothing. NEW however leaves machine code intact in RAM.

*NB: This instruction is not a formatting command for CBM floppy disks.
See 'HEADER' in the disk commands reference section.*

Syntax: NEW has no parameters; it may be followed by (optional) spaces, but must
be followed by a colon or an end of line zero byte.

Modes: Direct and program modes are both valid.

Examples: NEW
            50000 PRINT "GOODBYE": NEW: THIS WILL NEVER BE REACHED

In either direct or program mode the effect of this command is similar; the
program will no longer list, and the programmer is returned to direct mode;
'READY.' is printed.

Notes: [1] Everything in the cassette buffers, program variables, screen RAM,
stored machine code, and most of BASIC, is untouched. Because the memory
still holds most of what it did before NEW, an inadvertently erased program
can be recovered completely, except for the values of variables: see OLD.

[2] Syntax or out of memory or other errors and anomalous results occur
if the start of BASIC pointers don't point to $0401, or if $0400 does not
contain the normal 0 byte. Example: a machine code routine loaded from
disk or tape sets the start and end pointers as for BASIC; the same point-
ers are shared. NEW does not hardcode the value $0401 into RAM, but
relies on the accuracy of the pointers to BASIC. The solution, apart from
switching off or LOADing a BASIC program, is to set the pointers:

POKE 40,1: POKE 41,4: POKE 1024,0: NEW

Operation: First of all the syntax of NEW is validated; this simply uses a branch
which ensures that NEW is a statement on its own. Now the following
changes are carried out: Zero bytes are put at the link address at the
start of BASIC, in $0401 and $0402. The end of BASIC pointer is replaced
by start of BASIC+2. CHRGET's address is made equal to start of BASIC-1.
These changes are all that are needed to make the translator regard the
program as non-existent. Finally, things are tidied up with CLR:
the variables' pointers are made consistent with a new program; I/O activity
is aborted; the DATA pointer and several flags are set to their default
values. READY is printed, and the machine is prepared for a fresh pro-
gram to be keyed in and run.

	BASIC	Vars	Arrays		Strings	

Abbreviated entry: None

Token: $A2 (162)

ROM entry points:

        BASIC 1: $C551 (50513)
        BASIC 2: $C55B (50523)
        BASIC 4: $B5D2 (46546)

NEXT
BASIC command

PURPOSE: Changes program flow of control to the statement immediately after the matching FOR loop. If no loop variable is specified, the most recently encountered FOR loop is taken. In this way, loops may be automatically processed with relatively less programming effort.

Syntax: NEXT [real variable [,real variable][,real variable] ...]. Square brackets denote optional variables, which must be separated by commas.
?NEXT WITHOUT FOR ERROR is generated whenever the loop variable does not match that currently in the stack, or if there is no active FOR loop on the stack. See the notes for an explanation of this term.

Modes: Direct and program modes are both valid.

Examples: NEXT, like RETURN, operates on the 6502's stack, and can appear anywhere in a program. This type of structure, therefore, is possible:-

```
10 FOR J=1 TO 3: GOTO 100
20 NEXT K
30 NEXT J: END
100 FOR K=1 TO 2: GOTO 20
```

But, even with its processing omitted, it is difficult to read. Straightforward nested loops are therefore normal. Inclusion of the loop variable has a small slowing effect on a loop, but on the other hand makes a loop more readable since the corresponding FOR can be more easily found. Whenever the loop variable exists on the stack, but not at the most recent level, one or more loops will be lost; this is the source of some fairly obvious bugs. For example, this short program executes correctly, but the J loop is aborted repeatedly. Processing of this type has one practical use, which is when a loop is left prematurely, without completion of the entire range of values. As far as BASIC is concerned, the loop is still usable and active. Another NEXT will cause the loop to be re-entered. Given a stacked FOR structure, with free-format NEXTs allowed, this is inevitable. Active FOR loops can cause trouble; this program line

```
10 FOR I=1 TO 10
20 FOR J=100 TO 120
30 IF I<6 THEN NEXT I
40 NEXT J,I
```

```
100 FOR J=1 TO 10000: GET X$: IF X$="" THEN NEXT: REM 50 SECS DELAY
```

delays until a key is pressed, or for about 50 seconds, before continuing with the next line. If this line is within a loop, a keypress causes early exit so that J replaces the other loop's variable as the most recent loop. (Try it- it's hard to describe).   101 FOR J=0 TO 0: NEXT cures this bug.

Notes: [1] How the stack works. For those interested, the following short BASIC program shows what FOR does:-

```
10 FOR PQ=512 TO 480 STEP -1
20 PRINT PEEK(PQ),
30 NEXT
```

The output includes 18 bytes, like this:

Operation: FOR checks the syntax, and assigns its variable, setting it up if need be. The stack is searched, and variable mismatches rejected. If it's a new variable the stack is tested for at least 18 bytes' space. All the parameters are pushed on the stack while checking the syntax of TO and the arithmetic expressions. STEP is assigned 1, then overwritten if a STEP exists. Finally, it drops through to RUN and continues with the next statement. NEXT, if followed by variables, searches for the first, later reading its list. The FOR byte is checked, then STEP is added to the loop variable in acc.#1. The result is compared with the upper limit, and if less (or, with negative step, greater) CHRGET reset.

FOR:	BASIC 1:$C649	NEXT:	BASIC 1:$CC36	STEP:	BASIC 1:$C69C
	BASIC 2:$C658		BASIC 2:$CC20		BASIC 2:$C6AB
	BASIC 4:$B6DE		BASIC 4:$BD19		BASIC 4:$B731

NOT

BASIC unary logical operator

PURPOSE: evaluates the complement of any arithmetic expression (within the valid range). In the case of truth values, this has the effect of converting true to false and vice versa. This second case is by far the most commonly used.

Syntax: NOT must be followed by an arithmetic or logical expression. An arithmetic expression must evaluate to within the range -32768 to 32767. Non integral values will be rounded down.

Modes: Direct and program modes are both valid.

Examples:
```
5 IF PEEK(X)=34 THEN Q= NOT Q: REM SWITCH QUOTES FLAG ON CHR$(34)
PRINT NOT 23456
70 IF NOT OK THEN EM$="- pack type": GOSUB 20000: RETURN
1000 IF D=1 OR 0 AND NOT T$="TAPE" THEN OPEN 15,8,15:PRINT#15,"IO"
```

The first example shows how NOT can be used to switch the values of a flag; in this example, Q true means that the quotes flag is set; when the next quote mark is peeked, it is unset. This has application when writing special LIST routines in BASIC. The second, direct mode example, illustrates the numerical effect of NOT. The value is converted into a 2-byte integer and the bits all reversed: in this example, 23456=$5BA0, so NOT 23456 is computed to be $A45F, which in signed integer terms = -23457. This adding of 1 is general, and is because the bytes are complemented, but not 2's complemented. The third example is a program line which tests the flag OK; if this has been set false, the error message routine prints a warning to the operator. Finally, an example shows NOT in a logical expression.

Notes: [1] A fuller explanation of 2's complement numbers appears under AND.

[2] NOT has a higher position in the hierarchy of logical operators than OR and AND. NOT therefore takes precedence if there would otherwise be ambiguity. NOT A AND B is effectively identical to (NOT A) AND B.

[3] The usual rules of logic apply to NOT, OR, and AND, and may help when attempting to decipher elaborate expressions. Three of these are:

A=NOT (NOT A)
NOT (A AND B) = NOT A OR NOT B
NOT (A OR B) = NOT A AND NOT B.

The second and third of these are sometimes called d'Alembert's rules. These relationships can be demonstrated in many other ways, e.g. Venn diagrams:
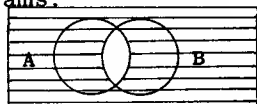
NOT (A AND B)
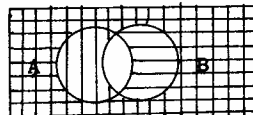
NOT A OR NOT B

Abbreviated entry: nO   Token: $A8 (168)

Operation: The expression following NOT is evaluated, and if valid, converted to a fixed point number in floating-point accumulator #1. The diagram applies to BASIC>1:

$5E	$5F	$60	$61	$62	$63
			A	Y	

Fixed-point here

The contents of $62 are reversed and transferred to the Y register; the contents of $61 are reversed in the accumulator. A standard routine which converts A low and Y high into floating-point form is called finally.

ROM entry points:

BASIC 1: $CDE8 (52712)   BASIC 2: $CDCF (52687)   BASIC 4: $BECC (48844)

OLD

BASIC command unavailable directly in CBM BASIC

PURPOSE: Restores a BASIC program which has been inadvertently erased by NEW. It does this by resetting zero-page pointers to the start of BASIC and the end. This has a further effect: a program LOADed from another program can have its pointers set correctly, so that (for example) a small menu program can safely LOAD a much larger program.

Versions: Several have been published: Practical Computing (Feb.81) has a 6502 routine, which, however, does not set end-of-program pointers. Printout (Jan.'81) had several using Toolkit calls with BASIC; Compute! had an UN-NEW.

My version below is relocatable and may be called from within a program. In direct mode the program will LIST and RUN as usual.

Operation: To decide what OLD is to do, we can start by examining NEW; this
[1] Puts zero bytes at the link address at the start of BASIC, $0401 & $0402.
[2] Changes end-of-BASIC pointer to $0403.
[3] Sets GETCHR address to $0400.
[4] --- Enters CLR routine at this point ---
Current string pointer is set to point to the very top of RAM.
[5] I/O activity is aborted and files closed.
[6] End of variables and end of arrays pointers are set to end of BASIC.
[7] DATA pointer is restored/ some flags are reset/ the stack is reset.

There is an inherent problem in distinguishing simple variables from arrays; this version therefore does not attempt this. The reversible steps are 1,2 and 6. Therefore OLD needs to:-
[1] Replace the link address in bytes $0401 and $0402,
[2] Recover the end of program pointer,
[6] Set the variable and array pointers to the end of BASIC.

[1] is carried out on the assumption that the next zero byte found marks the end of line; [2] assumes that three consecutive zero bytes mark the end of the program. I have also assumed that BASIC starts at $0401, and included this value explicitly; this clears up some problems when machine code has been loaded, so the starting address is assumed, by its pointers, to be somewhere else in RAM.

This is a flowchart of the part of the routine which searches for the end of program:

1
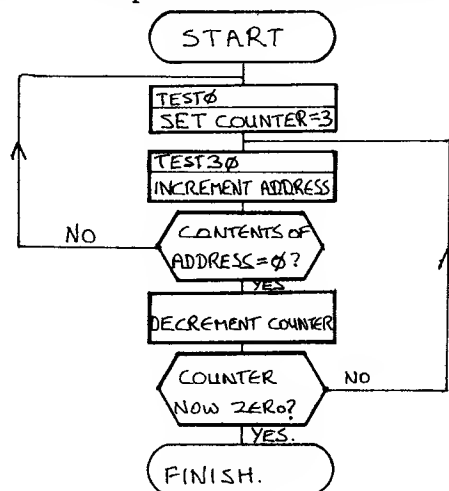
OLD ... BASIC 4.0

634	$027A	A9 01		LDA #$01	
636	$027C	A0 04		LDY #$04	
638	$027E	85 1F		STA POINTRL	;LOAD UTILITY POINTER
640	$0280	84 20		STY POINTRH	; WITH $0401
642	$0282	A0 03		LDY #$03	
644	$0284	C8	FIND0	INY	
645	$0285	B1 1F		LDA (POINTRL),Y	
647	$0287	D0 FB		BNE FIND0	
649	$0289	C8		INY	
650	$028A	98		TYA	;RESTORE LINK ADDRESS
651	$028B	18		CLC	; IN ($0401)
652	$028C	65 1F		ADC POINTRL	; BY FINDING THE NEXT
654	$028E	A0 00		LDY #$00	; ZERO BYTE
656	$0290	91 28		STA (BASICL),Y	
658	$0292	A5 20		LDA POINTRH	
660	$0294	69 00		ADC #$00	
662	$0296	C8		INY	
663	$0297	91 28		STA (BASICL),Y	
665	$0299	88		DEY	; Y NOW HOLDS #$00
666	$029A	A2 03	TEST0	LDX #$03	
668	$029C	E6 1F	TEST30	INC POINTRL	;FIND 3 CONSECUTIVE ZEROS
670	$029E	D0 02		BNE NOINC	; MARKING PROGRAM END
672	$02A0	E6 20		INC POINTRH	
674	$02A2	B1 1F	NOINC	LDA (POINTRL),Y	; (SEE FLOWCHART)
676	$02A4	D0 F4		BNE TEST0	
678	$02A6	CA		DEX	
679	$02A7	D0 F3		BNE TEST30	
681	$02A9	A5 1F		LDA POINTRL	
683	$02AB	69 02		ADC #$02	; ADD #2 TO POINTER TO
685	$02AD	85 2A		STA PROGENL	; 3 ZEROS AND STORE RESULT
687	$02AF	A5 20		LDA POINTRH	; IN END-OF-PROGRAM
689	$02B1	69 00		ADC #$00	
691	$02B3	85 2B		STA PROGENH	
693	$02B5	4C F0 B5		JMP CLEAR	; CLR AND READY.

Notes: [1] BASIC 2 is identical except that the last line must be replaced by:

| 693 | $02B5 | 4C 79 C5 | | JMP $C579 | ; BASIC 2 CLR AND READY. |

[2] SYS 634 calls the routine as written; it is relocatable, however.

[3] The start address of $0401 need not be hard coded in: if your BASIC is written to start elsewhere, use LDA $28/ LDY $29.

[4] The original ROM (BASIC 1) equivalent is this:

OLD - ORIGINAL ROM

826	$033A	A5 7A A4 7B 85 71 84 72
834	$0342	A0 03 C8 B1 71 D0 FB C8
842	$034A	98 18 65 71 A0 00 91 7A
850	$0352	A5 72 69 00 C8 91 7A 88
858	$035A	A2 03 E6 71 D0 02 E6 72
866	$0362	B1 71 D0 F4 CA D0 F3 A5
874	$036A	71 69 02 85 7C A5 72 69
882	$0372	00 85 7D 4C 6A C5

ON

BASIC conditional command

PURPOSE: Branches to one of a list of linenumbers, depending on the value of the variable following ON. ON ... GOTO and ON ... GOSUB are valid. This provides a readable method for programming multiple IF statements of the CASE type, particularly if the variable takes values 1,2,3, ...

Syntax: ON arithmetic expression GOSUB linenumber,linenumber, ...
or ON arithmetic expression GOTO linenumber,linenumber,...
Note that ON ... GO TO is disallowed.
If the expression, on evaluation, is outside the range 0-255, the message ?ILLEGAL QUANTITY ERROR is generated.
If necessary the value is rounded down. When the value=1, the first line in the list is the branch; when 2, the second, and so on.

Modes: Direct and program modes are both valid.

Examples: 1000 ON SGN(X)+2 GOTO 2000,3000,4000: REM FORTRAN CONVERSION
60 ON 1 + 10*RND(1) GOTO 100,200,300,400,500,600,700,800,900,1000

6240 ON X GOSUB 400,410,420,430,440,450,460,470,480,490,500 ...
6250 ON X-20 GOSUB590,600,610,620,630,640,650,660,670,680 ...

200 ON Q GOSUB 100,,200,300

The first example shows a three way branch, depending on the sign of the argument. When X is negative, SGN(X)=-1 so SGN(X)+2=1. So if X is a negative quantity, the first of the three linenumbers obtains. In the same way, if it has zero or positive sign, the second or third linenumber is chosen respectively. The language FORTRAN ('FORmula TRANslation') has this test; sample line 1000 shows the method of conversion to BASIC.

The second example is taken from a game. The random number generating function RND returns numbers in the range 0.000001 to .999999 (roughly!) so the argument evaluates, after rounding, to 1,2,3, ..., 10. Each of the routines has an approximately equal chance of running.

If all the options cannot be fitted on one line, they may overlap onto the next line, as the third example shows. See note [1] for explanation.

ON does not share the peculiarity that GOTO and GOSUB share, of allowing non-numeric characters in linenumbers. However, it does treat null line numbers, as in line 200, as if they were line 0.

Notes: [1] If the variable is 0, or 5,say, when only 4 linenumbers exist, there is no error message; the program merely begins on the next line. This is the reason why lines 6240-6250 in the examples work correctly if X is between 1 and 30 or whatever. The reason for this behaviour is explained below in the section on machine code operation.

Abbreviated entry: None

Token: $91 (145)

Operation: Firstly, the argument following ON is evaluated and validated. If, as it should be, the result is a single-byte numeral, this value is stored (in $62 with BASIC>1). Next, ?SYNTAX ERROR is printed if the following token is neither GOTO nor GOSUB. (This is the reason for ON .. GO TO's unacceptability). The token is stored on the stack: on exit it is pulled back into the accumulator, so the routine knows which of the two commands to execute. Before this, however, the list of linenumbers is processed. This is done in a loop. The first thing is to decrement the stored value of the parameter; if the *result* is zero exit occurs to either GOTO or GOSUB. If the result was not zero, CHRGET gets the next fixed-point linenumber and stores it in ($11) with BASIC>1. Provided a comma follows, the loop continues. So a variable value of zero is treated in effect as 256.

ROM entry points:BASIC1:$C843 (51267)   BASIC2:$C853 (51283)   BASIC4:$B8D6 (47318)

OPEN

BASIC input/output command

PURPOSE: Enters a file's 'logical file number' in a table, together with the device number and secondary address. When BASIC refers to a logical file, for example with PRINT#, the device and its secondary address are taken from the tables and used in processing. *Also,* where necessary, the device is prepared for input or output. Tape files have a header either read or written; disk files' parameters are sent on the IEEE bus to the disk unit.

Syntax: OPEN arith. expr. [,arith.expr. [,arith. expr. [,string expr.]]] . The first parameter is compulsory and must evaluate to 1-255 after rounding down. The second parameter is the device number, which must be 0-15, and is a hardware feature; see the table for CBM equipment's device numbers. The third parameter is the secondary address, which again is a hardware feature, and may not be present on non-CBM equipment. The string parameter is a file name, plus, in the case of CBM disks, other parameters giving drive number and so on. ?SYNTAX ERROR, ?DEVICE NOT PRESENT ERROR, and ?FILE OPEN ERROR return to direct mode, aborting files which are already open.

Modes: Direct and program modes are both valid.

Examples: Note that, while a logical file number is compulsory, the remaining parameters are optional. The device number, secondary address, and string are assigned 1 (i.e. cassette #1), 0, and null string respectively, in all versions of BASIC. All the parameters are evaluatable expressions.

Tape: OPEN 10:REM =OPEN 10,1,0,"" WHICH OPENS FILE #10 TO READ #1'S HEADER
     OPEN 1,1,0,"TAX": OPEN 2,2,1,"TAX UPDATE": REM SYSTEM WITH 2 CASSETTES

Disk: OPEN 15,8,15: REM OPENS ERROR CHANNEL TO CBM DISK AS LOGICAL FILE #15.
     OPEN 1,8,4,"#": REM OPENS A CHANNEL TO A DISK BUFFER FOR B-R, B-W, ETC
     OPEN 2,8,4,"0:ORDINARY FILE,SEQ,READ": REM OPEN CBM FILE FOR READING
     FILE$="1:FILE A": OPEN 3,8,10,FI$+"SEQ,W": REM OPEN FILE FOR WRITING

Other: 10 INPUT "OUTPUT TO DEVICE #"; D: OPEN D,D
     20 PRINT#D, ...:REM PRINT OUT TO SCREEN OR PRINTER ETC, DEPENDING ON
         D - E.G. 3=SCREEN. CONTROL CHARACTERS MUST WORK WITH BOTH DEVICES
     30 CLOSE D

Tape files always start by reading or writing a header. So two files to the same cassette are impossible. Our examples show a header being read into its buffer (where incidentally it may be examined by PEEK) and OPEN statements for a 2-tape system, where data input from #1 may be processed and output to #2. The disk examples (BASIC 4 has DOPEN, which is slightly easier) open files numbered 15,1,2, and 3, all to device 8, the normal disk device number. It is sometimes worthwhile to open files to the keyboard and/or screen.

Notes: [1] CBM Equipment and its Secondary Addressing.

Device:	Device #:	Secondary Address:				
		0	1	2	3-14	15
Keyboard	0					
Cassette #1	1	Read file	Write file+ end-of-file marker on CLOSE	Write file + eof + end of tape marker on CLOSE		
Cassette #2	2					
Screen	3					
Printer	4	---- Varies with type of printer* ----				
Modem	5					
Unassigned	6,7					
Disk Drives	8	Directory Read|Write				Error Channel
Unassigned	9-15					

*Models 4022/3 used 6, and model 4022 10, secondary addresses, for example.

As the table shows, many primary and most secondary addresses are unused so that secondary addresses may be picked at random in most cases. With disks, secondary addresses 3-14 may be used freely. The parameter can't simply be ignored, since the default value of 0 prevents a string being sent on the IEEE and anyway is compulsory before the string. This is why the examples of OPEN involving disks have rather miscellaneous third parameters. The keyboard, screen, and non-CBM printers can be opened as files simply with OPEN 1,0: OPEN 3,3: OPEN 4,4 or with whichever logical file numbers you like. Cassette files are, on the other hand, entirely dependent for their correct operation on the secondary address. The default value of zero is (sensibly) equivalent to reading a file; 1 causes a normal write; and 2 causes an identical write, plus, at the end, a 'header' holding a value which is interpreted as 'end of tape'.

CBM printers are designed with many secondary address features, many of which, unfortunately, don't work correctly or are absurdly complicated to use. Output to these printers is consequently often formatted in ways not compatible with output to the screen or to other printers. The 4022 has the following secondary addresses: 0=print 'as received'; 1=print in format; 2= store format; 3=lines/page; 4=error messages on; 5=define own character; 6=set linefeed space; 7=upper case; 8='lower case'; 9=error messages off; and 10=reset. Some models of printer have 4 and 6 transposed.

[2] RAM Tables. Three tables of ten entries each hold logical file numbers, device numbers, and secondary addresses as they are used by the IEEE bus or cassettes. They start, and may be peeked, at 593, 603, and 613 decimal respectively. (578,588,598 in BASIC 1). The number of entries in the table is stored in another location (174 or $AE; 610 or $0262 in BASIC 1). The overall effect is as shown in the diagram:

593	4	LOGICAL FILE
594	5	NUMBERS
595	15	
596	8	
. . . . . .		
603	4	DEVICE NUMBERS
604	4	
605	8	
606	8	
. . . . . .		
613	255	SECONDARY ADDRESSES
614	97	
615	111	
616	104	

Any new OPEN has its logical file number checked against those already present, and ?FILE EXISTS ERROR reveals duplication. BASIC 1 is inedequately protected against OPENing more than 10 files. Later BASICs print ?TOO MANY FILES ERROR. POKEs into these tables can cause problems, as spurious 'files' may apparently exist.

OPEN 4,4: OPEN 5,4,1: OPEN 15,8,15: OPEN 8,8,8,"0:DATA,S,R" gives the tabled values.

[3] BASIC 4 has DOPEN; it also has a modification of earlier BASICs: PRINT# followed by a logical file number with bit 7 low (i.e. 1-127) does not send a linefeed character with a carriage return. Earlier versions did. This was a result of the fact that PRINT referred to the screen at first; later, the tape processing system was arranged to omit linefeeds, but the IEEE bus still transmitted them to disk. On this subject, see PRINT#.

[4] Other locations. OPEN sets these locations: $D2=logical file; $D4=device; $D3=secondary address; $D1=length of string and, if this is non-zero, ($DA) points to the start of the string.

Abbreviated entry: oP   Token: $9F (159)

Operation: Chapter 13 has a schematic disassembly of this command. The version is BASIC>1; BASIC 1 is written less concisely, but is otherwise logically similar. Firstly, parameters are fetched from BASIC by a single ROM subroutine. This part can be skipped in machine-code programming, by setting the variables directly. ST is set to zero and the file table entries made, if there's room. The IEEE bus, as usual, is processed differently from the screen, keyboard, and cassettes. The string is sent to the IEEE device only if the secondary address is non-zero, and the string is not the null string.

ROM entry points: OPEN is a 'kernel' command. Its address is $FFC0.

OR

BASIC binary logical operator

PURPOSE: Calculates the logical inclusive OR of two expressions which evaluate into the range -32768 to 32767. The result is a 2-byte integer. With logical expressions, the result is true if either of the original conditions were true, or if both conditions together were true.

Syntax: Arithmetic or logical expression OR arithmetic or logical expression. Both expressions must be integers within the signed integer range, or floating point numbers within this range when rounded down. All logical expressions are valid, since they take values of -1 or 0 only.

Modes: Direct and program modes are both valid.

Examples:
```
100 IF A%<1 OR A%>10 THEN ? "OUT OF RANGE"
PRINT -1 OR 12345
PRINT 380 OR 75
6270 OK=OK AND D < 32+(M=4ORM=6ORM=9ORM=11)+(M=2)*(3+(INT(Y/4)*4=Y)
150 IF NL=60 OR TF THEN NL=2: TF=0: GOSUB 5000: REM NEW PAGE & TITLE
```

The first and fourth examples show typical applications, where OR checks the range of a single variable: the first example is, I hope, self-explanatory. The fourth is part of a date validation routine, which checks that the day of the month is acceptable. (NB: a leap-year test is included). It takes advantage of the fact that 'true' evaluates as -1 to calculate the acceptable upper limit of the day number. The second and third examples, on the other hand, do not compute logical functions, but operate directly on the numeral values: -1 ORed with any valid number leaves that number unaltered, because -1 is stored as $FFFF, and this pattern of bits, all 1s, has no effect when ORed with any other bit pattern. PRINT 380 OR 75 gives a bit pattern of %00000001 01111111 = 383; cp. AND. The final example shows OR used with different variables: a new page and two-line title is to be printed if either 60 lines exist on the page so far, or some other condition has set TF to true, for example change of client name.

Notes: [1] *The use of -1 for 'false' is not universal: Apple for example uses +1. Routines which run on a particular machine may need changes of sign if they are to be used with another.*

[2] 'OR' is lowest in the operator hierarchy, and is performed after NOT and AND. Because of this, PRINT 1 AND 2 OR 3 prints result 3, which is also obtained from PRINT (1 AND 2) OR 3. PRINT 1 AND (2 OR 3) is 1.

[3] The truth table for OR is:-

OR	T	F		OR	1	0
T	T	T		1	1	1
F	T	F		0	1	0

Where 1='true' or 'bit set on'
0='false' or 'bit set off'.

The following relationships are fairly easy to demonstrate:
A OR B is equivalent to NOT(NOT A AND NOT B)
A EOR B is equivalent to A OR B AND NOT (A AND B)
And these Venn diagrams show the difference between inclusive and exclusive OR:
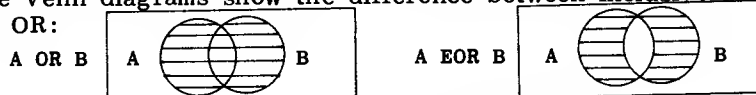
Abbreviated entry: None   **Token:** $B0 (176)

Operation: Identical to AND (q.v.) except for the use of a location holding #$FF which the ROM routine uses to reverse bytes.

ROM entry points:
```
BASIC 1: $CED6 (52950)
BASIC 2: $CEC8 (52936)
BASIC 4: $C086 (49286)
```

PEEK
BASIC arithmetic function

PURPOSE: Computes the decimal value of the contents of any memory location. PEEK, in conjunction with SYS and POKE and, to a lesser extent, USR, allows free access to RAM and ROM. Uses include: examination of ROM, of BASIC, and of variables and pointers; examining hardware locations; examining machine-code; and performing memory moves.

Syntax: PEEK(arithmetic expression). The range is 0-65535.

Modes: Direct and program modes are both valid.

Examples: PRINT "[CLR]" CHR$(34);: FOR J=1024 TO 1100: ?CHR$(PEEK(J));: NEXT
100 FOR J=1024 TO 1100: POKE 31744+J, PEEK(J): NEXT
7675 FOR LS=49TO54: O$(7)=O$(7)+CHR$(PEEK(LT+LS)):NEXT: REM SIZE
2000 IF PEEK(152)=1 THEN PRINT "SHIFT KEY IS PRESSED"

The two first examples, apart from the minor difference of mode, carry out similar functions. Each displays about 1000 bytes of a BASIC program directly on the screen, so that literals, tokens, linenumbers and so on are all made visible. The main difference is that the first example prints the characters, and so may fall foul of Commodore's special characters. The quote mark at the start prevents this, at least until a second byte holding 34 is found. The second example doesn't have this problem, and is a routine to memory-move the program into the screen area. Try them both.

The third example is a line from a program, in which information stored as a file in RAM is now PEEKed out again. O$(7) is the 7th string to be output, has length 6, and is the size description of the item.

The final example shows how knowledge of the system may be used in a program. When the keyboard scanning routine finds the shift key depressed it sets a flag which affects the character printed. The actual figure applies to BASIC 2 and BASIC 4. (BASIC 1 uses 516).

Notes: [1] BASIC 1. This ROM contains a test ensuring that addresses from C000 to E0FF have a PEEK of 0. This protection has been dropped in all later ROMs. BASIC 1 also has a bug, caused by the fact that a pointer it uses is shared by the function processing routine. Line 100 in the examples, and routines generally with several different PEEKs in a statement, don't work in BASIC 1. For both of these reasons PEEK may well be replaced by USR with this ROM.

DATA 165,8,72,165,9,72,32,208,214,160,0,177,8,168,104,133,9,104,
133,8,76,135,210: REM BUG-FREE PEEK FOR BASIC 1 WITH USR.

The 23 bytes above give a peek routine for BASIC 1   JSR D6D0
which is bug-free, so that PRINT USR(50000) prints   LDX #00
208, and POKE C, USR(D) transfers the contents of   LDA (08,X)
D to C. The 12 byte version to the right removes   TAY
the C000-E0FF protection, but doesn't correct the bug   TXA
concerned with function processing.   JMP D278

[2] A double-byte peek or DEEK is often convenient and can be written as a function definition: DEF FN DEEK(X) = PEEK(X) + 256*PEEK(X+1)

Abbreviated entry: pE   Token: $C2 (194)

Operation: BASIC>1 saves the contents of ($11) on the stack. (The omission of this step from BASIC 1 causes its bug). The routine to validate and convert a floating point number from 0-65535 is called; this also stores the 2 byte address in ($11), or ($08) with BASIC 1. It is a straightforward matter to load the accumulator from the address pointed to, restore the original contents of ($11), and jump to the ROM routine which loads the accumulator with #0 and converts Y to floating-point. BASIC 2 has 8 NOPs left from BASIC 1's protection routine, which are dropped with BASIC 4.

ROM entry points:BASIC1:$D6E6 (55014) BASIC2:$D6E8 (55016) BASIC4:$C943(51523)

POKE
BASIC command

PURPOSE: Each POKE replaces one RAM location with the byte value specified by the second parameter. With PEEK and SYS and to a lesser extent USR, POKE enables RAM to be freely accessed from BASIC. It is useful when entering machine code from BASIC, modifying pointers, programs, variables and files in RAM, and putting characters directly onto the screen.

Syntax:POKE arithmetic expression, arithmetic expression. The two parameters refer to the location and the byte. Their values must be within the ranges 0-65535 and 0-255. A POKE into ROM or into an area not occupied by RAM or ROM does not print an error message, and has no effect.

Examples: i.   FOR J=0 TO 255: POKE 8*256*16+J,J: NEXT
           ii. 10 DATA 162,0,138,157,0,128,232,208,249,96
                20 FOR J=826 TO 835: READ X: POKE J,X: NEXT
           iii. 10 REM ************************
                20 INPUT Y: FOR X=0 TO 9: POKE 1032+X, X+Y: NEXT
           iv. FOR J=2000 TO 9E9: POKE J,170: IF PEEK(J)=170 THEN NEXT

The four examples don't cover specific aspects of CBM BASIC operation, of which there are innumerable possible variations. See for example the notes in this section on HTAB/VTAB for zero page pokes, on DEL for pokes which control the keyboard buffer, and VARPTR for hunting variables in order to modify them by POKE.

Example i is a simple loop which pokes to the screen. Since this starts at $8000, the values 0-255 are taken and poked into the screen starting at its top left corner. (The calculation computes $8000 in decimal each time round which is slow but easy). Example ii illustrates how machine code routines may be poked into memory. The loop reads data one item at a time and pokes it into consecutive locations. SYS 826, executed after this short program has been run, produces in machine code the same effect that the BASIC routine achieved. The speed increase is considerable.
Example iii is a self-modifying BASIC program in which 10 consecutive bytes are POKEd into a REM statement. It provides an easy way to discover which tokens correspond to which values in BASIC.
The last example is a RAM test in BASIC. It performs a similar checking function that BASIC>1 executes when switched on. Locations 2000 and over are poked with 170 (bit pattern %10101010) and read back; this is repeated until the PEEKed value is no longer 170, marking either the end of RAM or a defect in a location. This process is far slower than machine-code.

Notes: [1] This command is not part of standard BASIC, and is missing on most larger machines to avoid the risk of changing other people's work. It is sometimes given other names, for example STUFF, on microcomputers.

[2] A double-byte POKE or DOKE cannot be implemented as a function definition, but requires a subroutine. DOKE Z1 (0-65535), Z2 (0-65535) is

POKE Z1, Z2-INT(Z2/256)*256: POKE Z1+1, Z2/256: REM LOW THEN HIGH

[3] This command is one of the few with a very simple machine-code equivalent, which examples i and ii illustrate. POKE 8*256*16,0 and LDA #$00/ STA $8000 each put a zero byte in the top left of the screen.

Abbreviated entry:pO   Token: $97 (151)

Operation:The parameters are evaluated by a subroutine shared with WAIT which evaluates and checks the first parameter, and converts this into a fixed point number which is stored in ($11) with BASIC>1. The comma and next* parameter are checked, and if the parameter is within the range 0-255 it is put into the X register and stored in the address in ($11) without a readback check. All the ROMs process this command similarly.

ROM entry points:BASIC1:$D6F9 (55033) BASIC2:$D707 (55047) BASIC4:$C95A (51546)

POP

BASIC command unavailable directly in CBM BASIC

PURPOSE: POP discards the last RETURN address from the BASIC stack. This in effect makes the previous GOSUB no longer effective, so that, if a RETURN is encountered, the address returned to will be the GOSUB before last's. This is useful in escaping from subroutines. For example, suppose a user is to be allowed to exit from a subroutine directly back to a menu, perhaps if the wrong routine is entered by mistake. It can often happen that a direct GOTO leaves the subroutine still active. Or imagine a game, written so that a long sequence of games can be played, and containing a routine to test for end of game: the test may check whether one player has collided with the board edge. If the test routine jumps straight to the routine which prints the score, after 24 or so games the program will stop with an ?OUT OF MEMORY ERROR.

Note that from the point of view of structured programming, this command ought to be unnecessary: such program demands the use of subroutines with one entry point, one exit, and no irregular exits with GOTO or POP.

Versions: The only previously published version I've seen is by Tom Mead, in the Liverpool Software Gazette (Oct.'80). My routine which follows is based on the RETURN command in BASIC and mimics this in all respects, except for the actual change in program control. So the address is erased but the program continues with its next statement, without a change in the flow of control. If there's no address on the stack to be popped, ?RETURN WITHOUT GOSUB is printed.

```
POP DEMONSTRATION (ALL ROMS)

0 A$=""
1 GET X$: IF X$="" GOTO 1
2 IF ASC(X$)=13 THEN PRINT: PRINT A$: RETURN
3 IF X$="X" THEN SYS 634: SYS 634: GOTO20
4 PRINT X$;: A$=A$+X$: GOTO 1
10 FOR I = 1 TO 4: PRINT I;: GOSUB 0: NEXT: RETURN
20 PRINT: PRINT "MENU": GOSUB 10: PRINT "END": END
50 FORI = 634 TO 657: READ X: POKE I,X:NEXT: GOTO 20
52 DATA 169,255,133,152,32,172,194,154,201,141,240,5,162,22
54 DATA 76,89,195,232,232,232,232,232,154,96
990 REM
1000 REM **********************************************************************
1010 REM * 'RUN 50' DEMONSTRATES POP AS AN ESCAPE KEY, TAKING USER BACK    *
1020 REM * TO MENU WHEN HE'S SELECTED A WRONG OPTION; X USED AS ESCAPE.    *
1030 REM * NOTE THAT POP WITHOUT GOSUB GIVES  ?RETURN WITHOUT GOSUB ERROR.*
1040 REM **********************************************************************
1990 REM
1995 REM
2000 REM **********************************************************************
2010 REM *            BASIC 3 VERSION IS VERY SIMILAR:                     *
2020 REM * 52 DATA 169,255,133,71,32,170,194,154,201,141,240,5,162,22      *
2030 REM * 54 DATA 76,87,195,232,232,232,232,232,154,96                    *
2040 REM *            AS IS BASIC 4 VERSION:                               *
2050 REM * 52 DATA 169,255,133,71,32,34,179,154,201,141,240,5,162,22       *
2060 REM * 54 DATA 76,207,179,232,232,232,232,232,154,96                   *
2070 REM **********************************************************************
```

POS

BASIC arithmetic function

PURPOSE: Computes the position of the cursor on its current screen line.
The range is 0-255. This is not the position on the screen line, but a measure of the distance the cursor has moved along its present line: some PRINT statements can return a value up to 255; more usually, when keying - in program lines for example, the maximum is 80.

Syntax: POS(expression). Like FRE, POS uses a dummy variable, the sole point of which is to make POS behave like a function. POS(0), POS(X), POS("") are all valid options which yield identical results.

Modes: Direct and program modes are both valid.

Examples: 61540 IF POS(0)+PEEK(196)>74 THEN PRINT CHR$(34);"[HOME][DOWN][DOWN]
         L=" L "+1:S=" J ":E=" E ":GOTO" G
     PRINT TAB(10)POS(0): PRINT SPC(10)POS(0)
     100 PRINT LEFT$("   ",12-POS(0)) ;X$

POS is arguably the least useful of all the BASIC keywords. Nevertheless it performs some useful services: the first example is taken from a routine which automatically writes the contents of RAM as DATA statements. If a system has no facility for dumping memory, as a RAM image, or if RAM has a relocatable routine, handling it as DATA may be convenient. The program line checks whether the data so far printed to the screen is in danger of reaching the end of the line. (In BASIC>1, location 196 holds 40, with a 40-column screen only, if printing is on the line one down from the top of the screen).

The second example is a direct mode line. The third figure printed depends on LEN(X$). If LEN(X$)=200, PRINT POS(0) returns 200.

The third example illustrates the close connection between POS and TAB(. If POS(0) is confined to the range 0-12, line 100 is equivalent to TAB(12). Suppose that the cursor is at position 4: then 8 spaces will be printed before X$, so the effect is the same as TAB(12).

Notes: [1] POS uses the same parameter as TAB(. Consequently POS cannot be used with printer commands unless the identical line is printed on the screen. Its usefulness is in practice limited to the screen.

Abbreviated entry: None

Token: $B9 (185)

Operation: Loads the Y register from the zero-page location storing the position of the cursor on its 'line'. This location is $C6 (198 decimal), or in the case of BASIC 1, location 5. The accumulator is loaded with #0 and a ROM routine entered which converts A and Y, as high and low bytes, into floating point form in accumulator #1.

All ROMs use identical logic to process this function. (The absolute addresses differ).

ROM entry points:

     BASIC 1: $D285 (53893)
     BASIC 2: $D27A (53882)
     BASIC 4: $C4C9 (50377)

PRINT

BASIC output command

PURPOSE: Evaluates and prints string expressions and numeric expressions to an output device, usually the screen. The appearance of the output is to some extent controllable by the punctuation of the statement, and also depends on the special graphics and screen editing characters of CBM BASIC.

Syntax: PRINT followed by arithmetic and/or string expression(s), separated from each other by one or more of: SPC(arith. exp.), TAB(arith. exp.), space, comma, semicolon, or no separator where this causes no ambiguity.

Each string and arithmetic expression must be valid and also evaluate within acceptable limits when the PRINT statement is run. The parameter for TAB and SPC must evaluate to within the range 0-255, after rounding down.

The statement terminates when a colon or end-of-line zero byte is found as part of the punctuation (i.e. not within quotes). See also the flowchart.

Modes: Direct and program modes are both valid.

Examples:
```
      FOR J=0 TO 255: PRINT CHR$(J);: NEXT: REM PRINT EVERY CHARACTER
      FOR J=0 TO 100: PRINT J,: NEXT:      REM SHOWS USE OF ','
      PRINT X+Y; 124; P*Q*(1+R%/100):      REM ARITHMETIC EXPRESSIONS
      PRINT "HELLO"; A$B$C$; LEFT$("ABCD",1):REM STRING EXPRESSIONS
      100 PRINT TI;TI$;ST;DS;DS$:           REM SPECIAL VARIABLES
```
These five print statements illustrate most of the major features of PRINT except TAB(and SPC(, which are explained elsewhere. The first is a loop which uses PRINT to output all 256 individual characters. The effect of the loop varies with the ROM; 80-column CBMs have a tremendous range of screen editing characters, so characters shift about and disappear, eventually, as the scrolling window becomes set, confining themselves to a small rectangle on the screen. Other CBMs are more sedate, merely clearing the screen and homing the cursor. The semicolon ensures that alphanumerics print next to each other; in the next example, the comma tabulates numbers into every 10th column. The third and fourth examples show typical arithmetic and string expressions respectively. Each expression is evaluated and printed from left to right, taking account of punctuation. Note that the arithmetic expressions are printed in the standard Microsoft format explained elsewhere. Semicolons have been omitted from the string output example; in fact, ABC$ prints the three strings one after the other exactly as A$;B$;C$ would. This is because the '$' symbol is recognised as a terminator. Similarly, a semicolon is not necessary after an integer's '%' or an array's ')'. Numeric variables require more careful punctuation, since their names allow a mixture of alphabetic characters and numerals. PRINT X+Y 124 is interpreted as PRINT X+Y1, as an example. This sometimes causes wrong output, but errors in PRINT statements are easy to correct. Finally, note that PRINT has routines within it to check for special values, including pi, the status indicator ST, TI and TI$, and (BASIC 4 only) the disk status variables DS and DS$.

```
      PRINT (7)(7)         :REM PRINTS  7  7
      PRINT 1.2..3         :REM PRINTS  1.2  0  .3    ['.' appears ' 0 ']
      PRINT READY.         :REM PRINTS  0  0  [i.e. value of RE then 0]
      PRINT ;,LK4*R6R;4     :REM PRINTS          0  4  [i.e. ',',LK*R6,then 4]
      PRINT 1/3(5*+--2).51 :REM PRINTS  .333333333  10  .51
```

Graphics. PET graphics characters are usually printed to the screen from a string in quotes. (They can also be poked directly into screen RAM). [RVS] is necessary to complete the character set, and doubles the number of available graphics. This extract from a program listing (in upper case mode) shows the type of thing:

```
1050 PRINT" ┌▨▨ ┐┐-─╴▼ ▉⊐────────────┐"
1055 PRINT" �,╬IT COST: "J$(5)" └────────────┐"
1060 PRINT" ▨       ⌐OME: "J$(6)"⊐|▨    FIXED? "J$(7)"⊐|"
1065 PRINT" ▨     ⌐XPORT: "J$(8)"⊐|▨    FIXED? "J$(9)"⊐|"
```

Notes: [1] The screen appearance is controlled by three factors: (i) The character generator ROM, (ii) Programmable hardware features, and (iii) The screen hardware. Taking these in order:-

(i) The oldest PETs use a character generator with upper and lower case transposed, a transitional feature from the days when upper-case was normal, so it seemed natural to produce lower-case with a shift key. All subsequent ROMs use the normal typewriter convention. The ROMs are incompatible.

(ii) POKE 59468,12 and POKE 59468,14* switch between upper case and graphics (no lower case obtainable) and lower case with upper case (losing all the QWERTY graphics, such as the card suit symbols). Try

         O POKE 59468,12: POKE 59468,14: GOTO O

to see the effect of this on a screenful of characters.

POKE 59458,62 is one of several equivalent fast-screen pokes, which cause a large and useful speed increase when printing to the screen. *CAUTION: BASIC 4 CBMs have improved screen printing speed; this POKE will not work, and can cause damage to the machine.* With early PETs and CBMs, this is perfectly safe and necessary if you wish to avoid slow screen printouts. The rule is: if the picture on the screen collapses, don't risk it again.

Wide-screen CBMs have a CRT (cathode ray tube) controller chip. This is programmable; see Chapter 9 for details.

(iii) The oldest PETs used a blue-white phosphor. All recent machines use green. Since about mid-1981, 12" screens only have been fitted, on 40 column and 80 column models. There is some incompatibility, as might be expected, between 40 column and 80 column PRINT statements. A program designed for 40 columns typically looks similar on an 80 column machine, but uses only the leftmost 40 columns - unless PRINT statements have been terminated with semi-colons, in which case the top half of the 80 column screen will be filled with double lines. Also of course BASIC 4 cursor control characters will not work on other ROMs, so scrolling windows, line erase characters and so forth cannot be downward compatible.

[2] The reverse key is necessary to obtain some characters:-

     PRINT "[RVS] [DOWN] [DOWN] "   :REM REVERSE SPACE IS A SQUARE
     PRINT "[RVS] ▪[RVSO] "   :REM PRINTS▐

This means that it is not always easy to convert a picture on the screen into a set of PRINT statements. Homing the cursor, then typing linenumbers followed by ?" and RETURN doesn't accept reversed characters; a tedious procedure of inserting [RVS] and [RVSO] will need to be used.

Abbreviated entry: ?

Token: $99 (153)

Operation: The flowchart, which applies to all the ROMs, outlines the way PRINT works. It is not a particularly long routine - a page or two of listing paper - but calls half a dozen or so other ROM subroutines.

ROM entry points:

PRINT: [SYS of this	BASIC 1: $C99F (51615)
address-6 has the same	BASIC 2: $C9AB (51627)
effect as PRINT]	BASIC 4: $BAA8 (47784)

SUBROUTINE TO PRINT ONE STRING:	
Accumulator holds low byte,	BASIC 1: $CA27 (51751)
Y-register high byte, of start	BASIC 2: $CA1C (51740)
of string; terminated by null.	BASIC 4: $BB1D (47901) has some changes

OUTPUT ROUTINE FOR SINGLE CHARACTER:	
Controls which character, if any,	BASIC 1: $CA44 (51780)
will be printed: has 5 entry	BASIC 2: $CA39 (51769)
points.	BASIC 4: $BB3A (47930)

---

*59500, which is easier to remember, may be used instead. Nick Green of Commodore UK pointed this out.

PRINT

REGET current char.

OTHER : TAB(SPC(, ;

validate.

CRLF & Return.

Validate parameter (0-255) Put into the X-register. Check for parenthesis ')'.

Subtract 10 repeatedly from cursor position till negative.

String Expression?   N

Print string using $FFD2

Convert numeral into ASCII string in buffer.

Subtract cursor position

Change sign (i.e. 2's comp-lement)

Print string + space.

Negative result?   N

Print X spaces

Y

Get next char.

End of print statement?   N

Get next char.

Cancel buffer. Print CRLF and return to BASIC.

N

End of print statement?

Return to BASIC.

FLOWCHART OF CBM BASIC'S PRINT STATEMENT PROCESSING

Notes: [1] After a colon or end of line, CRLF is always printed by BASICs 1 and 2. BASIC 4 however uses the other exit point, checking for device number; so that if location $10 (16) is < 128, C.Rtn. is output alone, without Line Feed. This was introduced to simplify writing to disks and tape. Previously, PRINT#8,X$;CHR$(13); was necessary. With BASIC 4, PRINT#8, X$ is fine (and is also compatible with the earlier form).

      [2] TAB(, SPC(, and comma have slightly different effects when printing, depending on the contents of 3 (BASIC 1), 14 (BASIC 2), and 16 (BASIC 4). If this location holds zero, the skip effect is achieved by CBM cursor right characters, and CRLF is printed at the end of the line; a non-zero value prints spaces instead, and no automatic CRLF, so non-CBM equipment may be used. In quotes mode, skip shows as reverse].

PRINT#

BASIC output command

PURPOSE: Evaluates and prints string expressions and numeric expressions to an output device, usually printer, disk, or tape. The appearance of the output is identical to that produced by PRINT, except for possible differences in interpretation of special CBM editing characters.

Syntax: PRINT# arithmetic expression [, expressions to be printed in format identical to PRINT]. The comma is a separator to make unambiguous such statements as PRINT#3,3 and PRINT#33. There must be no space between 'PRINT' and '#', because this will interpret into two bytes ('PRINT' and '#' separately, not the single 'PRINT#' token.) - Except in BASIC 1! Finally, the expression immediately following '#' must conform to two criteria: after evaluation and rounding-down if non-integral its range must be 1-255; secondly, a file with this number ('logical file number') must be open.

Modes: Direct and program modes are both valid.

Examples: 100 OPEN 4,4 : REM CHANNEL 4 TO PRINTER OPEN; NON-CBM PRINTER...
            1000 PRINT#4,CHR$(12) "PRICE LIST   no."N$"   page"P%

            100 OPEN 1,4,1: OPEN 2,4,2: REM 2 CHANNELS TO SAME CBM PRINTER...
            110 PRINT #2,"$$$$$9.99" : REM CBM PRINTER FORMAT USES SEC.ADDR.=2
            1000 PRINT#1,DOLLARS   : REM OUTPUTS IN DESIRED FORMAT (EG.$24.00)

            2000 PRINT#4,;: CLOSE4   : REM CLOSES WITHOUT C.RTN.(WITH PRINT,CMD)

            10 OPEN 5,8,5,"1:FIRST FILE,SEQ,W" :REM CBM DISK EXAMPLE
            20 FOR J=1 TO 20: PRINT#5,"RECORD NUMBER"J: NEXT :REM BASIC 4

            5000 PRINT#4,XYZ$; CHR$(13);: REM BASICS 1 AND 2 NEED THIS

These examples are confined to printers and disks only, but in practice of course files can be opened to tape, screen, or any IEEE device, CBM or otherwise. The first sets of examples contrast the way a non-IEEE printer (e.g. Qume) is controlled with CBM's IEEE device. Assuming a hardware interface exists to convert IEEE to (say) RS232, the file can be opened as usual, and control characters sent to the printer to alter its spacing or line separation or other feature, or, here, send a form feed command. CBM printers rely on the IEEE's secondary address feature to control the printer in addition to control characters, and the example shows how PRINT# can distinguish between several files open at one time. Line 2000 shows how a file is closed if CMD and PRINT were used: see note [1] on this. The final examples show another complication involving disk files (not tape, and not BASIC 4). The earlier ROMs wrote linefeed characters to disk, after the carriage returns which are used as record separators. Consequently, data when read back from disk started with an unwanted linefeed character, and this CHR$(10) could be suppressed only by printing CHR$(13), i.e. return, at the end of a record. Footnote 1 of the PRINT statement flowchart amplifies this is somewhat greater detail.

Notes: [1] PRINT#, PRINT, and CMD are intimately related:
            PRINT# = SEND 'LISTEN'/ PRINT/ SEND 'UNLISTEN'
            CMD   = SEND 'LISTEN'/ PRINT
            PRINT = PRINT
        This is why PRINT#n,;: and CMDn,;: are opposites. Each prints nothing, then PRINT# unlistens the device, while CMD leaves it listening. It is also the reason for line 2000 in the examples; before closing the file, if CMD was used, PRINT#,;: has the function only of unlistening the disk or printer. Note that PRINT# is better than CMD and PRINT with CBM printers, which are apt to turn the 'listen' off. In practice all this is easier than it may appear to be; it is not essential, in getting data stored and printed, to appreciate all the fine points of these commands.

        [2] Some printers (not CBM) won't print out their internally stored buffer until a carrage return (CHR$(13)) has been received. Earlier data is thrown away while it waits for this to happen. OPEN 4,4: FOR J=14 TO 255: PRINT#4,

CHR$(J);: NEXT may simply do nothing. An apparently unresponsive printer may owe its impassivity to oversight of this simple fact.

Another potential problem if carriage returns are omitted is that INPUT#, which is the mirror-image of PRINT# as far as individual records are concerned, can safely accept strings of only 80 characters or less. PRINT#n,X$; prints fields strung together, so INPUT# will take in a composite string, possibly with a ?STRING TOO LONG ERROR.

Abbreviated entry: pR (includes #) *

Token: $98 (152)

Operation: The ROM entry for this command has only two machine code instructions. The first calls CMD, which carries out the validation for parameter and comma, sets the output device, and performs PRINT. When it returns from PRINT the second routine unlistens the file and restores the normal devices of keyboard and screen. All of this helps explain the interconnectedness between PRINT#, PRINT, and CMD.

ROM entry points:

PRINT#:	BASIC 1:	$C97F	(51583)
	BASIC 2:	$C98B	(51595)
	BASIC 4:	$BA88	(47752)
CRLF	BASIC 1:	$C9CE	(51662)
ROUTINE:	BASIC 2:	$C9DE	(51678)
	BASIC 4:	$BADB	(47835)
RESTORE	BASIC 1:	$CAD6	(51926)
DEFAULT	BASIC 2:	$CAB7	(51895)
DEVICES:	BASIC 4:	$BBB4	(48052)

---

*If the abbreviation ?# is used, on listing the line will show PRINT# but give ?SYNTAX ERROR on running, because the interpreter 'sees' PRINT #, which means nothing. BUT if the cursor is positioned on such a line and Return pressed, the correct meaning of PRINT# is taken in with the line. P shift-R is simpler.

PRINT USING
Output command unavailable directly in CBM BASIC

PURPOSE: Prints data, often numeric, in a format specified by the program. Currency signs, trailing zeros, + or - signs, commas, 'CR' if a quantity is negative: these are all typical features of formatting commands. Strings usually pose less of a problem than numerals.

Versions: COBOL, a major business language, set a standard which most other formatting commands derive from. Its 'picture' clause enables the programmer to position the decimal point and insert spaces and commas within numerals, in addition to the features already mentioned. Thus, PIC $ZZZ,ZZ9.99CR causes -1234.5 to be printed as $ 1,234.50CR. The IBM 8000 series of desk-top machines and the TRS-80 use a similar notation, except that 9, which in COBOL prints a compulsory numeral, is replaced by #. Commodore's printers enable one (only) format to be pre-defined, and the formatting field is nearly pure COBOL.*

Users of CBM printers apart, there is a constant demand for routines to format the output both onto the screen and to hardcopy and disk file. Both 'Diskpro' and 'Commando' include a formatting command. It does not validate its input completely; it is possible for oversized strings to be printed wrongly. BASIC versions have been published, some of them ludicrously long. For such a useful command, this can be rather discouraging. In the hope of improving the situation I present on the next few pages details of a relocatable machine code formatter which is relatively bug-free.

Notes: The print routine can be called by SYS 47778 in BASIC 4 (and SYS 51621 in BASIC 2, SYS 51609 in BASIC 1). This allows for string and numeral expressions, TI, TAB(and SPC(and so on. For example, we may set PR=47778 so SYS (PR) 8*9"Hello" prints '72 Hello'. In fact this entire routine, the main parts of which are very compact, can be moved into RAM and modified there. This is *not* however the way the following routine operates; it uses a SYS call followed by the numeric expression to be printed. This one value alone is formatted and output.[2] The central piece of code is this (BASIC 4):

```
JSR $BD98 ; Input and evaluate any expression from BASIC
JSR $CF93, ; Convert contents of accumulator#1 into ASCII string
JSR $xxxx ; Code which processes the string, held in $0100 ff.
JSR $BB1D ; Print string using A (low), Y (high) pointers
RTS       ; Return to BASIC
```

The idea is simply to print the number in the way it would in any case be printed, but in addition to insert a piece of code to edit the output buffer from which the string is printed. The actual processing is controlled by several pokes, to control the type (decimal/integer), the desired string output length, the number of decimal places (where applicable), and filler characters. A leading character is selectable for positive numerals only. All this should be made clear by the examples.

---

*CBM printers are noted for bugs in their ROMs, so the actual output may not always be as expected.

[2] The parentheses are used purely to separate SYS (PR) from the subsequent data. It is perhaps worth pointing out that misunderstanding of the syntax can cause bugs. Thus: SYS (PR) 45 prints a formatted version of 45; but SYS PR45 inputs and evaluates PR45, which the translater considers to be the same as PR, so zero will be printed. And SYS PR(45) evaluates the 45th element of array PR(), if this array exists; if not, ?BAD SUBSCRIPT ERROR will be printed.

RELOCATABLE BASIC LOADER FOR 'PRINT USING' TYPE ROUTINE (NUMERAL FORMATTER).

```
0 DATA  1,8,2,32,162,0,221,0,1,240,6,232,224,12,208,246,24,96,169,69,32,-162
1 DATA  176,90,173,-166,240,94,173,2,1,208,11,172,-165,169,48,153,2,1,136
2 DATA  208,250,169,46,32,-162,168,144,2,160,48,169,0,32,-162,152,157,0,1,169
3 DATA  46,32,-162,172,-164,232,136,208,252,236,-165,176,33,172,-165,169,0
4 DATA  153,1,1,189,0,1,201,32,208,3,169,32,234,153,0,1,202,16,6,173,-163,136
5 DATA  16,244,136,16,231,169,0,133,97,160,1,132,98,96,169,0,32,-162,144,240
6 DATA  138,168,173,2,1,240,9,169,46,32,-162,144,2,138,168,152,170,202,16,181
7 DATA  0,32,159,204,32,233,220,32,-148,32,28,202,96
8 REM
9 REM
10 PRINT"[CLEAR][REVS] ROM2 RELOCATING 'PRINT USING' ROUTINE "
20 T = PEEK (52) + 256*PEEK (53) : REM T IS CURRENT TOP OF BASIC MEMORY
30 L = T - 166 : REM PROGRAM IS 166 BYTES IN LENGTH
40 FOR J = L TO T-1 : REM LOOP PLACES ROUTINE IN TOP OF AVAILABLE MEMORY
50 READ X%: IF X%<0 THEN Y = X% + T: X% = Y/256 : Z = Y - X%*256 : POKE J,Z: J=J+1
60 POKE J , X%
70 NEXT
100 X% = L/256 : Z = L - X%*256 : REM WILL BE HI & LO BYTES OF NEW TOP OF MEMORY
110 POKE 48,Z : POKE 50,Z : POKE 52,Z : REM SET LO BYTES OF MEMORY AND STRINGS
120 POKE 49,X%: POKE 51,X%: POKE 53,X%: REM SET HI BYTES OF MEMORY AND STRINGS
123 REM
124 REM
125 REM ### NOW PRINT OUT INSTRUCTIONS FOR USE, WITH ADDRESSES, ONTO SCREEN ###
130 PRINT "[DOWN]SYS (" ; L+153 ; ") FOLLOWED BY ANY NUMERIC
131 PRINT "EXPRESSION PRINTS THE FORMATTED VALUE,
132 PRINT"KEEPING THE CURSOR ON THE SAME LINE.
133 PRINT"DECIMALS ARE TRUNCATED; ROUND TO NEAREST";
134 PRINT"WITH +.[0][0]...[0]5 IF REQUIRED
140 PRINT"[DOWN]POKE" ; L ; "[LEFT],1 FOR DECIMAL,0 FOR INTEGER
150 PRINT"POKE" ; L+1 ; "TOTAL LENGTH OF OUTPUT - 1
160 PRINT"POKE" ; L+2 ; "NUMBER OF DECIMAL PLACES
170 PRINT"POKE" ; L+3 ; "FILLER CHARACTERS
180 PRINT"POKE" ; L+98; "LEADING CHARACTER WHEN +VE
190 PRINT"[DOWN]SAVE FROM"L"TO"T-1
200 PRINT"          ($";:GOSUB500:PRINT" TO $";:L=T-1:GOSUB 500:PRINT")"
210 PRINT"[DOWN]SET UP WITH LENGTH 9, 2 DECIMAL PLACES, AND LEADING SPACES.
250 END
497 REM
498 REM
499 REM   ### ONE LINE DECIMAL TO HEX CONVERTER ###
500 L=L/4096:FORJ=1TO4:L%=L:L$=CHR$(48+L%-(L%>9)*7):PRINTL$;:L=16*(L-L%):NEXT:RETURN
READY.
```

```
******   BASIC 4 ******
7 DATA 0,32,152,189,32,147,207,32,-148,32,29,187,96
10 PRINT "[CLEAR][RVS] ROM4 RELOCATING 'PRINT USING' ROUTINE "

******   BASIC 1 ******
7 DATA 0,32,184,204,32,175,220,32,-148,32,39,202,96
10 PRINT "[CLEAR][RVS] ROM1 RELOCATING 'PRINT USING' ROUTINE "
20 T = PEEK(134) + 256*PEEK(135) : REM T IS CURRENT TOP OF BASIC MEMORY
110 POKE 130,Z : POKE 132,Z: POKE 134,Z
120 POKE 131,X%: POKE 133,X%:POKE 135,X%
```

```
ROM4 RELOCATING 'PRINT USING' ROUTINE

SYS ( 32755 ) FOLLOWED BY ANY NUMERIC
EXPRESSION PRINTS THE FORMATTED VALUE,
KEEPING THE CURSOR ON THE SAME LINE.
DECIMALS ARE TRUNCATED; ROUND TO NEAREST
WITH +.[0][0]...[0]5 IF REQUIRED

POKE 32602,1 FOR DECIMAL,0 FOR INTEGER
POKE 32603 TOTAL LENGTH OF OUTPUT - 1
POKE 32604 NUMBER OF DECIMAL PLACES
POKE 32605 FILLER CHARACTERS
POKE 32700 LEADING CHARACTER WHEN +VE

SAVE FROM 32602 TO 32767
         ($7F5A TO $7FFF)

SET UP WITH LENGTH 9, 2 DECIMAL PLACES,
AND LEADING SPACES.
```

This example shows the effect of running the routine with a 32K machine
containing no machine-code in the top of memory. The screen output should
appear exactly as shown. The routine occupies 166 bytes just below the
screen RAM. Instructions, with the relevant memory locations, are shown,
and may be noted by the programmer for future use. Note that this routine
is protected in memory by the loader; if the program is stored and reloaded,
it will need to be memory-protected, and also of course must not overwrite
other routines. It is set up to print a string of length 9, in decimal format,
(so integers appear with '.00' at the end) and with leading spaces. So:-

      SYS(32755) 1/3   prints   .33
POKE 32604,4 sets the number of decimal places to 4; now
      SYS(32755) SQR(12)   prints   3.4641
POKE 32700,ASC("$") makes the leading character for positive numbers the $:
      SYS(32755) 123+10   prints $133.0000

Demonstration program: To make the POKEs more comprehensible, I have used
    meaningful variable names. The actual POKE values will differ for non-32K
    machines. Note the last line of formatted printout, which is exactly what will
    appear on a screen or any printer. If an 'E' is present in the output, this
    routine does not attempt to process it, but prints it verbatim.

```
0 PRNT=32755:SWITCH=32602:LNGTH=32603:DECPTS=32604:CHAR=32605:LDGCHAR=32700
5 FOR J = -10 TO 100 STEP 10: PRINT
10 POKE SWITCH,0: POKE LNGTH,4: POKE CHAR,42: POKE LDGCHAR,42: SYS(PR)J
20 POKE SWITCH,1: POKE LNGTH,7: POKE CHAR,32: POKE LDGCHAR,32:
   POKE DECPTS,4: SYS(PR)1/(1+J)
30 POKE DECPTS,2: POKE LDCHAR,ASC("$"): SYS(PR)100*(1+J/100)+.005
40 POKE SWITCH,0: POKE LDGCHAR,32: POKE LNGTH,7: SYS(PR)J*J*J
50 PRINT " ";: POKE SWITCH,1: POKE LDGCHAR,48: POKE CHAR,48:
   SYS(PR)SQR(ABS(J))
60 NEXT
```

```
**-10 -.1111  $90.00     -1000 00003.16
****0 1.0000 $100.00         0 00000.00
***10  .0909 $110.00      1000 00003.16
***20  .0476 $120.00      8000 00004.47
***30  .0322 $130.00     27000 00005.47
***40  .0243 $140.00     64000 00006.32
***50  .0196 $150.00    125000 00007.07
***60  .0163 $160.00    216000 00007.74
***70  .0140 $170.00    343000 00008.36
***80  .0123 $180.00    512000 00008.94
***90  .0109 $190.00    729000 00009.48
**100 9.9009901E-03 $200.00 1000000 00010.00
READY.
```

READ
BASIC program data input command

PURPOSE: Reads data stored in DATA statements. Each READ fetches one item of data and assigns it a variable name. Originally this command was the primary means by which a program obtained its data, which the machine accepted from punched cards.

Syntax: READ must be followed by a variable, or a list of variables separated by commas. These may be integer, string, or numeric, and can be arrays. If the type of variable does not match the corresponding DATA, this can be detected only at run time, and will cause a type mismatch error.

Modes: Direct and program modes are both valid. Direct mode requires the presence of a program containing data statements in memory, otherwise an ?OUT OF DATA ERROR message appears. (This is what happens when 'READY.' is under the cursor and return is pressed).

Examples: 0 DATA 154: READ X: DIM X$(X): FOR J=1 TO N: READ X$(J): NEXT
Where the total amount of data is not fixed, a routine like this   may be valuable: the first item of data, which will require periodic updating, holds the number of current data items; an array is dimensioned with this number and therefore capable of holding all the items; finally, the data is read directly into the array in a loop.

   50 READ MC%: IF MC%<0 THEN ADDRESS=MC%+T: HI%=AD/256: LO%=AD-256*HI%

This example also shows how special values can be used as indicators that special processing is required. In relocating loaders, most machine code bytes are straightforward POKEs; only absolute addresses vary with the situation of the code. A minus sign, holding the difference between (say, as here) the top of memory after relocation and the position within the code, is a signal to compute the low and high bytes needed.

   FOR L=1 TO 10: READ X$: PRINT X$: NEXT

This direct-mode line reads the next 10 data items from the stored program and prints them in a column on the screen.

Notes: [1] This routine shares the ROM routines of INPUT and GET, and has a lot in common with them. The statement READ AA%,B,N$(6),C(20) is valid, and provided that the DATA is stored to correspond, will execute successfully. If it is not, a type mismatch error will be printed; this is caused by something like READ X when the data pointer indicates, say, NW3. READ with an integer variable rounds down floating-point numbers. Generally, READ X$ will give no trouble, and will read any data; but it can be good to check the number of numeric data items by reading into a numeric variable. No evaluation is carried out by this routine, any more than GET or INPUT; DATA 15*1.25 can be read only as a string.

[2] Mismatches cause ?SYNTAX ERROR, but in the DATA statement line. A similar bug occurs when a function definition is incorrect; in this case too the associated line is wrongly flagged as containing the error.

Abbreviated entry: rE   Token: $87 (135)

Operation: The elaborate routines shared by GET, GET#, INPUT, INPUT# and READ are distinguished within the routine by flags: a special location ($0B; $62 in BASIC 1) holds #$98 for READ, #$40 for GET, #$00 for INPUT. Thus READ is signalled by the N bit, GET by the V bit, and INPUT by the Z bit. Also two variable locations check for mismatches: $07 has #$FF for a string, #$00 if numeric; $08 has #$80 for an integer, #$00 for floating point. ($5E, $5F in BASIC 1). READ uses a routine which scans BASIC statements - not lines - searching for DATA tokens. All the ROMs process this routine in roughly the same way.

ROM entry points: BASIC1:$CB24 (52004) BASIC2:$CB07 (51975) BASIC4:$BC02 (48130)

REM

BASIC remark command

PURPOSE: Permits comments to be included in BASIC programs. These comments can in general be LISTed with the program, but are ignored by the program during execution.

Syntax: REM may be followed by any characters, including :, which in this case does not function as a statement separator. Everything after REM and before the next line is ignored.

Modes: Direct and program modes are both valid.

Examples: 7000 REM *** MAIN CONTROL LOOP ***
              7003 GOSUB 51000: REM LOWER MEMORY TO ALLOW ROOM FOR 2 BUFFERS
              7006 GOSUB 59000: REM PRINT INSTRUCTIONS AND AWAIT SPACE BAR
              7009 GOSUB 50000: REM SET UP NUMERALS, ARRAYS, STRINGS, VARIABLES
              7012 GOSUB 57000: REM PRINT SCREEN FOR PARAMETER INPUT

              30000 REM

              ** MOVE TO TOP OF NEW PAGE AND PRINT TITLE **

              NP$="":FORLS=59TO62:NP$=NP$+CHR$(PEEK(LT+LS)):NEXT:REMSUB90:O$(1)=NP$

The first example shows the most common use of remark statements, making the functions of a program clearer than they would otherwise be. A block of REM statements may be written before a program or subroutine, giving very detailed explanations of the working. See elsewhere in this book for examples. Line 30000 illustrates one of the many tricks available with REM, which rely on the fact that syntax after the REM is unimportant. Here, 2 carriage return characters have been POKEd into the comment statement, which therefore prints its message onto a new line. REM is sometimes usable in direct mode; the last example is a line taken from a program, with its linenumber erased so it will run in direct mode; the REM near the end of the line prevents execution of an unwanted part of the code.

Notes: [1] LIST may produce strange effects with REM. Unshifted alphanumerics after REM appear as ordinary text, but shifted characters (unless within quotes) are interpreted as tokens and converted into reserved words, often expanding the LISTed line a great deal. Other characters - clear screen, form feed, cursor down, and so on - can, as we've seen, be incorporated into comments: for programs to do this, see Chapter 2. ?SYNTAX ERROR is caused on LIST, and the listing will stop, if shift-K (BASIC 1), shift-L (BASIC 2), or CHR$(219) in BASIC 4, are included in a REM statement. The 8032 keyboard does not possess chr$(219), which is shift-[.

[2] If your intention is to remove remarks from the finished program, it may be worth reserving linenumbers (say) ending 6-9 for REMs, and never branching to these lines with GOTO or GOSUB.

Abbreviated entry: None

Token: $8F (143)

Operation: This routine scans for an end-of-line zero byte; when this is found, the Y register holds the number of bytes offset from the current CHRGET position. Y is transferred to the accumulator and added to CHRGET, so program control is transferred from REM to the next BASIC line.

This routine is embedded in the middle of IF, so that, should a condition be false, the equivalent of REM is carried out - i.e. the remainder of the line is ignored. DATA is a similar routine, except that, in addition to the end-of-line byte, it accepts a colon; so it skips to the next statement, not the next line.

ROM entry points:

         BASIC 1: $C833 (51251)   BASIC 2: $C843 (51267)   BASIC 4: $B8C6 (47302)

RENUMBER
BASIC system command unavailable directly in CBM BASIC

PURPOSE: Changes linenumbers of a BASIC program non-manually, either to give an improved appearance or to permit additional lines to be inserted. Other reasons may exist, too: a range of linenumbers (say, all over 60000) may be required to permit successful appends; a program containing very low line numbers runs (slightly) faster than otherwise; and so on.

Versions: Many versions of RENUMBER exist. At the time of writing none has all the features required by a professional utility. This is odd, since for example Apple has had a good renumber routine for years. The earliest versions include J Butterfield's BASIC program and Bill Seiler's machine code routine.* Later, Toolkit included a routine similar in effect to Seiler's which renumbers only in constant increments. (So lines typically emerge as 100,110,120,...). Eventually, so-called '4-parameter' renumbers were written, to use a format like this:
>   RENUMBER 999,1500,1000,10

which would convert lines 999-1500 into 1000,1010,1020,... Ideally a renumber should also resequence, so that for example a subroutine could be shifted to a new position in a program. None seem to be available that do this.

Operation: This command is more difficult to program than may appear at first sight. The problems lie in modifying the references within lines. As this short program shows, there is no problem in changing the linenumbers themselves:

```
59999 REM*** TINY RELOCATABLE RENUMBER **
60000 A=1025:B=256:PRINT"LO/HI LINES, NEW START & INCREMENT:":INPUT L,H,S,I
60005 FOR R=0 TO 5E4:IF PEEK(A+2)+B*PEEK(A+3)<LTHENA=PEEK(A)+B*PEEK(A+1): NEXT
60010 FOR R=0TO5E4: X=S+R*I: IF A=0 OR PEEK(A+2)+B*PEEK(A+3)>H THEN END
60015 POKEA+3,INT(X/B): POKEA+2,X-(INT(X/B))*B: A=PEEK(A)+B*PEEK(A+1): NEXT
```

BASIC holds linenumbers as ASCII strings, so these have to be sought and changed. At least two passes are necessary, the first to store current linenumbers, the second to change references. Since GOTO 5000 may be renumbered GOTO 10000, lines must be expanded to accommodate extra bytes. (Some inferior routines require lines all to be written with five figures, GOTO 01000 style). The syntax generally has to be assumed to be correct. IF X=0 THEN 10=X is syntactically wrong, but might be renumbered as though 10 were a linenumber. Some instructions may contain line references which only the programmer can deal with; for example, SYS or USR commands/ functions, or DATA statements, or computed GOTOs and GOSUBs written to include true computed destination lines.

These esoteric problems aside, all renumbers need to deal with:
i.   IF ... THEN linenumber.
ii.   GOTO linenumber and GO TO linenumber. (In BASIC>1 these differ).
iii. GOSUB linenumber.
iv.   ON ... GOTO and ON ... GOSUB have a list of lines to be changed.
v.   LIST with optional linenumbers and RUN with optional number are the only other commands which include linenumbers among their parameters. RUN linenumber is more important than LIST.
vi.   If the destination line doesn't exist, this must be flagged as an error.
vii. The renumbered lines must be checked for overlap with original lines.
viii. The renumbered lines may be outside the valid range.
ix.   Possibly the new program or one of its lines may become too long. Fortunately this is very unlikely.

---

*Slightly misprinted in Micro (first line should hold T=0), and in PET User Notes (Nov-Dec. '78), respectively. Others, e.g. IPUG, have their own versions. Power, Disk-O-Pro, etc. have four-parameter renumberers.

RESTORE

BASIC data command

PURPOSE: Resets the data pointer within a program so that data stored in the program is READ from the earliest DATA statement. Data will then be READ sequentially until a further RESTORE.

Syntax: Restore has no other parameters.

Modes: Direct and program modes are both valid.

Examples:

```
15000 REM RELOCATABLE DATA ROUTINE FOR HASHTOTAL
15010 DATA HASHTOTAL,169,0,141,202,3,162,1,160,20
      . . .
15050 RESTORE: FOR L=0TO9E9: READ X$: IF X$<>"HASHTOTAL" THEN NEXT
15060 FOR L=971 TO 1016: READ LS: POKE L,LS: NEXT

2000 READ X$: IF X$="END" THEN RESTORE
```

Program lines 15000-15060 demonstrate a method to ensure that DATA and READ routines always correspond correctly. All that's needed is a name for the routine, included at the start of the data, which acts as a label and can be searched for. With only a few routines, this method is not necessary. But with a great many it may be valuable.

Line 2000 is an analogous, but different, situation where there is DATA which is required to be recirculated: perhaps in a games program with stored 'random' names or objects. If the last item of data is END, and each READ tests for it as line 2000 does, there will never be an ?OUT OF DATA ERROR.

Notes: [1] NEW, RUN, and CLR call RESTORE as an automatic part of their functioning. RESTORE in direct mode followed by RUN therefore does nothing which RUN on its own would not do. However, RESTORE in direct mode followed by GOTO linenumber allows separate control of DATA and program variables, and can be useful sometimes.

[2] It is possible to set the pointer, not to the start of BASIC, but to point at other items of DATA. See G.Yob's Creative Computing article on this theme, also in Printout of Oct. 1980.

Abbreviated entry: reS

Token: $8C (140)

Operation: Decrements and stores the contents of ($28) - start of BASIC - into ($3E) - DATA pointer. In BASIC 1, from ($7A) into ($90). Connoisseurs of small programming points might like to compare this with a routine to reset GETCHR. This routine, just after CLR, adds $FFFF to the start of BASIC, saving 1 byte over RESTORE, which subtracts 1.

ROM entry points:

```
BASIC 1: $C70D (50957)
BASIC 2: $C730 (50992)
BASIC 4: $B7B7 (47031)
```

RETURN
BASIC command

PURPOSE: Changes program flow of control to the statement immediately after the most recent GOSUB statement. These two commands therefore permit subroutines to be automatically processed without the need to keep a note in the program of the return addresses.

Syntax: RETURN stands alone with no parameters. It may only be followed by spaces (these are optional) and must be followed by either a colon or an end of line. If no GOSUB corresponds to a RETURN - for example, when 0 RETURN is RUN - the error message ?RETURN WITHOUT GOSUB appears.

Modes: Direct and program modes are both valid.

Examples: `10 INPUT L: GOSUB 10000: GOTO 10: REM TESTS SUBROUTINE AT 10000`

```
10000 L=(INT((L-.005)/RQ)+1)*RQ: REM RQ=ROUNDING QTY; .01,.05,.50 &c
10010 PRINT L;: RETURN
```

This example shows a test routine which allows the user to input any number and responds with the result of the subroutine's processing. Here, we have the first stage of a BASIC round-and-format routine which rounds up by an amount varying with parameter RQ. The next stage adds decimal points to integers, and generally tidies up, but the point here is that the subroutine is tucked away in a completely different part of memory, but the RETURN automatically transfers control back, to line 10 in this case.

```
50 GO TO 1000
100 REM *** MORE PROGRAM ***
270 GOTO 1000
300 REM *** MORE PROGRAM ***
1000 REM SUBROUTINE WITHOUT A 'GOSUB'
1010 REM *** PROCESS DATA WITHIN THE ROUTINE ***
1100 REM GOTO 100? GOTO 270? GO ELSEWHERE??
```

This second example is an attempt to explain the difficulty of having no GOSUB/ RETURN commands. What is handled effortlessly with these commands becomes a problem without them.

Notes: [1] GOSUB and FOR share the same method of using the stack: data is pushed on the stack in either case, and the stack pointer is left pointing to a token, which may be either FOR or GOSUB. This double use doesn't cause conflict unless certain combinations of BASIC are tried. This for instance causes ?NEXT WITHOUT FOR ERROR:

```
10 GOSUB 1000: NEXT J
1000 FOR J=0 TO 10: RETURN
```

Occasionally, a FOR variable may be erased like this; but it is an easy programming error to avoid.

[2] RETURN can sometimes be used in direct mode; for example:
10 GOSUB 100: PRINT "RETURN" / 100 END. This is not often useful.

[3] *This command has no connection with the carriage return key.*

Abbreviated entry: reT   Token: $8E (142)

Operation: After validating the command, the routine to check FOR and GOSUB tokens is called: a flag is set before entry to show that RETURN is the command, not NEXT. FORs are removed and $8E tokens looked for; if none is found, ?RETURN WITHOUT GOSUB is printed. When found, the BASIC linenumber and CHRGET pointer are recovered, as they were left by GOSUB. The routine now merges with DATA. So it searches for the next statement after the pointer; thus, ON X GOSUB 10,20,30:PRINT X ... RETURN causes the remaining list of variables to be ignored; the colon, or if no colon the end of line zero byte, marks the point at which execution recommences.

ROM entry points: BASIC1:$C7CA (51146) BASIC2:$C7DA (51162) BASIC4:$B85D (47197)

RIGHT$
BASIC string function

PURPOSE: Extracts a substring, consisting of the rightmost characters, from a string. This function, with MID$, RIGHT$ and the string concatenation operator +, is used in text and string processing in BASIC.

Syntax: RIGHT$(string expression, arithmetic expression). The string expression must be valid, i.e. made up from string functions and/or literals and/or string variables. Its length cannot exceed 255. The maximum value of the arithmetic expression is 255; the minimum value depends on the ROM. BASICs prior to 4 will not accept a value of zero. BASIC 4 has been modified to accept zero, returning the null string with RIGHT$(X$,0).

Modes: Direct and program modes are both valid.

Examples:
```
PRINT RIGHT$("REAGAN",2)     :REM RESULT IS AN
10 PRINT RIGHT$("          "+STR$(N),10) : REM ANOTHER TAB(

99 REM ** W$ HOLDS TARGET WORD; G$=GUESS LETTER; S$=STRING SO FAR
100 FOR J=1 TO LEN(W$): S$=S$+"-": NEXT: REM IF W$=HELLO, S$=-----
    ...
200 FOR J=1 TO LEN(W$): IF G$=MID$(W$,J,1) THEN GOSUB 1000
210 NEXT
    ...
1000 S$=LEFT$(S$,J-1) + G$ + RIGHT$(S$,LEN(S$)-J): RETURN

51220 CC$="[DOWN][RIGHT][DOWN][RIGHT][DOWN][RIGHT]"
```

The first example shows a straightforward application of this function. The second is another version of a TAB(style routine to help align printed output.

The object of the third example, taken from a word game ('Hangman'), is to illustrate complicated string handling. It is fairly obvious that any new string can be built up from existing strings by subdividing as far as is necessary using MID$ (or the related LEFT$ and RIGHT$ functions) and putting the bits together with +. Line 1000 is a subroutine which does this, breaking S$ into two parts and connecting them with the correctly guessed letter in between. If G$="L", the loop in line 200 calls subroutine 1000 twice, when J=3 and J=4. The result is to convert "-----" into "--LL-" or "H----" into "H-LL-" or whatever. A similar scanning process can be applied to many problems in word guessing, multiple-choice questions, foreign language quizzes, and so on. See Chapter 4, section 4.1.15.

Lastly, since string functions may be used in PRINT statements, PET's screen formatting characters can be processed in this way too; line 51220 defines CC$ so that PRINT RIGHT$(CC$,2*N) moves the cursor diagonally down the screen when N>0 and N<4. (It's only a very simple example!).

Notes: [1] This diagram should make the operation of this function clear:

X$="	S	T	R	I	N	G	"
Position from right:	6	5	4	3	2	1	

PRINT RIGHT$(X$,4) prints the four rightmost characters: RING.

[2] RIGHT$(X$,N) can be replaced by MID$(X$, LEN(X$)-N+1). A special case is the expression RIGHT$(X$, LEN(X$)-1) which removes the leading character from a string; MID$(X$,2) performs the same function. BASIC<4 rejects a zero length parameter; this conversion therefore can be useful in avoiding clumsy code. Line 1000, which fails with BASIC<4, shows this.

Abbreviated entry: rI (includes $)   **Token:** $C9 (201)

Operation: The string parameters are recovered from the stack. Then a 2's complement routine computes the length of the (original) string minus the parameter. This value (one byte) is held in A. Now LEFT$ is entered and processing proceeds as for LEFT$, except that A does not contain #0.

ROM entry points: BASIC1:$D604 (54788)   BASIC2:$D606 (54790)   BASIC4:$C862 (51298)

RND

BASIC arithmetic function

PURPOSE: Generates a pseudo-random floating-point number in the range 0-1 excluding the limits. RND can mimic statistical data in simulations, help generate test data, and introduce unpredictability generally.

Syntax: RND (arithmetic expression). The arithmetic expression may take any value within the valid range of floating-point numbers (± 1.7 E 38 approx.). Only the sign influences the result, not the magnitude.

Examples:
```
100 FOR J=0 TO 3000*RND(1) : NEXT : REM DELAY OF 0 TO THREE SECONDS
100 N=70: FOR J=0 TO N*RND(1): READ X$: NEXT: REM READS 1 DATA ITEM

100 FOR J=1 TO 9: P= PEEK(32809 + RND(1)*920):IF PEEK(P)=32 THEN
    POKE P,176+J : REM IF A SCREEN LOCATION IS EMPTY, PUT RVS 1-9 IN
110 NEXT

100 IF RND(1)<.1 THEN PRINT "A VERY GOOD MORNING TO YOU"

0 PRINT "[CLEAR]": I=33228: GOTO 10
1 J=-41:RETURN/2 J=-40:RETURN/3J=-39:RETURN/4J=-1:RETURN
5 J=1: RETURN/6 J=39:RETURN/7 J=40:RETURN/8 J=41:RETURN
10 ON RND(1)*8+1 GOSUB 1,2,3,4,5,6,7,8
20 M=I: I=I+J
30 IF I<32768 THEN I=I+1000
40 IF I>33767 THEN I=I-1000
50 POKE I,81: POKE M,32: GOTO 10: REM 'BALL' AND BLANK POKED IN
```

The first and second examples use a loop in which the final value varies with the random number selected: this causes a random delay in the first example (usable perhaps in a reaction-time game) and the selection of a random string in the second, assuming a list of 70 data items exists in the program. (Usable in a foreign-words quiz or guessing game). The next example is a short program, designed to place 9 values onto a 40-column screen, at random, but ignoring the top and bottom lines. Still another line 100 follows, and this one has a one in ten chance of printing its greeting. Finally, we have a comparatively long program, which relies on RND to pick one of eight subroutines. (The slash marks are there to save space). It is a simple version of a 'random walk'.

Notes: [1] RND generates determinate numbers, not 'random' numbers, if indeed these can exist. The sign of the argument (+,0,or-) affects the numbers computed. A special location holds the last random number: at switch on this has a constant put into it, and every subsequent call of RND resets it. Any constant value is called a 'seed'. RND(+ve) computes the next value in an infinite sequence. It is like taking the remainders after dividing 10 by 7; the pseudo-random sequence 3,2,6,8,5,7,1, ... is formed and continues indefinitely. After about 45000 repetitions I have the impression that the CBM series lose their 'randomness' and become more predictable. If the seed is fixed, the subsequent random numbers can be repeated; and RND(-ve) puts a function of the accumulator into the seed area. So X=RND(-1): PRINT RND(1): always prints the same value, and is the start of a repeatable sequence. RND(0) loads the floating-point accumulator from the VIA timers, two of which change at the same frequency as the chip (1 MHz). RND(0) therefore is not repeatable, and makes a good seed value. However, it may not be suitable for repetitive programming: try it in the random walk. Also BASIC 1 doesn't work correctly with RND(0). RND(-TI) therefore is a good function to use when a non-repeatable sequence is aimed at.

[2] A random number in the range A-B, excluding the exact end limits, is generated by: A + RND(1)*(B-A). A special case is 1 + RND(1)*2 which generates random numbers from -1 to +1. For integers from A% - B%, A% + INT(RND(1)*(B-A+1))covers the range, including both limits.

ROM entry points:BASIC1:$DF45 (57157)   BASIC2:$DF7F (57215)   BASIC4:$D229 (53801)

RUN
BASIC system command

PURPOSE: Executes a BASIC program in memory either from the beginning or from any linenumber. Previous values of variables are all lost on RUNning.

Syntax: RUN [linenumber]. The linenumber is optional. See also note [1].
If a line of the specified linenumber doesn't exist, ?UNDEF'D STATEMENT ERROR is printed and nothing more happens.

Modes: Direct and program modes are both valid.

Examples: RUN   :REM CLEAR VARIABLES AND RUN
           RUN 1000 :REM CLEAR VARIABLES AND RUN FROM LINE 1000

These direct mode commands, as typed at the keyboard, execute BASIC.

```
5000 INPUT "RETURN TO START";YN$: REM APPEARS AS 'RETURN TO START?'
5010 IF LEFT$(YN$,1)="Y" THEN RUN
10000 LOAD "NEXT TAPE PROGRAM": REM RUN IS IMPLICIT IN THIS
```

RUN may be called within a program; all variables and arrays are cleared, so this is useful if restarting BASIC from scratch. Line 10000, which uses LOAD from within a program, implicitly RUNs the new program too.

Notes: [1] When RUN is not followed by a colon or end-of-line, it is presumed to be followed by a line number which is evaluated by a part of the GOTO routine. The linenumber is therefore not completely validated (it need not be an arithmetic expression). Consequently, RUN X and RUN "PRG" are both equivalent to RUN 0. And RUN 25QQ is equivalent to RUN 25.

[2] RUN does not load and run, like (say) Apple. The shift-stop key has this function. In BASICs 1 and 2, this key inserts the string "LOAD[RETURN]RUN[RETURN]" into the keyboard buffer, which causes the usual tape loading sequence to be activated, starting with the request to 'PRESS PLAY ON TAPE#'. BASIC 4 uses the string "dL"*[RETURN]run[RETURN]" which loads and runs the first disk program.

[3] Not all RUNs share Microsoft's conflation of CLR with RUN. Some Sharp BASICs, when RUN, retain their old variables, so that CLR:RUN would be their equivalent of Microsoft's RUN command. Conversely, to run Microsoft BASIC without resetting all the variables requires GOTO linenumber.

[4] If location 1024 (and end of line bytes generally) hold some non zero byte, RUN will stop with a ?SYNTAX ERROR.

[5] If the end-of-program pointers are wrong, typically through loading one program from within another, RUN, either implicitly on loading or explicitly, may corrupt the program as soon as variables are given values. See OLD.

Abbreviated entry: rU   Token: $8A (138)

Operation: RUN alone sets GETCHR's pointer to the start of BASIC-1, then drops into CLR, which erases data, resets the DATA pointer, aborts open files, and resets the stack; it saves the top return address on the stack, which points to the RUN routine itself. RUN linenumber also CLRs, but without resetting GETCHR; then finds the line, and enters the RUN routine as before.

Programs are executed by a loop which performs single statements. The loop has this structure: (i) Test stop key, (ii) Store CONT pointer, (iii) Test for zero byte: if found, either end the program, or update the stored current linenumber and CHRGET, (iv) Get the current character, (v) Execute one statement, (vi) Start over at the beginning of the loop.

The routine can be rewritten by a programmer, excluding, for example, the testing for the stop key which is otherwise performed before each statement. Some timesaving is possible in this way. A few fast RUN programs are on sale.

ROM entry points:
```
RUN KEYWORD:BASIC1:$C775 (51061) BASIC2:$C785 (51077) BASIC4:$B808 (47112)
EXECUTION:  BASIC1:$C6B5 (50869) BASIC2:$C6C4 (50884) BASIC4:$B74A (46922)
```

SAVE

BASIC system command

PURPOSE: Writes a consecutive block of RAM to an output device, usually disk or tape. Normally this is a BASIC program, which is saved with a name for easy retrieval, although a name is optional with tape. The converse process to SAVE is LOAD.

Syntax: Identical to LOAD, including the difference between tape and disk SAVE, where a name is compulsory with disks. Unlike LOAD, the secondary address has a purpose: a tape program, saved with secondary address 2, writes an end-of-tape block so the cassette won't read past it. If the device number is 0 or 3 (keyboard or screen) ?DEVICE NOT PRESENT, rather oddly, appears. If a file of the same name exists on disk, ?FILE EXISTS ERROR will result.

Modes: Direct and program modes are both valid. When using tape, SAVE ["NAME"] is followed in both modes by PRESS PLAY AND RECORD ON TAPE#1 or 2. WRITING [NAME] also appears in direct mode only. (Unless the cassette was already running).

Examples:
```
SAVE               : REM SAVES THE BASIC IN MEMORY ON TAPE 1 (NO NAME)
SAVE "PROG006",2,2: REM SAVE BASIC AS 'PROG006' ON CASSETTE #2 WITH EOT

SAVE "",8          : REM NO NAME ... GIVES ?SYNTAX ERROR
SAVE "0:PROG=5",8  : REM SAVE 'PROG=5' ON DRIVE 0
SAVE "@0:PROG",8   : REM SAVE-WITH-REPLACE 'PROG' ONTO DRIVE 1*
SAVE CHR$(8)+"TEST RATE"+CHR$(146),1 :REM NAME APPEARS IN REVERSE
12000 SAVE "1:TEST"+TI$,8: REM NAME SAVED WITH UNIQUE TIME ATTACHED
```

These CBM tape and disk examples are (I hope) reasonably easy to follow. The tape examples show a full default (equivalent to SAVE "",1,0) and an example which writes end-of-tape after saving the program. The disk examples show the slightly more complex syntax needed, including the optional disk drive number and the mandatory device number 8. The string is also mandatory. The third disk example shows the 'save-with-replace' variation of disk save, which avoids the ?file exists error. This form of SAVE is however suspected to contain a bug; use it at your own risk. The final examples are intended to emphasize the fact that the string parameter is computed: the first example has its name saved in reverse text, the second is a program-mode SAVE which may be used to store successive versions of BASIC during development of a program; the time parameter shows when SAVE occurred.

Notes: [1] .S "NAME",01,027A,0300 and .S "1:NAME",08,027A,0300 are cassette #1 and CBM disk drive 1 versions of machine-code saves from the monitor. These use almost exactly the same routine as SAVE and in fact the same result can be achieved within BASIC. Compu/think uses $S,1,"NAME","027A","0300". With CBM BASIC, this routine is necessary:
```
SYS (62526) "0:HELLO",8: REM GETS THE PARAMETERS FOR NAME & DEVICE
POKE 251,LO-INT(LO/256)*256: POKE 252,LO/256: REM LOW ADDRESS IN (FB)
POKE 201,HI-INT(HI/256)*256: POKE 202,HI/256: REM HIGH ADDRESS IN (C9)
SYS 63140 : REM ENTER 'SAVE' SLIGHTLY LATER THAN USUAL.
```
This version is BASIC 2: for BASIC 4, substitute 62589 and 63203 for the SYS addresses. Remember to make the end address a byte longer than it should be: CBM's save excludes the final byte. BASIC 1 needs 62515 and 63153 as SYS addresses, and (F7) and (E5) for its low and high addresses.

Chapter 13 discusses other modifications of SAVE and LOAD.

[2] There is no readback check with tape. If 'Play' is pressed, but not 'Record', SAVE appears to operate correctly, but in fact nothing is written.

[3] BASIC 4 and BASIC 2 with 'Disk-o-Pro' have DSAVE too (q.v.)

---

*There seems to be no definitive statement available on the bugs in SAVE with replace, which saves to disk and erases the previous file of the same name, using the extra parameter '@'. Some commercial software does use it; some people swear by it, others swear at it! SCRATCH then SAVE is safest.

Abbreviated entry: sA

Token: $94 (148)

Operation: The outline that follows explains how SAVE works. BASIC 1 is similar to this schema, but its detailed arrangement, and its working storage areas, differ.

GET PARAMETERS FROM BASIC. ST is set to zero. Locations $D1, $D3, $D4, and ($DA) hold string length, secondary address, device number, and, if the string is not null, a pointer to the start of the string in BASIC. Where these parameters aren't specified, string length defaults to 0, secondary address to 0, and device number to 1.

MOVE BASIC START AND END POINTERS TO (FB) AND (C9). BASIC's pointers in ($28) and ($2A) usually hold $0401 and some higher value; the contents of RAM will be SAVEd between these two locations. The monitor's save with .S carries out the identical functions to these two routines, then enters SAVE at the next point:

DEVICE NUMBER CHECK. A device number of zero or three generates a ?DEVICE NOT PRESENT ERROR. SAVE "HELLO",,2 for instance does this. (These devices - in case you've forgotten - are the keyboard and screen).

SEPARATION INTO CASSETTE AND IEEE PROCESSING. Devices numbered 1 and 2 are cassettes, and are processed by a separate routine from IEEE devices 4-15.

IEEE PROCESSING.

SECONDARY ADDRESS/ NAME CHECK. Secondary address is set to #$61, equivalent to 1, to enable writing to the disk directory. A string parameter of zero length is rejected with ?SYNTAX ERROR. (This provides incomplete validation, because a string "1:" or ":" may still be sent).

WRITE NAME AND START ADDRESS TO IEEE BUS. Firstly, the string is sent character by character down the bus, after handshaking has been established. LISTEN plus the secondary address (overwritten so that it is always 1) are sent. (FB) is moved to (C7); this address is used to load, compare and increment from now on. (C7) points to the current RAM location being sent, (C9) holds the final location which is not sent. The low and high bytes of the start address, C7 and C8, are sent on the IEEE. (Then, when LOAD reverses this process, it knows which RAM address to store the bytes from).

LOOP, SENDING SINGLE CHARACTERS ALONG IEEE BUS. The sequence of activities here is: (i) Compare address (C7) with (C9); if they are now equal, exit without sending the final character. (ii) Load the accumulator with the byte and send it; (iii) test the Stop key; (iv) increment the address in (C7); (v) continue with loop - provided that (C7) did not increase from $FFFF to $0000.

EXIT. Finally, LISTEN, the secondary address, and UNLISTEN are sent.

CASSETTE TAPE PROCESSING.

PREPARE TO WRITE TO TAPE. This sets (D6) to $027A or $033A depending on the device number, prints PRESS PLAY AND RECORD ON TAPE# 1 or 2, and, if in direct mode, WRITING plus the optional program name, when a key on the cassette is detected down.

TAPE WRITE. The accumulator is loaded with #1 and the ROM routine to write a block (a 'header') called. #1 denotes a program. Then the tape write routine is called, and finally, if the secondary address was 2, another 'header' is written, this time with the type character #5, indicating an end-of-tape block. See Chapter 8 for more detail on the actual writing to tape.

ROM entry points: SAVE is a 'kernel' command; its jump address is $FFD8.

BASIC1:$F69E (63134)   BASIC2:$F69E (63134)   BASIC4:$F6DD (63197)

SET

BASIC graphics command unavailable directly in CBM BASIC

PURPOSE: Plots a 'point' (in fact a small square) on the screen at a position determined by two parameters, which represent horizontal and vertical distances or X and Y coordinates from some starting point.

Versions: Both BASIC and machine code versions of this routine exist for the CBM. BASIC usually is too slow. Some versions include straight-line plotting algorithms so that lines can be drawn without further calculation. The resolution is 80 by 50 for 40 column machines, 160 by 50 for 80 columns; this is useful, but not 'high resolution'. (Many other machines have rather similar displays: the TRS-80 has 128 by 48, Sharp MZ80K 80 by 50, Sinclair ZX81 64 by 48). Higher resolution in one direction can be achieved, for bar charts and similar diagrams, very simply by plotting solid blocks and adding a final part of a block, which has a resolution of one part in eight. And approximations to sloping lines can be made with line segments, so a curve will appear as a series of steps. (There is for example a ROM chip called 'PicChip' which does this from BASIC). For more detail on this, see Chapter 9. The SET here is designed to plot double-density squares only with a fast machine-code algorithm. When called from BASIC it is still slow. This is because of the computing time which BASIC takes. However, it is perfectly usable.*

Algorithm: 'Micro' had an early version of this. Other publications, such as the first issue of 'Printout', followed. The basis of the method is as follows:
Suppose we use the convention that horizontal (X) coordinates start at the bottom left of the screen with 0, and vertical (Y) cordinates also start at the bottom left, with 0, so 0-79 or 159 is the range of X values, and 0-49 is the range of Y values. Taking a concrete illustration, suppose we wish to plot a white square at (1,1). The complicating factor is that there will be squares already plotted in the vicinity of (1,1), and since the character generating ROM only allows one entire character to be changed, the plot has to take account of the character already present. In our example, this
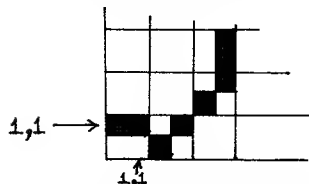
is the character in the bottom left of the screen. We can do this with a look-up table which arranges the screen graphics characters (16 of them are relevant to our purpose) in order determined by whichever quadrants are turned on: we assign an arbitrary bit position to each quadrant. The diagram shows how the quadrants are numbered, and the corresponding order, from lowest to highest, which the graphics characters take.

All we have to do is find the screen ASCII value, find its position in the table, and ORA with 1,2,4 or 8; the result, looked up in the table, gives the new character to be poked to the screen. Overleaf is a 40-column, and an 80-column, routine to do just this. Its operation is explained elsewhere. To use it, POKE 0,X coordinate: POKE 1, Y coordinate: SYS 634 will plot a square. POKE 729,0 for a black square; any non-zero value gives a white square. The zero-page locations used are these:

    $00=X coordinate; overwritten by X coordinate of screen, 0-39 or 0-79.
    $01=Y coordinate; replaced by screen pointer's low byte.
    $02=screen pointer's high byte.
    $94=remainder after halving both X and Y coordinates. The conflated remainders are overwritten by 1,2,4 or 8.

Note that $94 is used by the NMI line; if you're using non-maskable interrupts you'll need another zero-page (or other) location. BASIC 1 can substitute $59 near the end of the input buffer.

---

*SET is not optimised for speed: a lookup table of screen-line start positions, for example, could improve the running time. But with BASIC, the difference isn't great.

This monitor listing is appropriate for a *40 column machine* running either *BASIC 2 or BASIC 4.* See below for (i) 80 column modifications, (ii) BASIC 1 modifications, and (iii) relocation.

Incidentally, it is worth mentioning that the cheapest method of increasing the dot density - if you know someone with an EPROM blower - is to use a character generating ROM containing the entire 256 2 by 4 characters. In association with a hardware device to switch between ROMs, this gives with an 80-column CBM a resolution of 160 by 100, about 1/3 of Apple's dot density. Since the 80-column characters are somewhat elongated upwards, this ought to improve the appearance too. The graphics character set does include a 4 by 4 character, CHR$(222) in one of the graphics modes. But it is easy to see that its entire character set equivalent can't be displayed, because 2^16=65536. 8 on/off alternatives is the maximum obtainable. See Chapter 9 for further explanation.

```
     PC   IRQ  SR AC XR YR SP
.;   B780 E455 2C 34 3A 9D FA

.:   027A A9 00 85 94 A9 (20) 85 02
.:   0282 A5 00 C9 (50) B0 38 A5 01
.:   028A C9 32 B0 32 A9 32 E5 01
.:   0292 46 00 26 94 6A 26 94 85
.:   029A 01 0A 0A 65 01 0A 0A 26
.:   02A2 02 0A 26 02 (EA EA EA) 85
.:   02AA 01 A6 94 BD (DD 02) 85 94
.:   02B2 A4 00 B1 01 A2 0F DD (E1 02)
.:   02BA 02 F0 04 CA 10 F8 60 AD
.:   02C2 (DC 02) F0 06 8A 05 94 AA
.:   02CA D0 08 8A 49 FF 05 94 49
.:   02D2 FF AA BD (E1 02) A4 00 91
.:   02DA 01 60 (01) (01 02 04 08) 20
.:   02E2 7E 7B 61 7C E2 FF EC 6C
.:   02EA 7F 62 FC E1 FB FE A0
```

B/w Switch,
Quadrants,
& Graphics.

(i) 80-COLUMN MODIFICATIONS:

Replace the two indicated bytes by #$10 and #$A0.

And replace the three NOPs by 0A 26 02.

(ii) BASIC 1 MODIFICATIONS:

Replace all #$94s with #$59s.

(iii) RELOCATION:

The three double-byte pointers marked on the listing point to the tables at the end of the routine. The fourth pointer loads the character which determines the black/white switch. Each must be changed on moving this code.

This short demonstration program will plot a Lissajou figure. Note that non-graphics characters are ignored - i.e. no plot takes place there.

```
1000 INPUT "TWO NUMBERS, E.G. 4,7";A,B: FOR J=0 TO 9E9 STEP . 2
1010 X= (1+SIN(A*J)) * 40: Y= (1+COS(B*J))*25: REM OR 80 FOR X
1020 POKE 0,X: POKE 1,Y: SYS 634: REM OR OTHER SYS VALUE
1030 NEXT
```
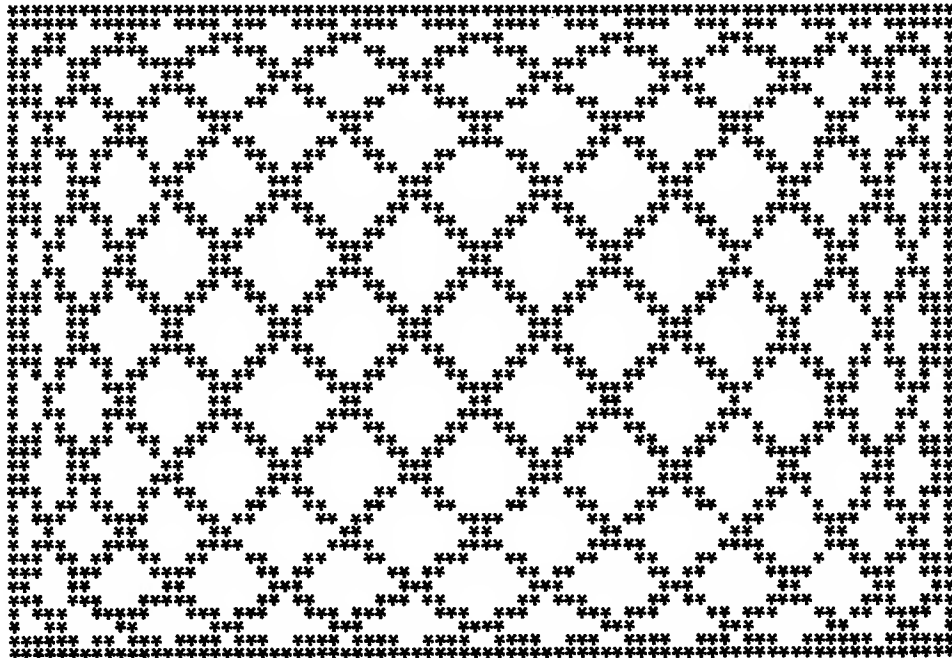
SGN

BASIC arithmetic function

PURPOSE: Computes the sign of an arithmetic expression. SGN returns the value −1 if the expression is negative, 0 if zero, and +1 if positive.*

Syntax: SGN(arithmetic expression). The arithmetic expression must be valid and must evaluate to an acceptable value.

Modes: Direct and program modes are both valid.

Examples:

```
10 IF SGN(X) > 0 THEN PRINT X; "IS POSITIVE"
20 IF ABS(SGN(X))<>0THEN PRINT X; "IS NON ZERO"

FOR J=-100 TO 100: PRINT J, SGN(J), SGN(J)*J, SGN(J)*ABS(J): NEXT

DEF FN A(ZZ) = INT(ZZ*100 + SGN(ZZ)*.5)
ON SGN (X) + 2 GOSUB 100,120,140
```

SGN is one of the less exciting BASIC functions. It is closely related to ABS ,<,=, and >, in the sense that these functions and operators can, when permuted, produce identical results. SGN(X-Y) for instance returns zero if X=Y, +1 if X>Y, and −1 if X<Y. The two first examples show how SGN may be used. Although the logic is correct, the function is entirely redundant. Usually therefore this function needs to make use of the fact that explicit values of 0 or ±1 are returned if it is not to be superfluous. The third example is a direct mode loop showing some possibilities in this direction. The separation of sign from magnitude is illustrated.

Of the remaining examples, one is a function definition which I've quoted from someone else's program. It is intended as a rounding routine, in which a sum of money, either positive or negative, is converted into the same amount in cents/pence, but rounded to the nearest cent/penny. Again it shows how the value of ±1 may be used; unfortunately, in the case of negative numbers, it rounds down too far. A sign that the programmer was trying to be over-clever? The other example, which is quoted under ON, converts −1,0, and 1 into 1,2, and 3, the range required by ON ... GOSUB in our example. This is equivalent to the FORTRAN construction

```
        IF (X) 100,120,140
```

and is sometimes useful when converting engineering-style programs for such purposes as pipe diameter calculations to run on microcomputers.

Abbreviated entry: sG

Token: $B4 (180)

Operation: Firstly, a short subroutine is called which loads the accumulator with 0, 1, or $FF depending on the sign of the contents of accumulator #1. This is determined firstly by the exponent: a zero exponent conventionally denotes a zero result in the accumulator. If this is non-zero, the high bit of the sign byte is tested; if set, the number is negative. The accumulator is loaded according to these tests. Just after SGN is a routine which converts integers to floating-point values, and this is simply dropped into, after the low byte has −1/0/1 put in it, and the high byte 0. The exponent is set at #$88 since the maximum is 255. (Other entries from the main fixed-to-floating point routines load #$90 into the exponent).

ROM entry points:

```
BASIC 1: $DB0B (56075)
BASIC 2: $DB45 (56133)
BASIC 4: $CD6F (52591)
```

---

*Aside from the identical pronunciation, this function has little in common with SIN.

SIN
BASIC arithmetic function

PURPOSE: Evaluates the sine of the argument, which is assumed to be in radians. The sine is a ratio which is constant for any angle; the diagram illustrates this ratio.

Syntax: SIN(arithmetic expression). The expression must be syntactically correct and within the range accepted by the floating point logic (±1.7 E38 approx).

Modes: Direct and program modes are both valid.

Examples:
    PRINT SIN(1)   prints sine of 1 radian = .842 approx.
    PRINT SIN(360 * [PI]/180)   prints sine of 360° = 0.
    10 Q=180/[PI]: FOR J=0 TO 90: PRINT SIN(J*Q): NEXT
    120 X=A+SIN(A)/2: Y=A+ SIN(A)*3/2: REM TROCHOID

     The first examples show SIN used in direct mode calculations. The third example is a loop which prints out the value of sine, as calculated by the CBM, for angles from zero to ninety degrees. Example four calculates two coordinates which depend on a single parameter A. Innumerable formulas of this type exist.

Notes: [1] The diagrams show the sine in terms of the sides of a right-angled triangle, and the concept of a radian. 'O' and 'H' by convention represent the opposite side to the angle and the hypotenuse respectively.
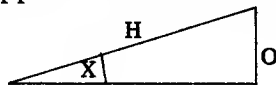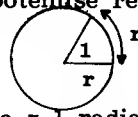
SIN(X) = O/H   Angle = 1 radian

     [2] Accuracy is not greatly affected by the size of the angle: this function operates by dividing the argument by 2*pi and taking the remainder, so there is no series approximation error related to the size of the argument, only the error caused by the limited precision to which the argument is held.

     [3] See the appendices for the inverse function ARCSIN.

Abbreviated entry: sI

Token: $BF (191)

Operation: The argument is evaluated and stored in both floating point accumulators. Accumulator #1 is divided by 2*pi (6.283..), and this result moved to accumulator #2. The integer value of accumulator #1 is generated, and the difference between accumulator #2 and accumulator #1 stored in accumulator #1. This completes the processing of the argument. Its sign byte is pushed onto the stack, and on exit recovered; if negative, the sign of the result is made negative. The calculation has five constants and an additive constant; powers up to and including the fifth are therefore used.

ROM entry points:

    BASIC 1: $DFA5 (57253)
    BASIC 2: $DFDF (57311)
    BASIC 4: $D289 (53897)

SORT

System command unavailable directly in CBM BASIC

PURPOSE: Arranges data in alphabetic, alphanumeric, or ASCII order. ASCII order is the most common, since it corresponds directly with the way data is stored, but any other sort criteria may be used. Generally, computer-sorted data will not always correspond exactly with data sorted by manual means, because some of the underlying conventions may differ: for example, 'Mc' or 'Mac' unless specifically checked will not precede all other 'M's.

Versions: Sorting (of cheques &c) is in widespread use on large computers, often with merge routines, by which daily transactions update master files. This is a ponderous technique which is rather rare on micros. IBM's 8100 machines are fully equipped with the necessary commands of this type, but this is unusual. Some multi-key sorts have been written for CBM hardware including a Compu/-think disk version. There is a sort-merge in Nick Hampshire's 'Library of PET Subroutines'. The easiest sorts to write are for 1-dimensional arrays, and BASIC versions embodying all the common algorithms exist. (Knuth's many-volume work is a source of algorithms). Some machine-code hardware sorts are available. M Lake's bubble sort (Practical Computing, Apr.'80) is in machine-code; CCN, Oct.'81, has a Shell-Metzner sort - for integers only.

In any but the most trivial applications, sorting tends to come to grief on the twin prongs of space and time. If the whole of a batch of data cannot be fitted into RAM, subsets must be sorted individually, and the resulting files merged. The necessary disk (or tape) manipulations are likely to be slow. In practice, this may be tolerable, since long processing times may nevertheless compare favourably with the time needed to type data in. Section 4.11.4 has more on the subject, including descriptions of the merits of the sorts presented here, of which there are seven, including one in machine-code. A graph (on the final page of this section) shows the approximate range of timings to be expected.

Notes: [1] <u>ORDER</u>. Numerals are especially liable to be sorted into what appear to be strange sequences. String comparisons in most micro BASICs compare successive characters until either a string comes to an end, or one character differs from the other and the 'smaller' is found. So "49" is less than "5", and "5" is less than "51" in CBM BASIC. The strings 0 to 25, sorted like this, emerge: 0,1,10,11,12,13,14,15,16,17,18,19,2,20,21,22,23,24,25,3,4,5,6,7,8,9. If the sort deals with numbers only, they can be output in numeric order, but many sorts deal with string data because of its universal applicability.
[2] <u>SORT FIELDS</u>. It follows from note [1] that programming can often be simplified by careful choice of the way in which items to be sorted are arranged. For instance, a date held in the format DDMMYY may need three separate comparisons, of year, month, and day, But YYMMDD automatically sorts into the correct order, because years are more significant than months, and months than days. Similarly, the fact that the comma has a lower ASCII value than any letter ensures that names, held with commas, sort correctly - "Williams,P.R." when sorted on Commodore's criteria comes before "Williamson,A.B.".

1. The Tournament Sort.

```
10 INPUT "SORT HOW MANY ITEMS";N: B=N-1: DIM N$(B),I(2*B)
20 FOR J=0 TO B: INPUT N$(J): NEXT: REM SETS UP DEMONSTRATION DATA
200 X=0: FOR J=0 TO B: I(J)=J: NEXT:REM INDEX ARRAY SET UP WITH 0,1,2,3,...
210 FOR J=0 TO 2*N-3 STEP 2: B=B+1: REM ORDERS INDEX ARRAY IN PAIRS
220 I(B)=I(J): IF N$(I(J+1))<N$(I(J)) THEN I(B)=I(J+1): NEXT
250 X=X-1:C=I(B): IF C<0 THEN END: REM SORT FINISHED
260 PRINT N$(C) " ";:          REM PRINT ONE SORTED ITEM OF DATA
270 I(C)=X:                    REM SORT LOOP IS HERE
280 J=2*INT(C/2): C=INT(C/2)+N: IF C>B GOTO 250
300 IF I(J)<0    THEN I(C)=I(J+1): GOTO 280
310 IF I(J+1)<0 THEN I(C)=I(J)   : GOTO 280
320 I(C)=I(J): IF N$(I(J+1))<N$(I(J)) THEN I(C)=I(J+1)
330 GOTO 280
```

2 & 3. The Exchange Sort and the Bubble Sort.

```
0 REM ####################
1 REM ### EXCHANGE SORT ###
2 REM ####################
3 REM
4 REM ## SORTS N STRINGS OF 21 CHARACTERS LENGTH; AND PRINTS TIME TAKEN.    ##
5 REM ##  RUN 0     ... EXCHANGE SORT                                       ##
6 REM ##  RUN 500   ... BUBBLE  SORT                                        ##
7 REM ##  GO TO 50  ... RE-SORT SAME DATA BY EXCHANGE SORT                  ##
8 REM ##  GO TO 550 ... RE-SORT SAME DATA BY BUBBLE SORT                    ##
9 REM
10 INPUT "NO. OF STRINGS";N
20 DIM A$(N)
21 REM
22 REM ########################################################################
23 REM ## FILL THE ARRAYS WITH FAIRLY LONG STRINGS; INCLUDING RANDOM LETTER  ##
24 REM ########################################################################
25 J = RND(-1) : REM NOTE: THIS SEEDS A CONSTANT RANDOM NUMBER
30 FOR J = 1 TO N: A$(J) = CHR$(65+RND(1)*26) +"***********++++++++++"
40 NEXT: REM THE SEEDED VALUE ENSURES IDENTICAL STRINGS IN THE BUBBLE SORT
50 T = TI : REM STORE CLOCK TIME
95 REM
96 REM          ############################
97 REM          ### PERFORM EXCHANGE SORT ###
98 REM          ############################
99 REM
100 FOR J = 1 TO N-1
110 FOR K = J+1 TO N
120 IF A$(J) > A$(K) THEN TEMP$ = A$(J): A$(J) = A$(K): A$(K) = TEMP$
130 NEXT K
140 NEXT J
145 REM
146 REM
190 PRINT(TI-T)/60 "SECS"
200 REM FOR J=1 TO N: ? A$(J);: NEXT: REM OPTIONAL PRINTOUT OF SORTED STRINGS
300 END
305 REM ########################
310 REM ## END OF EXCHANGE SORT ##
315 REM ########################
320 REM
500 REM ###################
501 REM ###  BUBBLE  SORT ###
502 REM ###################
503 REM
510 INPUT "NO. OF STRINGS";N
520 DIM A$(N)
525 J = RND(-1)
530 FOR J = 1 TO N: A$(J) = CHR$(65+ RND(1)*26 ) + "***********++++++++++"
540 NEXT
550 T = TI
595 REM
596 REM          ############################
597 REM          ###  PERFORM BUBBLE SORT ###
598 REM          ############################
599 REM
600 FOR J = N-1 TO 1 STEP -1: FIN=-1
610 FOR K = 1 TO J
620 IF A$(K) > A$(K+1) THEN FIN=0: TE$ = A$(K): A$(K) = A$(K+1): A$(K+1) = TE$
630 NEXT K: IF NOT FIN THEN NEXT J
645 REM
646 REM
690 PRINT (TI-T)/60 "SECS"
700 REM FOR J=1 TO N: ? A$(J);: NEXT: REM OPTIONAL PRINTOUT OF SORTED STRINGS
READY.
```

4 & 5. The Shell-Metzner Sort and Quicksort.

```
4 REM   ####################################################################
5 REM   ## SORTS STRINGS USING SHELL-METZNER SORT AND PRINTS TIME TAKEN  ##
6 REM   ##   RUN    ... PERFORMS SHELL-METZNER SORT                      ##
7 REM   ##   GO TO 50 ... RE-SORTS ARRAY A$()                            ##
8 REM   ####################################################################
9 REM
10 INPUT "NO. OF STRINGS";N
11 DIM A$(N)
12 J = RND(-1)
13 FOR J=1 TO N: A$(J) = CHR$(65+RND(1)*26) + "***********+++++++++++"
14 NEXT : REM  WE USE SAME STRINGS AS OTHER PROGRAMS TO TEST SORTING
50 T = TI :REM STORE CLOCK TIME
59000 REM
59001 REM        ##########################################
59002 REM        ###     START OF SHELL-METZNER SORT    ###
59003 REM        ##########################################
59004 REM
59005 M = N
59010 M = INT(M/2): IF M = 0 THEN 59200: REM SORT COMPLETED
59020 J = 1: K = N - M
59030 I = J
59040 L = I + M
59050 IF A$(I)>A$(L) THEN TE$=A$(I): A$(I)=A$(L): A$(L)=TE$: I=I-M: IF I>0 THEN 59040
59060 J = J + 1: IF J > K GOTO 59010
59070 GOTO 59030
59200 PRINT (TI-T) / 60 "SECS"
59210 REM FOR J = 1 TO N: ? A$(J);: NEXT:REM OPTIONAL PRINTOUT OF SORTED STRING
59220 END


4 REM   ####################################################################
5 REM   ##      SORTS STRINGS USING 'QUICKSORT' AND PRINTS TIME TAKEN    ##
6 REM   ## RUN    ... PERFORMS QUICKSORT                                 ##
7 REM   ##   GO TO 50 ... RE-SORTS ARRAY A$()                            ##
8 REM   ####################################################################
9 REM
10 INPUT "NO. OF STRINGS";N
11 DIM A$(N)
12 J = RND(-1)
13 FOR J=1 TO N: A$(J) = CHR$(65+RND(1)*26) + "***********+++++++++++"
14 NEXT : REM  WE USE SAME STRINGS AS OTHER PROGRAMS TO TEST SORTING
30 REM
31 REM      ############################
32 REM      ### PERFORM  'QUICKSORT' ###
33 REM      ############################
34 REM
40 DIM ST ( (LOG(N)/LOG(2) + 4) ,1 ): REM THIS ARRAY HOLDS LEFT AND RIGHT STACK
50 T = TI :REM STORE CLOCK TIME
100 S = 1: ST(1,0) = 1: ST(1,1) = N
110 L = ST(S,0): R = ST(S,1): S = S - 1
120 J = L: K = R: X$ = A$( (L + R)/2 ): REM PIVOT VALUE TAKEN TO BE MIDWAY
124 REM
125 REM NOTE THAT LINES 130 AND 140 ARE VARIATIONS OF EACH OTHER; ACTUAL SPEED
126 REM OF RUNNING DEPENDS ON LENGTH OF PROGRAM AND NUMBER OF VARIABLES IN IT,
127 REM SO SELECT THE APPROPRIATE FORMAT FOR BOTH LINES EXPERIMENTALLY.
128 REM  MANY SIMILAR ALTERATIONS MAY BE TRIED WITH THE PROGRAM.
129 REM
130 IF A$(J) < X$ THEN J = J + 1: GOTO 130
140 FOR V = 1 TO 1E6: IF A$(K) > X$ THEN K = K - 1: NEXT
150 IF J = K THEN J= J + 1: K = K - 1 :GOTO 130
160 IF J < K THEN TEMP$ = A$(J): A$(J)=A$(K):A$(K)=TEMP$: J=J+1:K=K-1 :GOTO 130
170 IF J < R THEN S = S + 1: ST(S,0) = J: ST(S,1) = R
180 R = K
190 IF L < R THEN 120
200 IF S > 0 THEN 110
240 REM
241 REM      ##########################
242 REM      ### END OF 'QUICKSORT' ###
243 REM      ##########################
244 REM
250 PRINT (TI-T) / 60 "SECS"
260 REM FOR J = 1 TO N: ? A$(J);: NEXT:REM OPTIONAL PRINTOUT OF SORTED STRINGS
300 END
```

6. 'Scatter Sort'.

```
0 REM            ######################
1 REM            ### 'SCATTER SORT' ###
2 REM            ######################
3 REM
4 REM  #################################################################
5 REM  # VERY RAPID SORT USING A SUBSIDIARY ARRAY FOR A PRELIMINARY ROUGH SORT.#
6 REM  #ASSUMES FAIRLY EVEN DISTRIBUTION OF STRINGS' INITIALS FROM A THROUGH Z.#
7 REM  #  'RUN' RUNS SCATTER SORT; 'GOTO 50' RE-SORTS; 'GOTO 230' BUBBLE SORTS.#
8 REM  #################################################################
9 REM
10 INPUT "NO. OF STRINGS";N
12 INPUT "APPROX. AVERAGE LENGTH";LE
14 LE = LE + 3: REM HOLDS LEENGTH OF STRING AND ITS POINTER
16 J = RND(-1): REM SET SEED
20 DIM A$(N)
30 FOR J = 1 TO N: A$(J) = CHR$(65 + RND(1) * 26) + "***********+++++++++"
40 NEXT
50 T = TI : REM STORE CLOCK TIME
90 REM
91 REM  #################################################################
92 REM  # EXAMPLE ASSUMES (ASSUMPTION CAN BE CHANGED) THAT EVERY STRING BEGINS  #
93 REM  # WITH AN ALPHABETIC CHARACTER: HENCE VALUES IN LINE 100 FOR LOWER AND  #
94 REM  # UPPER LIMITS. THE SIZE OF A SUBSIDIARY ARRAY IS DETERMINED IN 110-140 #
95 REM  # AND IS LARGE ENOUGH TO ENSURE A REASONABLE ROUGH SORT:-               #
96 REM  #################################################################
97 REM
100 L = 64: U = 91: Z=0: K=0 :FI=0:TE$="": PL = 0: PH = 0
101 REM PREVIOUS LINE SETS UP ALL VARIABLES BELOW THE ARRAY, SO IT'LL STAY PUT
110 B = FRE (0)/(LE * N ):REM B=NUMBER OF DUPLICATE ARRAYS WHICH COULD FIT MEM.
120 IF B < 2 THEN PRINT "INSUFFICIENT MEMORY": END
130 IF B > 4 THEN B = 4
135 PL = PEEK(46) : PH = PEEK(47): REM STORE CURRENT END OF ARRAY POINTER
140 DIM B$( B*N + 30 ) : REM SUBSIDIARY ARRAY; MOST OF IT WILL REMAIN NULL.
142 REM  #################################################################
143 REM  ## CALCULATE APPROXIMATE POSITION IN B$() TO WHICH EACH ELEMENT FROM   #
144 REM  ## A$() SHOULD PROPORTIONALLY BELONG; FILL IN SOME OF B$() WITH THESE  #
145 REM  ## VALUES, SO THAT B$() IS SPARSELY FILLED WITH ROUGHLY SORTED STRINGS#
146 REM  #################################################################
147 REM
150 Z = B * N / ( U - L ): REM Z IS A SCALE FACTOR COMPUTING THE LIKELY PLACE..
160 FOR J = 1 TO N: K = ( ASC(A$(J)) - L ) * Z: REM ..OF THE STRING IN B$().
170 IF B$(K) = "" THEN B$(K) = A$(J): NEXT: GOTO 190
180 K = K + 1: GOTO 170
182 REM
183 REM  #################################################################
184 REM  ## PUT B$() BACK INTO A$(), IGNORING NULL STRINGS; SO THAT A$() NOW ##
185 REM  ## CONTAINS ITS OWN ELEMENTS AGAIN, BUT ROUGHLY SORTED:-            ##
186 REM  #################################################################
187 REM
190 J = 1: FOR K = 1 TO B*N + 29: IF B$(K) = "" THEN NEXT: GOTO 210
200 A$(J) = B$(K): J = J + 1: NEXT
210 REM
211 REM  #################################################################
212 REM  # RESET OLD POINTERS TO END OF ARRAY - ERASING SUBSIDIARY ARRAY:- #
213 REM  #################################################################
220 POKE 46,PL: POKE 47,PH: REM THE SUBSIDIARY ARRAY B$() NO LONGER EXISTS.
222 REM
223 REM  #################################################################
224 REM  ## FINALLY, USE THE BUBBLE SORT TO COMPLETE THE SORTING PROCESS: ##
225 REM  #################################################################
226 REM
230 FOR J = N-1 TO 1 STEP -1: FIN = -1
232 FOR K = 1 TO J
234 IF A$(K) > A$(K+1) THEN FIN=0: TE$=A$(K): A$(K) = A$(K+1): A$(K+1) = TE$
236 NEXT: IF NOT FIN THEN NEXT
272 REM
273 REM            #############################
274 REM            ### END OF SCATTER SORT ###
275 REM            #############################
276 REM
280 PRINT (TI-T) / 60 "SECS"
290 REM FOR J = 1 TO N: ? A$(J);:NEXT :REM OPTIONAL PRINT OF SORTED STRINGS
```

7. Machine-code Bubble Sort.

634	$027A	20	C2	00	85	10	A9	80	85		,:	7F02	20	70	00	85	5E	A9	80	85

Line	Addr																	
634	$027A	20 C2 00 85 10 A9 80 85
642	$0282	11 20 C2 00 FO 07 09 80
650	$028A	85 11 20 C2 00 A5 7E 85
658	$0292	12 A5 7F 85 13 A0 00 A5
666	$029A	10 D1 12 DO 07 C8 A5 11
674	$02A2	D1 12 FO 16 18 A0 02 B1
682	$02AA	12 65 12 85 1B C8 B1 12
690	$02B2	65 13 85 13 A5 1B 85 12
698	$02BA	90 DB A0 05 B1 12 85 15
706	$02C2	C8 B1 12 85 14 DO 02 C6
714	$02CA	15 C6 14 18 A5 12 69 07
722	$02D2	85 12 A5 13 69 00 85 13
730	$02DA	A5 14 DO 02 C6 15 C6 14
738	$02E2	DO 04 A5 15 FO 12 85 18
746	$02EA	A2 00 86 16 86 17 A5 12
754	$02F2	85 19 A5 13 85 1A FO EO
762	$02FA	FO 72 18 A5 19 69 03 85
770	$0302	19 A5 1A 69 00 85 1A E6
778	$030A	16 DO 02 E6 17 A0 02 B1
786	$0312	19 99 BO 00 88 10 F8 A0
794	$031A	05 B1 19 99 B7 00 88 CO
802	$0322	02 DO F6 AA 38 E5 BO 90
810	$032A	02 A6 BO A0 FF E8 C8 CA
818	$0332	DO 08 A5 BA C5 BO 90 0A
826	$033A	BO 22 B1 BB D1 B1 FO EE
834	$0342	BO 1A A0 02 B9 BA 00 91
842	$034A	19 88 10 F8 A0 05 B9 AD
850	$0352	00 91 19 88 CO 02 DO F6
858	$035A	A9 00 85 18 A5 14 C5 16
866	$0362	DO 98 A5 15 C5 17 DO 92
874	$036A	A5 18 FO 8A 60

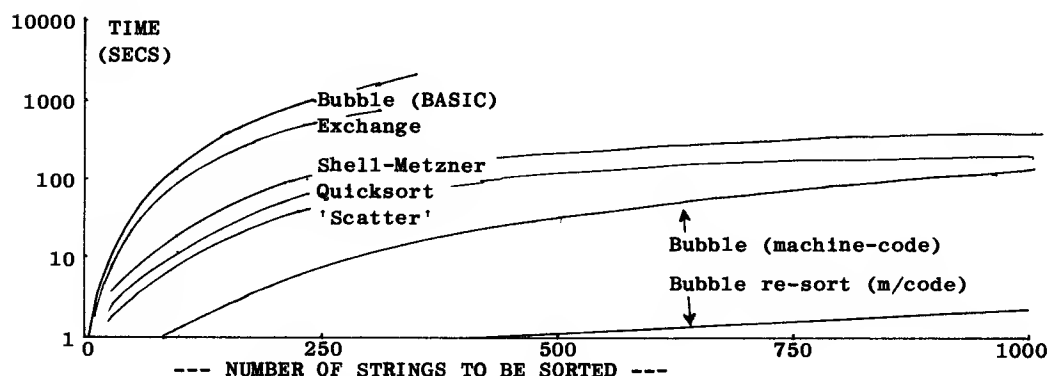| Addr		
7F02	20 70 00 85 5E A9 80 85
7F0A	5F 20 70 00 FO 07 09 80
7F12	85 5F 20 70 00 A5 2C 85
7F1A	60 A5 2D 85 61 A0 00 A5
7F22	5E D1 60 DO 07 C8 A5 5F
7F2A	D1 60 FO 16 18 A0 02 B1
7F32	60 65 60 85 69 C8 B1 60
7F3A	65 61 85 61 A5 69 85 60
7F42	90 DB A0 05 B1 60 85 63
7F4A	C8 B1 60 85 62 DO 02 C6
7F52	63 C6 62 18 A5 60 69 07
7F5A	85 60 A5 61 69 00 85 61
7F62	A5 62 DO 02 C6 63 C6 62
7F6A	DO 04 A5 63 FO 12 85 66
7F72	A2 00 86 64 86 65 A5 60
7F7A	85 67 A5 61 85 68 FO EO
7F82	FO 72 18 A5 67 69 03 85
7F8A	67 A5 68 69 00 85 68 E6
7F92	64 DO 02 E6 65 A0 02 B1
7F9A	67 99 6A 00 88 10 F8 A0
7FA2	05 B1 67 99 6A 00 88 CO
7FAA	02 DO F6 AA 38 E5 6A 90
7FB2	02 A6 6A A0 FF E8 C8 CA
7FBA	DO 08 A5 6D C5 6A 90 0A
7FC2	BO 22 B1 6E D1 6B FO EE
7FCA	10 1A A0 02 B9 6D 00 91
7FD2	67 88 10 F8 A0 05 B9 67
7FDA	00 91 67 88 CO 02 DO F6
7FE2	A9 00 85 66 A5 62 C5 64
7FEA	DO 98 A5 63 C5 65 DO 92
7FF2	A5 66 FO 8A 60

BASIC 1 ('OLD ROM')   BASIC 2 ('UPGRADE ROM') AND BASIC 4

Both these routines are freely relocatable; the old ROM version has been put into cassette buffer #1; the BASIC 2 or 4 version is positioned in the top of 32K memory, where POKE 53,127:CLR will protect it from BASIC. The syntax needed to run the sort is shown by this demonstration program:-

```
0 INPUT "NUMBER OF STRINGS";N: DIM A$(N): REM ARRAY A$() IS SET UP
2 FOR J=1 TO N: A$(J)=STR$(INT(RND(1)*10000)): NEXT: REM FILL ARRAY
4 SYS 32514:A: REM THIS SORTS ARRAY A$(). BASIC 1 VERSION IS SYS 634.
6 FOR J=0 TO N: PRINT "STRING NUMBER" J " IS " A$(J): NEXT
```

These routines operate in program mode only. The string array must exist and be 1-dimensional. The sorted order is the same as BASIC; for example, null strings come first. The strings need not be sorted from their initial character: a sort key can be defined starting within a string. See elsewhere for details. Finally, note the syntax is SYS 634:A to sort A$(), and SYS 634:PQ to sort PQ$(), and so on.

TIME (SECS) vs NUMBER OF STRINGS TO BE SORTED — curves: Bubble (BASIC), Exchange, Shell-Metzner, Quicksort, 'Scatter', Bubble (machine-code), Bubble re-sort (m/code)

SPC(

BASIC format function

PURPOSE: Prints a number of spaces or cursor-right characters on the screen
or to a printer. The number depends on the parameter in brackets. This
instruction is normally only used within a PRINT statement.

Syntax: PRINT ... SPC(arithmetic function). The arithmetic function must take,
after rounding down, a value in the range 0-255. No spaces may appear
between SPC and (, except in BASIC 1. The interpreter will treat such a
construction as the array SP().

Modes: Direct and program modes are both valid.

Examples: 20 PRINT "[CLR]": FOR J=0TO20: PRINT "[SHIFT-&]" SPC(38) "[SHIFT-&]":
NEXT
30 FOR J=0 TO 19: PRINT SPC(J) "*" SPC(38-J*2) "*": NEXT

These examples show how SPC may be used within a loop to print certain
repetitive types of design on the screen. The first provides a border down
each side of the screen; the second a V-shaped pattern.

       765 PRINT SPC(10)"The PET can run while the disk is processing"

This program line illustrates the typical use of SPC in the straightforward
printing of literals. Since SPC(when tokenised occupies 1 byte, line 765
is six bytes shorter than PRINT "   The ...". This may or may
not be a worthwhile saving. And the appearance on listing may or may
not be improved by the function.

Notes: [1] SPC(and TAB(share a peculiarity concerned with printers. The
point is that SPC(and TAB(are processed in virtually identical ways by
PRINT, sharing the bulk of the same routine. In addition, the earliest
BASIC, presumably with screen PRINTing in mind, uses PET's cursor right
characters to generate 'spaces' in each of the commands. These characters
are not of course universal; BASIC 2 and 4 were modified so that spaces
(#$20 characters) are sometimes produced by SPC(and TAB(. In this way
other printers could be used with these functions without printing spurious
information. The following short program illustrates the difference, and
the location to poke if either command is giving trouble.* Its effect is clear
when the program is run: in one case (when the device is 0, i.e keyboard)
SPC(does not print 'spaces', but skips right, leaving the previous screen
contents as far as possible unchanged. But the poke, which is interpreted
as a change of device, causes exactly the same instruction to print spaces.

```
10 INPUT X
20 POKE 14,X
30 PRINT "[HOME]";: FOR J=0 TO 30: PRINT "X";: NEXT
40 PRINT "[HOME]**" TAB(10) "**" SPC(10) "**"
READY.
```

```
**XXXXXXXX**XXXXXXXXXX**XXXXXXX
**          **              **XXXXXXX
```

Top line when X=0; bottom line when X=1.

Abbreviated entry: sP (includes left parenthesis)

Token: $A6 (166)

Operation: See PRINT. If SPC(is found in a print statement, the expression in
parentheses is evaluated and validated as usual. (The right hand bracket
has its own check). The resulting single byte is held in the X register
and counts the characters as they are singly output.

ROM entry_points: BASIC1:$CA1B (51739)   BASIC2:$CA0D (51725)   BASIC$:$BB0E (47886)

*The poke is helpful in other ways: e.g. when printing a toolkit DUMP or FIND.

SQR

BASIC arithmetic function

PURPOSE: Calculates square roots.

Syntax: SQR(arithmetic expression). The expression must be valid and evaluate to a non-negative quantity which is within the range accepted by the floating point logic. A negative argument yields an ?ILLEGAL QUANTITY ERROR.

Modes: Direct and program modes are both valid.

Examples: PRINT SQR(2)   :REM 1.4142...
              PRINT SQR(9)   :REM 3

              1000 X1 = (-B + SQR(B*B - 4*A*C)) / (2*A)
              1010 X2 = (-B - SQR(B*B - 4*A*C)) / (2*A)

              1240 D = SQR(X*X + Y*Y + Z*Z)

SQR, like EXP, is not really needed in BASIC; either function can be obtained using the ordinary power (upward arrow) evaluation. The first examples can be replaced by 2↑.5 and 9↑.5, for instance. It is generally included because square roots occur more often than most other powers, and because in any case it looks better to have more functions. SQR is faster than raising a number to the point five, and is also more readable, and is perhaps justifiable on the latter ground alone. At any rate, the two first direct mode examples print the results of typical calculator-style square root evaluations. Only the positive root is printed: the two-line program embodying the solution to the 'general' quadratic equation has a repeat line in which the negative root is processed. The final example is another formula having a square root within it: this is an extension of Pythagoras' theorem to find the diagonal within three planes at right angles (i.e. a room or box etc.)

Notes: [1] A square root, as those people who have forgotten may like to be reminded, when multiplied by itself gives the original number, of which it is the square root. Thus, 3 is the square root of 9, because 3*3=9; and (more debatably...) 1.4142135... is the square root of 2. Early computers worked this out by iteration: the relation

$$x_{n+1} = \frac{x_n^2 + Y}{2x_n}$$ provided continually better estimates for SQR(Y).

[2] Other powers can be set up in machine code by imitating the SQR mode of operation. In BASIC 4, this routine, called by USR, will return fourth roots instead of square roots:

    JSR $CD42   (Decimal equivalents:
    LDA #$08
    LDY #$D3   32,66,205,169,8,160,211,76,15,209)
    JMP $D10F

A and Y point to an address in ROM holding .25; more generally, the value won't exist in ROM and will have to be put into RAM.

[3] In lower-case mode, shift-colon prints a square root symbol. (Or a tick?)

Abbreviated entry: sQ

Token: $BA (186)

Operation: SQR is positioned immediately before the power routine which computes x^y. It moves the argument up into floating point accumulator #2, then loads accumulator #1 with the floating point form of .5. Then it drops into the power function, so that x^y is evaluated as the special case $x^{.5}$.

ROM entry points:

    BASIC 1: $DE24 (56868)
    BASIC 2: $DE5E (56926)
    BASIC 4: $D108 (53512)

ST

BASIC reserved variable

PURPOSE: provides a record of the status of the system after any read or write operation to tape, printer, disk, or other peripheral. In this way, an error condition can be noted without stopping BASIC. The variable ST is reset to zero at the start of each input/output process.

Syntax: ST resembles a real variable, and may be assigned, printed, and used in tests with a conditional statement. ST is not stored in the RAM area which holds other variables; it is generated when needed from a single byte. For this reason ST is not a keyword and is not tokenised. BEST therefore is a legitimate variable name, equivalent to BE. STATUS is equivalent to ST.

Modes: Direct and program modes are both valid.

Examples: The table shows the possible meanings of ST, with notes on interpreting them. In each case ST takes the value of a power of 2 (1, 2, 4, 8, etc.) when it is non-zero; each error-type ORs the byte holding ST with #1, #2, or whichever is the conventional value. Note that cassette processing, which does not use the IEEE bus, has a different set of meanings from those returned by all devices 4 and upward, i.e. printer, disk units, modem.

ST (hex)	ST (dec)	Cassette #1 or Read	#2 Write	IEEE (e.g. all Read	CBM peripherals) Write
01	1				Print time out *
02	2			Input time out *	
04	4	Short block on input²			
08	8	Long block on input²			
10	16	Mismatch on checking³	None		
20	32	Checksum error³			
40	64	End of file on input**		End of file (EOI)	**
80	-128	End of tape marker ***		Device not	present ***

*Means that after PRINT#4,"MESSAGE" ST takes the value of 1, implying that the device responded in a longer time than 65 milliseconds, or may not have responded at all. Some CBM printers return 1 even when working perfectly. This is more important with modems than printers, where it is usually obvious if the device isn't printing correctly. Error ST=2 means the device is slow, and has responded too slowly or not at all. Typically, a statement like this: 100 INPUT #5,M$: IF ST=2 GOTO 100 uses ST with a slow peripheral. Note that in BASIC 4, the time out feature may be disabled: see Chapter 15's RAM map.
²Tape data files are read into the cassette buffer. One block occupies 192 bytes. If a program file is read instead, one of these errors will occur, since the anticipated separation into blocks won't be present.
³Either or both of these errors may occur on reading tape, with INPUT# or GET#, or LOAD, or VERIFY. They are part of the tape security system. On LOAD, for example, if the inconsistencies between the two programs which are recorded on tape are too great, ST is set to 16 and ?LOAD ERROR stops BASIC. (A checksum error doesn't generate this message, because ST is ANDed with #$10, which is why tape loads can be faulty but nevertheless seem to be OK). Note that VERIFY can be run in program mode, i.e. from within a program, so self-checking of a program load is possible, though time-consuming, with tape.
**A file, if it has been CLOSEd correctly after being written, has a marker to indicate end-of-file. So BASIC like this: 100 INPUT#2,X$: IF ST=64 GOTO 1000 provides a typical means of checking for end-of-file. Since files may be of any length, some such method is necessary, of course, but often it is easier to write one's own marker, or keep a record of the number on file at the start of a file.
***An end-of-tape marker is simply a block on tape holding a special number. The tape need not, in fact, end there; its function is purely to stop the tape recorder from attempting to read blank tape or tape holding unwanted data. This will cause BASIC to crash with a ?FILE NOT FOUND ERROR. The corresponding IEEE status means device doesn't respond; if the entire IEEE

bus is unresponsive, again BASIC will crash, in this case with ?DEVICE NOT PRESENT ERROR. But if the bus is partly active, ST is set to -128 without a program crash, so that a program loaded from disk can use this test to determine whether a printer is turned on and, in the event that ST returns set to -128, print a warning message to the screen.

```
OPEN 5,5: PRINT#5: PRINT ST : REM GIVES ST=-128 IF DEVICE 5 DOESN'T EXIST
```

Notes: [1] ST is set to zero by GET, INPUT, and PRINT in addition to the input/output commands CMD, GET#, INPUT#, and PRINT#. The information within ST therefore must be tested after every input or output, if you are using ST. This ephemerality does not apply to the disk status variables DS and DS$. When both are in use, test ST first, with, for example:

```
IF ST OR DS THEN: REM ERROR OR END OF FILE PROCESSING ROUTINE.
```

[2] <u>Limitations of ST</u>. This variable illustrates one of the dilemmas which face anyone attempting to design a good computer system. If the hardware is reliable, but not infallible, how can errors be signalled to the computer? The program may simply stop, or alternatively some indicator can be used, but this may be ignored. Either way has its drawbacks. The status byte, used from BASIC, combines a bit of each. Some errors, those which are more difficult to detect, are not implemented. For example, there is no facility to read back tape immediately after writing to it, so tape write errors are undetectable, except for programs which may be VERIFYed. Many of ST's messages can be programmed around, notably the end-of-file status. It is, in fact, entirely feasible to ignore ST altogether.

[3] BASIC>1 holds ST in location $96 (130 decimal). BASIC 1 uses $020C (524). When reading files in machine code it is common practice to check for end-of-file by reading ST, which is a simple operation. Chapter 14 has details with examples. This method is not always usable, because some devices don't set EOI true on the last byte of data, but send carriage return and line feed instead.

[4] Like TI, ST can be set up as an ordinary numeric variable. If a variable is assigned a value from BASIC, found with VARPTR, and altered to ST, you will have an assignable ST which can be given values like 999 at will - to the considerable surprise of some other programmers.

Abbreviated entry, Token: Neither of these are applicable.

Operation: A special ROM routine performs an inclusive OR with the location which holds ST whenever an error is found. The accumulator is loaded with #80 or whatever and the routine called to enter it. For this reason it may be possible that several errors simultaneously are included in ST. Apart from this side of things, several routines exist in BASIC to watch for ST and process it if found; PRINT has a subroutine to evaluate variables in which ST is checked and this is used by assignments too ('X=ST' say). The routine to create a new variable tests for all the reserved words, rejecting attempts to set ST.

ROM entry points:

Flag in error:	Look for ST, process it:
BASIC 1: $FBE5	BASIC 1: $CE82
BASIC 2: $FB7F	BASIC 2: $CE75
BASIC 4: $FBC4	BASIC 4: C00F

STOP

BASIC command

PURPOSE: Causes a program to exit to immediate mode and print a message indicating the line at which STOP was encountered. Like END, this command may be used to set breakpoints in BASIC programs. CONT causes a program to continue at the next instruction after STOP.

Syntax: STOP has no parameters. It may be followed by spaces, and must be followed by an end of statement byte - either a colon or a zero byte at the end of the line.

Modes: Direct and program modes are both valid. Direct mode is of little use.

Examples: STOP is not often used in finished programs, since users of a program are unlikely to want information about esoteric matters like the internal workings of BASIC. Its importance lies in this fact: since the line on which it was found is printed, breakpoints can be scattered throughout programs, and particularly in difficult parts of a program with bugs.

```
1901 IF X<> VAL(X$) THEN PRINT"ROUNDING ERROR": STOP
510  J$="£$#& ": FORJ=1TO LEN(J$): IF IN$<>MID$(J$,J,1) THEN NEXT:
     PRINT "**VALIDATION WRONG" :STOP
```

Both these examples illustrate the use of STOP as a temporary measure, put in to trap errors which may occur through faults in other parts of a program or subroutine. In the first case, X$ is supposed to have value equal to X; in the second, only characters in the string J$ are supposed to be present as IN$.

Notes: [1] When a program exits to direct mode, any variable can be printed and changed without effect on the program. CONT will still operate. If entirely new variables are input, CONT usually still operates.

[2] Editing a program will cause CONT to reply with ?CAN'T CONTINUE ERROR. Some BASICs (eg Sharp) permit a shortened, edited program to retain its variables, and machine code routines to do this can be written for Commodore BASIC. Usually though editing loses the variables, so the program must be run again to reach the same position as obtained before.

Abbreviated entry: sT

Token: $90 (144)

Operation: This routine is virtually identical to END, except that on entry, the carry bit is set, so that the final branch near the end of the routine prints "BREAK IN" followed by the linenumber.

Note that FFE1, the ROM routine to test the stop key, calls this routine; if the stop key was pressed, this sets the zero bit and STOP is entered as though the command STOP had been encountered.

Rom entry points:

```
BASIC 1: $C71C (50972)
BASIC 2: $C73F (51007)
BASIC 4: $B7C6 (47046)
```

STR$

BASIC string function

PURPOSE: Converts a number, or numeric expression, into a string. When held in string form a number cannot be added to or multiplied, but is instead a string literal, which can be edited, formatted, and modified like other strings.

Syntax: STR$(arithmetic expression). The arithmetic expression can take any value accepted by the floating-point accumulators (up to about ±1.7 E 38). However, not all the accuracy of the original is necessarily retained. This function uses the same buffer as PRINT, and its results are held in exactly the same way. Thus, STR$(.005) is *not* ".005" but as " 5E-03". There must be no space between STR and $ unless you wish array ST$() to be understood. (BASIC 1, however, allows this space).

Modes: Direct and program modes are both valid.

Examples:
```
     PRINT STR$(123) + ".00"        : REM RESULT IS 123.00
     1210 N$=MID$(STR$(N),2)        : REMOVES LEADING SPACE FROM +VE N
     PRINT "$" + STR$(DOLLARS) + " =TOTAL."
```

The major uses of this function are probably routines to round and edit numerals, and routines to compress numerals for storage when disk space is limited. Both these techniques are too elaborate to discuss here: see PRINT USING and Chapter 4 respectively for programs.

The first of the three examples above is a small scale editing program. PRINT 123 simply prints 123, and so does PRINT 123.00. Trailing zeros can be introduced in BASIC only by use of STR$. The second example is another editing routine; the leading space which CBM BASICs introduce in positive numbers (this space holds the minus sign with negatives) can be unwanted. Suppose you wish to edit numbers less than 1 to appear as 0.05, 0.36, etc. "0"+STR$(N) for small N gives 0 .05 which of course is the wrong format. Line 1210 removes the space, and will also remove a minus sign if there is one; I'm assuming all the quantities are positive. (Incidentally, not all BASICs do this. Apple BASIC hasn't the space). The final example shows that STR$ is a genuine string function, and can be concatenated and processed like other strings and literals.

Notes: [1] <u>Summary</u>. For numbers in the everyday range, this function is fine. The most likely bug is caused by the conversion routine mentioned already. STR$(1234000000000) is *not* "1234000000000" but "1.234E12". Many rounding routines fall into this trap at the low end of the scale. A number supposed to be zero may accumulate rounding errors and appear as 1E-9. The leading space feature can be demonstrated like this:
```
     PRINT "*"STR$(24)"*"  prints  * 24* , while
     PRINT "*"STR$(-24)"*" prints  *-24*.
```

Abbreviated entry: stR (includes $)

Token: $C4 (196)

Operation: This function is a good example of a string assignment and would be helpful if a string USR function or string function definition existed with BASIC. First of all, the numeric mode flag is checked; if location 7 (in BASIC>1) has its high bit not set, indicating that the expression was numeric, this is accepted. So STR$("ONE") is rejected. Now the ROM routine, shared with PRINT, is called which puts the ASCII equivalent of the number into $00FF - $010F. A subroutine return address is popped from the stack, pointers are set to $00FF, and the string set-up routine processes the string expression of which STR$ is a part, perhaps giving a ? FORMULA TOO COMPLEX ERROR, but probably, we hope, not.

ROM entry points:

```
     BASIC 1: $D349 (54089)
     BASIC 2: $D33F (54079)
     BASIC 4: $C58E (50574)
```

SYS

BASIC system command

PURPOSE: Transfers control to the address following SYS. This is executed as 6502 machine code until an RTS instruction or equivalent is encountered which corresponds to the SYS command; at this point control is resumed by BASIC. This command is essential in running machine code subroutines and is often used to run large machine code programs. Some knowledge of 6502 programming is necessary to understand this command.

Syntax: SYS arithmetic expression. The expression must evaluate to a numeral within the range 0-65535; non-integers are rounded down. This is a command, not a function: X=SYS 634 is invalid. Brackets are *not* needed.

Modes: Direct and program modes are both valid.

Examples: SYS 11*4096: REM THE EFFECT IS IDENTICAL TO SYS 45056 (OR SYS 45056.4)
1230 SYS CH: REM CH HAS BEEN ASSIGNED 826
12500 SCROLL=57377: SYS SC: REM SCROLL UP WITH 8032

These three lines of code show typical SYS calls: the first enters a routine at $B000, perhaps a Toolkit. The second and third show algebraic addresses where the SYS call may be varied: the effect of the first depends on the code contained at address CH; the second is an actual entry point for the 8032. SYS of a ROM routine is of course always reliable - provided that the address is right and the ROM set hasn't been changed - whereas RAM routines must always be loaded or poked into memory. Readers not yet familiar with machine code might try the following short demonstration: POKE the values 162,0,138,157,0,128,232,208,249,96 into 900 to 909. These numbers correspond to this machine code:

SYS 900 displays all 256 VDU characters on the screen. The routine can also be called at other entry-points: SYS 902 for instance has results depending on the entry value of X.

```
900 $0384 LDX #$00; Load X with zero
902 $0386 TXA       ; Transfer X to A
903 $0387 STA $8000,X; Put A into VDU
906 $038A INX        ; Increment X
907 $038B BNE $0386  ; Branch if X<>0
909 $038D RTS        ; Return
```

The routine can itself be modified from BASIC; this direct-mode line calls the routine 256 times, modifying the screen address each time:-

FOR J=0 TO 255: POKE 904,J: SYS 900: NEXT

Examples in ROM: BASICs 1 and 2 extend from $C000 to $FFFF; BASIC 4, with added disk instructions, is longer and occupies $B000-$FFFF. Each BASIC has ROM missing from $E800-$EFFF; so early BASICs occupy about 14000 locations, while BASIC 4 takes up 18000. There is about an evens chance of a location taken at random will directly enter ROM as BASIC runs it; there is a high chance that SYS of such a location will either go into some variety of loop, or change some of the locations and variables used by BASIC. Machine code may become corrupted. Because of these uncertainties it may be advisable to reset the machine if a SYS has been wrongly run. This can happen inadvertently: SYSS826 if entered by 'return' will be interpreted as SYS S8. This is likely to be zero, and if so SYS 0, having the same effect as USR(0), will be performed; but S8 could take any value.

Two short demonstration programs follow, illustrating strange effects which may occur with indiscriminate SYSing. The cure generally is to switch off, or, what is better from the hardware point of view, call the routine to set up BASIC.

[i] POKE 634,248: POKE 635,96: SYS 634

This short routine sets a flag in the 6502 known as the decimal bit, D. Now BASIC tries to do its calculations in packed decimal mode. The attempt is not a success. Exit to the monitor will clear the decimal bit, if you can get to it.

[ii] 50 REM *** JOKE WHICH LEAVES MEMORY IN A STRANGE STATE ***
     100 POKE 52,255: POKE 53,255: POKE 40,0: POKE 41,0
     110 PRINT "[CLEAR]"
     120 SYS 57757 : REM BASIC 2; BASIC 4 IS SYS 54321.

     (BASIC 1 VERSION: POKE 134,255:POKE 135,255:POKE 122,0: POKE 123,0:
     PRINT "[CLEAR]": SYS 57690)

Examples. A small selection:

	BASIC 1		BASIC 2		BASIC 4	
RESET BASIC AS IF SWITCHON:	$FD38	(64824)	$FCD1	(64721)	$FD16	(64790)
PRINT "? OUT OF MEMORY ERROR":	$C357	(50007)	$C355	(50005)	$B3CD	(46029)
PERFORM NEW (I.E. ERASE BASIC):	$C553	(50515)	$C55D	(50525)	$B5D4	(46548)
ENTER MONITOR BY 'CALL' ENTRY:	not applicable		$FD11	(64785)	$D472	(54386)
SCROLL SCREEN:	$E559	(58713)	$E53F	(58687)	$E3E8	(58344)
½ SECOND DELAY LOOP:	$E5C8	(58824)	$E5A8	(58792)	$E412	(58386)
SCROLL SCREEN DOWN:	not applicable				$E3C8	(58312)
TINKLE THE BELL:	not applicable				$E6A4	(59044)
SWITCH TO LOWER/ TO UPPER CASE:	not applicable				$E07A	(57466)
					$E082	(57474)

Notes: [1] Although usual, it is not necessary to end machine code with an RTS.
    An RTI can be used, provided an extra byte has been pushed onto the
    stack to be treated as a processor status register. It is common for
    JMP $ABCD to replace JSR $ABCD/ RTS. Commodore's BASIC is full of
    jumps where the return is stored on the stack: the point is that the RTS
    of the called routine is used, saving one address on the stack.

    [2] It is often easier to use SYS rather than a wedge when writing routines
    to be used by BASIC. The wedge will probably need to coexist with other
    wedges, and a lot of checking for symbols like ! or @ may be required.
    SYS is also easier to handle from the point of view of returning control to
    BASIC. The drawback is that a large number of SYS addresses may be an
    irritant.

    [3] *Some BASICs use 'CALL' for this command, for instance Apple, which
    also uses (rather clumsy) signed integers.*

    [4] The fact that SYS computes its destinations can be used to access
    jump tables; SYS 826 + 3*A for example with $033A JMP $ABCD/ JMP $1234...

    [5] Some SYS commands (e.g. of VlC) allow A,X,Y, and perhaps SR to be
    set from BASIC, and this is often convenient. This type of SYS begins with
    something like this: LDA SR-STORE/ PHA/ LDA A-STORE/ LDX X-STORE/ LDY Y-STORE
    / PLP before jumping to the SYS address specified from BASIC.

Abbreviated entry: sY   Token: $9E (158)

Operation: The expression is input, validated, and converted into a 2-byte
    integer which is stored in ($11) in BASlC>1 in the normal 6502 way, with
    the low byte first. An indirect jump is made to this address, and since
    it holds the address after SYS, the correct machine code is executed. (An
    indirect jump is represented JMP ($0011); the destination address is loaded
    from ($11)).

    NB: the argument is converted into two bytes, of which the high byte is
    held in the accumulator and the low byte in the Y register, in addition to
    the special zero-page locations. For example, SYS 1024 shows A=4 and Y=0
    on entering the monitor. SYS therefore does not behave in the same way as
    BASIC when entering ROM routines; the address in the table above for
    NEW is not quite the same as that given under NEW itself, because the
    validation process of BASIC doesn't apply to SYS.

ROM entry points:

    All ROMs use the Kernel jump table entry of $FFDE, from which is called:
    BASIC 1: $F695 (63125)
    BASIC 2: $F684 (63108)
    BASIC 4: $F6C3 (63171)

TAB(

BASIC format function

PURPOSE: Prints spaces or cursor-rights to the screen, if the TAB parameter in brackets exceeds the current cursor position. This instruction is normally only used within a PRINT statement. The function is not usable with a printer.

Syntax: PRINT ... TAB(arithmetic expression). The expression must take, after rounding down if necessary, a value in the range 0-255. No spaces may appear between TAB and (, except in BASIC 1. The interpreter will treat such a construction as TA(x), an element of an array TA().

Modes: Direct and program modes are both valid.

Examples:
```
3020 IF OP$(P)="" THEN L=P: PRINT TAB(15);: GOSUB 200:
     PRINT TAB(25) "???": NB=1: GOTO 3065

20 PRINT "[HOME]"TAB(120) "[RVS][40 or 80 SPACES]"
30 FOR X=40 TO 79:  REM OR 80 TO 159 WITH 80 COLUMNS
40 PRINT "[HOME]" TAB(X) "[RVS] " TAB(X+40) "[RVS] " TAB(X+120)
   "[RVS] " TAB(X+160) "[RVS] "
50 PRINT "[HOME]" TAB(X-1) " " TAB(X+39) " " TAB(X+119 " "
   TAB(X+159) " ": NEXT
```

The first example is a typical print-to-screen statement, in fact part of a disassembler in BAS1C which prints '???' if an unrecognised opcode has been found. The effect is similar to that achieved by SPC(, except that TAB(enables easier left-justification. The second example is a small program in which the argument of TAB(runs across three lines. It draws a bar across the screen and walks a cross-piece along it (rather slowly).

Notes: [1] TAB(can be made to print spaces - not just cursor rights - to the screen, erasing the matter over which it prints. See note 1 of SPC(for information and for a demonstration program including both functions.

[2] TAB(works by subtracting the current cursor position from the TAB(parameter, and, if the result is positive, dropping into the SPC(routine to print that number of characters. Consequently, it will not work properly with a printer, unless the duplicate information is printed onto the screen. The same considerations apply to the TAB key as implemented on the 8032. For these reasons, there is a lot to be said for avoiding TAB(and SPC(, unless you are certain to be using screen output only for the PRINT statements involved. And when modifying programs for other machines to run on a PET some conversion routine may be needed; typically,

```
SP$="                        ": REM SPACES
PRINT LEFT$(SP$, 10-LEN(E$)) ; E$
```

might be used to right-justify, in this example spanning 10 columns.

Abbreviated entry: tA (includes left parenthesis)

Token: $A3 (163)

Operation: See PRINT. IF TAB(is found in a PRINT statement, the expression in parentheses is evaluated and validated as usual. (The right hand bracket has its own check). The processor status flags are pushed onto the stack; carry bit set distinguishes TAB(from SPC(within the routine.* Now the current cursor position (as in HTAB/VTAB) is subtracted from the evaluated parameter, and, if the result is non-negative, characters, either space or cursor right, are printed singly by the SPC(routine.

ROM entry points:

```
BASIC 1: $CA13 (51731)  *Processed slightly differently.
BASIC 2: $CA07 (51719)
BASIC 4: $BB08 (47880)     TAB KEY: $E2A0 (58016)
```

TAN

BASIC arithmetic function

PURPOSE: Evaluates the tangent of the argument, which is assumed to be in radians. The tangent is a ratio which is constant for an angle; the diagram illustrates this. Note that this function has nothing to do with the tangent to a circle.

Syntax: TAN(arithmetic expression). The expression must be syntactically correct and within the range acceptable to the floating-point logic (± 1.7 E 38 approx.)

Modes: Direct and program modes are both valid.

Examples:
```
PRINT TAN([PI]/2)              prints tan of 90°; ?OVERFLOW ERROR
PRINT TAN(45 / 57.29578)       tangent of 45° = 1
100 X= (TAN(A) + TAN(B)) / (1 - TAN(A) * TAN(B))
500 A = ATN( TAN(A) ) : REM CONVERTS TO RANGE -90° to +90°
```

The first two examples show the use of radians and of degrees; a radian is 57.29578°, so this value may be used to interconvert between the two measures. 90° (pi/2 radians) and all the cyclically repeating equivalents like 270° and -90° yield an overflow error; if such values are not tested for, the program, although otherwise perfectly correct, will crash if such a value is generated in the course of processing.

The third example is one of the very many functional relationships which hold between trigonometrical ratios; in this case, the value of A is made equal to TAN(A+B) by the use of the standard formula.

Fourthly, program line 500 demonstrates the connection between the arctan function and the tangent. A function can only return a single value. For this reason ATN(A) returns only the principal value of angle A, not an infinite sequence of alternative solutions. So line 500 converts angles out of the normal range into their principal value equivalents.

Notes: [1] The diagrams show (i) the sides of the right-angled triangle from which the tangent is computed, and (ii) how a radian is related to a circle. 'O'and 'A' conventionally represent the sides opposite and adjacent, respectively, to the angle.

TAN(x)=O/A
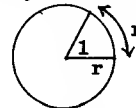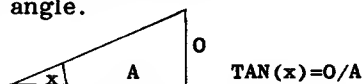
[2] Another measure of angles sometimes used is the grad, where 100 grads make up a right angle. Pi radians therefore = 200 grads, and the conversion figure is 63.6620.

[3] This function is evaluated by calculating the sine of the angle, and dividing by the cosine. Its speed and accuracy therefore are not so good as these other functions.

Abbreviated entry: None

Token: $C0 (192)

Operation: The argument is evaluated and checked, then goes through these stages:
The argument in stored in the temporary storage area starting $54 (BASIC>1)
Sine is evaluated /
The result is stored in the temporary storage area starting $4B (BASIC>1) /
The argument is retrieved from temporary store/
Cosine is evaluated/
A pointer is set to the area holding sine; and the division routine entered.

ROM entry points:
```
BASIC 1: $DFEE (57326)
BASIC 2: $E028 (57384)
BASIC 4: $D2D2 (53970)
```

TI & TI$

BASIC reserved variables

PURPOSE: Gives BASIC access to the internal clock. This is updated at each interrupt by software. It can be used to keep time, display the time, calculate elapsed time, and perform processing for timed intervals.

Syntax: TI and TI$ resemble variables, and may be accessed by PRINT or in assignments, as in PRINT TI$ or TX=TI. In each case the name TI or TI$ is specifically looked for, and, if found, processed by a set of special routines. TI and TI$ are not stored in the usual RAM area for variables. Instead, they are generated when required from three bytes which make up the jiffy clock storage area. For this reason TI and TI$ are not keywords, and ANTIC for instance is a legitimate BASIC variable. The enclosed TI is ignored. ST,DS and DS$ are processed similarly.

Modes: Direct and program modes are both valid.

Examples:
```
        TI$="130500"
        100 TI$=H$ + M$ + S$

        200 T=TI
        210 IF TI-T<120 GOTO 210: REM 2 SECOND DELAY LOOP

        1000 PRINT LEFT$(TI$,2) ":" PRINT MID$(TI$,3,2) ":" RIGHT$(TI$,2)

        5260 TI$="000000": FOR J=0 TO 9E9: IF TI=600 THEN J=9E9: GOTO 6000
             ....
             ....
        6000 NEXT
```

TI$ can be assigned a value; TI cannot. The value must be within the 24 hour range (and "240000" is accepted). The first two examples show this. The string must be of length 6, and leading zeros must be included if necessary, to conform to the hhmmss format. The time is set to five past one in the afternoon in the first example; in the second, each part of the time should be a two digit number.

The third example shows a two second delay loop. (120 jiffies is 120/60 seconds which gives 2 seconds). Some CBM manuals have an obscurely worded warning against this construction, which is repeated more comprehensibly in Osborne/Donahue. The point is that TI is not a monotonic increasing function. If you're very unlucky it may just happen to change from the equivalent of, say, "235940" to "000130", so the delay loop will be rather longer than intended.

The fourth example prints the time in hours, minutes, and seconds in this format: hh:mm:ss. Many routines have been written to present attractive graphics versions of the time, almost always as digital clocks.

Finally we have an example of the clock timing a loop. At the start the clock is reset, to avoid any possibility of the twenty-four hour trap occurring. We set up a huge loop with an arbitrary terminating value, and with a test-and-exit routine which is called each time the loop repeats. It is set for 10 seconds in the demonstration line; of course if the contents of the loop process slowly this time may well overrun.

Notes: [1] Three bytes hold the jiffy clock: 141-143 in BASIC>1. These locations are all set to zero on switching on the machine.
    PRINT 256*256*PEEK(141) + 256*PEEK(142) + PEEK(143) is the same as TI. TI$ is evaluated from TI by a long routine involving a table of values in ROM. It is related to TI by the following formula, which converts TI$ to jiffies:

PRINT 60*(3600*VAL(LEFT$(TI$,2)) +60*VAL(MID$(TI$,3,2)) +VAL(RIGHT$(TI$,2)))

BASIC 1 uses locations 200-202 for its jiffy clock.

[2] The clock update routine is called from the interrupt routine. All ROMs use $FFEA in the 'kernel' jump table as the clock update routine, so if this is repeatedly called the clock will advance at an abnormal rate. There is a software 'correction clock' included in the implementation on all ROMs. Two locations, treated as 16 bits, are successively incremented at the same time as the clock itself; however, when it has counted to a certain point, it is reset to zero, and the clock is not, on that occasion, incremented. Every 623rd interrupt does not increment the clock; the interrupts happen about 1/6% more than 60 times per second. The routine has a further feature, connected with the STOP key. Obviously, to implement a key which is able to stop a program at any time, it is not sufficient to leave the key's ASCII value in the keyboard buffer, because until there is a GET or equivalent, it will be ignored. It would be possible to cause a stop key to generate an interrupt of its own. But since the PET has interrupts continuously occurring, $FFEA not only updates the clock but also, in a short piece of code near the end, stores the keyboard PIA contents in zero-page ($9B in BASIC>1). Now $FFE1,the routine to test the stop key, simply has to look at this location to determine whether STOP has been pressed. This is the reason that a change of IRQ address can turn off the clock and the stop key simultaneously.

[3] If the interrupt disable flag is set for any length of time (the 6502 command is SEI) the clock will lose time. Compu/think disks and the screen scroll with BASIC<4, for example, both do this. The clock cannot therefore be assumed to be completely accurate; in any case, because 1/60th second is the smallest quantum of time that TI can deal with, decimal points beyond the second are meaningless.

[4] TIME and TIDE and TIMER$ are a few of the many possible equivalent names of these variables.

[5] With a little jiggery-pokery we can assign any value to TI, and any string to TI$, by evading the normal mechanisms for checking these variables. (The same trick can be done with ST, DS, and DS$).

```
10 TJ$="AHA!"
20 POKE 1084, ASC("I")+128
30 PRINT "TI$="; TI$
```

Spacing is important with this program, which does not use a VARPTR type of function. Old ROMs will require POKE 1085. (Line 20 pokes ASCII of I to replace J. The extra 128 sets the high bit, to distinguish TI$ from TI).

Abbreviated entry, Token: Neither of these are applicable.

Operation: There are three fragmented parts to the operation of TI and TI$.
[1] The routine to assign TI$ checks that the string contains only ASCII numerals and is six of these characters long. If these constraints are satisfied, the value of the string is calculated, and the result left in accumulator #1. The most significant three bytes are transferred into the clock area, where they are incremented with every interrupt.

[2] & [3] A routine is needed to assign the value of TI$ to a new string, to accomplish T$=TI$ for example; another routine allows numeric assignments, for example T=TI+60. When these details are correct, TI$ is "AHA!".

ROM entry points:

[1] Assign TI$:	[2] Assign string to time:[2]	[3] Assign numeric var:
BASIC 1: $C8DC (51423)	$CE43 (52803)	$CE71 (52849)
BASIC 2: $C8EF (51439)	$CE2E (52782)	$CE60 (52832)
BASIC 4: $B972 (47474)	$BFAD (49069)	$BFF3 (49139)

The clock update routine, $FFEA, calls these ROM subroutines:-

BASIC 1: $F736 (63286)   BASIC 2: $F729 (63273)   BASIC 4: $F768 (63336)

---

*The interrupt routine which tests Stop, checks the keyboard, and updates the clock occurs 60 times/second with 8" screen machines, 50 times/second with 12" machines. The latter have a software cheat which increments the clock twice every 5 interrupts.
[2]E.G. JSR CE2E/ JSR CA1F (BASIC 2) prints TI$. See IPUG newsletter, July '81.

TRACE

BASIC utility unavailable with CBM

PURPOSE: a TRACE is a diagnostic routine which provudes information on the way a program runs; this information is collected during an actual program run. The facilities provided by trace routines vary; some print only linenumbers, some list lines during execution, some monitor particular variables or commands only.

Versions: Brett Butler's relocating loader is a well-known trace; it appears to be in the public domain, and is listed in 'Pet Revealed' and elsewhere. It displays statements in reverse at the screen-top. Several traces display linenumbers only. Brad Templeton's Power chip has a routine somewhat similar to the following, which uses the CBM's own LIST function to print lines. Readers may

```
0 DATA 169,0,133,48,133,50,133,52,169,0,133,49,133,51,133,53
1 DATA169,76,133,129,169,35,133,130,169,64,133,131,96
2 DATA 169,56,133,129,169,233,133,130,169,48,133,131,96,0,0,0,0,0,0
3 DATA0,255,0,141,-572,142,-571,140,-570,174,18,232,224,254,208,13,236
4 DATA18,232,240,251,173,-569,73,255,141,-569,173,-569,240,119,224,253,208
5 DATA127,236,18,232,240,251,160,0,140,-568,140,158,0,32,228,255,240,251,24
6 DATA105,198,141,-565,165,54,164,55,205,-567,208,5,204,-566,240,77,172
7 DATA-568,208,184,174,-565,142,-564,169,255,32,163,229,238,-564,208,246
8 DATA162,0,181,0,157,-256,202,208,248,32,87,226,162,79,169,32,157,0,128
9 DATA202,16,250,165,54,164,55,141,-567,140,-566,133,17,132,18,32,44,197
10 DATA32,-374,162,0,189,-256,149,0,202,208,248,32,93,226,173,-572,174,-571
11 DATA172,-570,76,10,225,224,127,208,149,142,-568,236,18,232,240,251,120
12 DATA174,158,0,169,0,157,111,2,88,240,164
30 T=PEEK(52) + 256*PEEK(53)
40 L=T-614
50 FOR J = L TO L+239
60 READ X%: IF X%<0 THEN Y=X%+T: X%=Y/256: Z=Y-X%*256: POKE J,Z: J=J+1
70 POKE J,X%
80 NEXT
230 X = 0
240 FOR J = L+240 TO L+357
250 POKE J , PEEK (X+50658)
270 X=X+1: NEXT
280 FOR J=L+251 TO L+253: POKE J, 234: NEXT
290 POKE L+315, 96
310 X% = L/256: Z = L - X%*256
320 POKE 48,Z: POKE 50,Z: POKE 52,Z
330 POKE 49,X%:POKE 51,X%:POKE 53,X%
340 X% = (L+51)/256: Z = (L+51)-X%*256
350 POKE L+21, Z: POKE L+25, X%
360 X% = L/256: Z = L - X%*256
370 POKE L+1, Z: POKE L+9, X%
```

TRACE FOR BASIC 2

('UPGRADE ROM')

```
500 REM
501 REM ************************************************************************
502 REM *    FINALLY, PRINT ON/OFF ADDRESSES IN DECIMAL AND HEXADECIMAL    *
503 REM *                   AND INSTRUCTIONS FOR USE                       *
504 REM ************************************************************************
505 REM
510 PRINT "[rvs]██ BASIC <4 'TRACE' BY RAY WEST "
520 PRINT "█ENABLE:   SYS";L
530 PRINT "█DISABLE:  SYS";L+29
540 PRINT "█SAVE FROM" ; L ; "TO" ; L+360
550 PRINT "         ($";: GOSUB 600: PRINT " TO $";: L=T-250: GOSUB 600: PRINT
")"
555 PRINT "██ INSTRUCTIONS "
560 PRINT "██RVS█           PUTS TRACE ON/OFF/ON
565 PRINT "█=█            SINGLE STEPS UNTIL:-
570 PRINT "█C█ THEN █0-9█  SETS SPEED FOR STEPPING.
575 PRINT "█SPACE█         FOR FAST TRACE.
580 END
598 REM
599 REM DECIMAL TO HEXADECIMAL CONVERSION ROUTINE FOLLOWS:
600 L=L/4096:FORJ=1TO4:L%=L:PRINTCHR$(48+L%-(L%>9)*7);:L=16*(L-L%):NEXT:RETURN
```

Like to try the new trace, which I've listed here as relocating loaders for
BASICs 1 and 2. It is controllable from the keyboard and also uses the PET's
own list subroutines, so lines appear on the screen top just as they do when
LISTing. In addition there is a single-step feature. The program is discuss-
in Ch. 13, section 13.4.3 . Note that the instructions are shown on the screen
by the same routine in each version, so I've printed them once only.

RELOCATING LOADER FOR BASIC 1 TRACE

```
0  DATA169,0,133,130,133,132,133,134,169,0,133,131,133,133,133,135
1  DATA 169,76,133,211,169,35,133,212,169,64,133,213,96
2  DATA 169,56,133,211,169,233,133,212,169,48,133,213,96,0,0,0,0,0,0
3  DATA 0,255,0,141,-570,142,-569,140,-568,174,18,232,224,254,208,13,236
4  DATA 18,232,240,251,173,-567,73,255,141,-567,173,-567,240,119,224,253,208
5  DATA 127,236,18,232,240,251,160,0,140,-566,140,13,2,32,228,255,240,251,24
6  DATA 105,198,141,-563,165,136,164,137,205,-565,208,5,204,-564,240,77,172
7  DATA -566,208,184,174,-563,142,-562,169,255,32,195,229,238,-562,208,246
8  DATA 162,0,181,0,157,-256,202,208,248,32,105,226,162,79,169,32,157,0,128
9  DATA 202,16,250,165,136,164,137,141,-565,140,-564,133,8,132,9,32,34,197
10 DATA 32,-372,162,0,189,-256,149,0,202,208,248,32,219,229,173,-570,174,-569
11 DATA 172,-568,76,198,224,224,127,208,149,142,-566,236,18,232,240,251,120
12 DATA 174,13,2,169,0,157,14,2,88,240,164
20 REM
21 REM *******************************************************************
22 REM *THESE DATA STATEMENTS SET UP THE CONTROL PART OF TRACE (NO LIST YET)*
23 REM *                                                                  *
24 REM *AND LINES 30-70 POKE MEMORY, RELOCATING ADDRESSES MARKED BY NEGATIVE*
25 REM *******************************************************************
26 REM
30 T = PEEK(134) + 256*PEEK(135)  : REM T = CURRENT TOP OF MEMORY (OLD ROM)
40 L = T - 612                    : REM ENTIRE TRACE IS 612 BYTES LONG.
50 FOR J = L TO L+239             : REM DATA STATEMENTS OCCUPY 239 BYTES.
60 READ XX: IF XX<0 THEN Y=XX+T: XX=Y/256: Z=Y-XX*256: POKE J,Z: J=J+1
70 POKE J,XX                      : REM POKE DATA (OR HIGH BYTE, IF -VE.)
80 NEXT
200 REM
201 REM *******************************************************************
202 REM *    OLD ROM: MOVE $C5D5-$C648; NEEDS USR ROUTINE TO PEEK (BORING!) *
203 REM *              THIS  ROUTINE WILL LIST 1 LINE WHEN MODIFIED SLIGHTLY.*
204 REM *******************************************************************
205 REM
208 REM *** USR SET UP IN CASSETTE BUFFER £2 (ROUTINE IS RELOCATING) ***
209 REM
210 DATA32,109,219,165,180,133,178,160,0,177,178,168,169,0,32,120,210,96
220 FORI=826TO843:READX:POKEI,X:NEXT:POKEO,76:POKE1,58:POKE2,3
230 X = 0
240 FOR J = L+240 TO L+356
250 POKE J , USR (X+50645)
270 X=X+1: NEXT:                  :REM X COUNTS POSITION OF PEEK IN ROM
280 FOR J=L+251 TO L+253: POKE J,234 :NEXT: REM   3 NOP OPCODES ERASE CRLF
290 POKE L+313, 96                : REM   FROM 'LIST'.  THEN RTS.
300 REM
301 REM *******************************************************************
302 REM *   SET END-OF-MEMORY POINTERS BELOW 'TRACE' TO ENSURE NEWLY  *
303 REM *   LOADED MACHINE-CODE DOESN'T GET OVERWRITTEN.              *
304 REM *******************************************************************
305 REM
310 XX = L/256: Z = L - XX*256:  REM HIGH AND LOW BYTES OF END OF MEMORY
320 POKE 130,Z:  POKE 132,Z:  POKE 134,Z
330 POKE 131,XX:  POKE 133,XX:  POKE 135,XX
340 XX =(L+51)/256: Z = (L+51)-XX*256:  REM HIGH, LOW BYTES OF WEDGE
350 POKE L+21, Z: POKE L+25, XX: REM PUT WEDGE ADDRESS INTO ENABLE.
360 XX = L/256: Z = L - XX*256:  REM HIGH AND LOW BYTES OF NEW MEMORY TOP
370 POKE L+1, Z: POKE L+9, XX :  REM PUT NEW MEMORY TOP INTO ENABLE.
500 REM
```

UNLIST

System command unavailable directly on CBM BASIC

PURPOSE: To prevent LISTing and editing of BASIC programs to reduce the risk of unauthorised copying or modification.

Versions: Some microcomputers (e.g. IBM's) include commands of this sort. Their success lies in the relative inaccessibility of systems software and operating system knowledge generally. It seems unlikely that foolproof protection is possible with any widely-sold microcomputer; it is only possible to go some way towards it, relying on temporary expedients and perhaps legal measures. Many BASIC programs on sale make no attempt to conceal their inner workings but nonetheless methods have been tried;* a collection of suggestions follows, arranged roughly from simple to complex.

[1] Inclusion of characters which stop the LIST or clear the screen or affect the printer, in either REM statements or dummy lines, gives some minimal LIST protection. A program loaded from another, with the stop kev disabled may be made tiresome to get at, by including screen editing characters in its name so that it takes some effort to load directly. In this way, a directory (or line of a program) can be made rather misleading; parts of the name may appear on different screen lines, say. Unfortunately, even if a program were made completely anonymous, hardware reset methods (see Chapter 13) may be able to break into the program: the IRQ and NMI vectors therefore need to be changed to prevent this.

[2] A promising approach is to modify BASIC as it runs. A self-modifying program might change its own link addresses or linenumbers or zero page pointers. Practical Computing had some correspondence on the idea of making the first line, numbered 0, point to itself. POKE 1025,1: POKE 1026,4 does this.
Another idea is to change the link address of the first line; LIST therefore will go astray on the second line, the first still listing normally. Unfortunately, the link addresses are also used by GOTO and GOSUB when searching for earlier lines, so these commands need prefacing by a correcting POKE, like this: 75 POKE 1025,0: GET X$: IF X$="" THEN POKE 1025,34: GOTO 72
where 34 (or whatever) is the correct link value. The drawback is of course that peeking or using the monitor soon enables anyone familiar with the idea of link addresses to calculate the correct poke or pokes.
A modification of this idea is to include zero bytes within lines so that invisible bits of BASIC can (say) call machine-code hashtotal routines to detect any changes in the program. Hidden BASIC though is rather vulnerable to editing and tends to reappear.

[3] Overlong program lines (length exceeding 255) can be used, and the resulting program is genuinely unlistable; LIST can't handle it. Some other commands will also come to grief, e.g. READ; the ideal candidate for such a line is a full screen of data printed by a huge line. Such a line at the start of a program will stop LIST, unless zero bytes are reinserted with links.

[4] A method reportedly present in a prototype 'Toolkit' works as follows: (See Liv. Soft. Gaz. Dec '80): Each line begins with 5 colons, like this: 10:::::A=5. Or in fact any five characters or tokens will do. The UNLIST puts a zero byte after each linenumber; LIST stops at the zero byte, so only linenumbers list; but RUN interprets the zero as end-of-line, and continues 4 bytes on, so the program runs successfully. This BASIC subroutine will put in the zeros; it assumes that 5 characters are present, unlike UNLIST as quoted which puts these in for you:

```
50000 :::::A=1025: FOR J=1 TO 1E8: IF PEEK(A+4)>4 THEN POKE A+4,0:
      A=PEEK(A)+256*PEEK(A+1): NEXT
50010 :::: END
```

This is however very simple to re-list; this direct-mode line will POKE colons back again: A=1025:FORJ=1TO1E8:POKEA+4,58:A=PEEK(A)+256*PEEK(A+1):
IF A<>0 THEN NEXT ,so the method is not a great success. The best you can

---

*Hardware protection includes the 'dongle', plugged into the back of the PET, and periodically checked by the software. Such protection is often easy to remove.

is to set a few traps in the hope they won't be noticed, such as including spurious BASIC in the four ignored bytes which are revived by relist. For instance, 100::::AA=50, when 'UNLISTED', is translated as A=50; on relist, unless the A is edited out, the program will run incorrectly.
200:ANEWBCA=50 is another version of A=50, since it is preceded by four bytes, one of them a token. On relist and RUN, this line will of course erase the program (or at least its pointers).

[5] Compu/think disks have a security device which works like this: to use these disk units, an initial SYS call changes the CHRGET routine so that BASIC is intercepted by the 'Diskmon' ROM. Additional commands are then identified by a leading '$' - $D,1 for instance is the directory command. The extra commands include load - $L,1,"PROG" typically. Throughout this process CHRGET remains modified. Now, if an asterisk is found in location 1034, just after the start of BASIC, (=$040A), only three commands are allowed, RUN, and two special commands which clear the memory and run machine-code.*
LIST or SYS or any other direct command returns only READY. Moreover, linenumbers cannot be erased or edited: typing 100 Return won't delete line 100 when the machine is set as described. Setting the BRK vector to print READY makes this system, so far as memory storage is concerned, pretty foolproof. All that's needed is a REM statement in the first line, so an extra asterisk won't matter, then POKE 1034,42: $S,1,"PROG" and the unlistable program is saved on drive 1. (42 is ASCII for '*'). When this program is loaded, the intercepting routine tests direct-mode commands and rejects all except the three mentioned. This process is adaptable to other systems, but only with hardware add-ons, or with some software method for ensuring that a modified CHRGET is obligatory, since otherwise a program can simply be loaded and listed, asterisk and all, as usual. The weak point of such methods is located in the disk storage, rather than RAM storage. If the disk storage system is understood, it is possible to load a relevant part of a program, modify it, and replace it on the disk. See Chapter 6 for concepts and methods applicable to Compu/think.

[6] The most thoroughgoing systems for concealing BASIC rely on machine code routines. Disk-based BASICs, which are loaded into RAM, can of course be modified in situ; LIST can simply be deleted, or the operating system changed. In CBM BASIC this would require a change of ROM. Instead, let's consider ways of scrambling BASIC so that it will run, but not LIST. Each line may be written like this: **LINENUMBER SYS X: BASIC LINE: SYS Y**, where everything after SYS X is stored in coded form, including SYS Y. In this way each line can be decoded before execution, and encoded on leaving, with some exceptions like GOTO statements. An encoding algorithm has to be fairly subtle: adding 1 to each character would be easily undone. Typically, EOR of several variables gives a repeatable offset. More elegantly, this scrambling and unscrambling process can be carried out by rewriting RUN. The major loop controlling RUN processes single statements individually, and schematically looks like this:-

```
            JSR CLR                        A similar routine may be put into
      LOOP  TEST STOP                      RAM where a SYS call will run
            LOAD CURRENT CHR.              BASIC. One is then free to insert
            BNE NEXT ST'MENT               decoding/ coding routines before
            TEST FOR END; IF NOT,          and after the statement-processing
            UPDATE CHRGET &                call. This isn't very easy: colons
            LINENUMBER ETC.                and zeros in the original must be
NEXT ST'MENT JSR GET CHR FETCHES TOKEN     preserved as special cases, and
            JSR ACTION TOKEN (OR 'LET')    a record must be kept of the num-
            JMP LOOP PROCESSES : OR 0      ber of bytes altered by decoding.
```

A test for direct-mode, with a system reset if found, can be put into the IRQ. There are, of course, many other possible attacks on this problem. Perhaps the last word should go to Tommy Turnbull: 'We've had all kinds of protection here. The longest took an hour ...'

---

*$G, which should run machine-code, contained a bug which often caused return to the monitor. Adjusting the program's length until PEEK(42)=96 prevents this.

USR

BASIC arithmetic function

PURPOSE: Arithmetic function calling user-written machine code. Some knowledge of machine code is necessary to understand this function.

Syntax: USR(arithmetic expression). The expression, when evaluated and,if need be, rounded down, must fall within the range 0-65535. *In addition,* locations 0-2 must contain valid machine code: usually, because of the small space in zero-page, a JMP to the user's own routines. This is a function, and may be used validly in such statements as PRINT USR(345), X=USR(2), Y=USR(X).

Modes: Direct and program modes are both valid.

Examples: [i] See PEEK for details of a USR function which acts like PEEK in BASIC 1. It can be used in statements like: POKE X, USR (X-1) which are not valid in BASIC 1's implementation of PEEK, due to a bug.

[ii] The machine code routine listed here and called by USR displays the contents of floating point accumulator #1 at the top left of the VDU. 6 bytes are poked directly to the screen, so 0 appears as @, 1 as a, 2 as b, and so on, when in lower case mode. Since USR, in common with all functions, puts the argument into this accumulator, its contents and method of storage can be examined. For example:-

```
$0000 JMP $027A
$027A LDX #$05
$027C LDA $5E,X
$027E STA $8000,X
$0281 DEX
$0282 BPL $027C
$0284 RTS
```

USR(0) gives @@@@@@   or 0 0 0 0 0 0.
USR(1) gives A@@@@@   or 129 128 0 0 0 0.

[iii] $0000 JMP $CD6F   ; THIS EXAMPLE IS SGN IN BASIC 4

If you have BASIC 1, the USR replacement for PEEK is very useful. The second example (BASIC>1 only) displays the contents of accumulator #1 in separate bits, which is of interest to those readers who want to learn how floating point numbers are processed. The final example is a special case of a function, and shows how USR can access routines in ROM; in the given case, USR(X) returns the same value as SGN(X). Experienced machine code programmers may be able to write their own mathematical routines along the same lines as those of Microsoft; if so, they will be callable from BASIC by USR. It is a safe bet that this is not done often.

Notes: [1] Bytes 0-2 are initialised on switchon to print ?ILLEGAL QUANTITY ERROR, so USR without a modified instruction in 0-2 gives this message.

[2] SYS 0 carries out the same instructions as USR(0), but is a command, not a function, and so cannot be assigned or printed.

[3] Locations 0-2 need not contain a JMP. An RTS (=$60, 96 decimal) for example gives USR(N) the value N. (Or to be exact, INT(N)). An indirect jump is valid. BRK (=0) in location 0 will cause the monitor to display the contents of its registers when executing a function.

Abbreviated entry: uS

Token: $B7 (183)

Operation: The value of the argument is computed and validated and put into floating-point accumulator #1. This is normal behaviour on encountering a function. The difference is that the address now jumped to is $0000. This is an easy function to add to BASIC, since once the function-handling routines are written, hardly any more work is needed to incorporate USR.

ROM entry points:

All ROMs jump to $0000.

VAL

BASIC arithmetic function

PURPOSE: Computes the 'value' of a string or string expression: the entire string, or as much as is syntactically acceptable, is converted into a number. This function is an important converse of STR$, enabling calculations to be performed on a quantity which, perhaps for formatting reasons, is held as say " 123.45" or as "1.23E04".

Syntax: VAL(string expression). This is an arithmetic function of a string argument. The string expression must be valid; it can consist only of literals, string variables, and string functions concatenated by +. Its maximum permissible length is 255 characters. If spaces are included, when using BASIC>1 - for instance V AL(Q$) - an array VA() will be assumed, and a ?TYPE MISMATCH ERROR generated whenever this code is encountered.

Modes: Direct and program modes are both valid.

Examples:

```
PRINT VAL("123.456")               :REM RESULT IS 123.456
PRINT VAL("-127")                  :REM -127
PRINT VAL("1.2 E2")                :REM 120
PRINT VAL("E")                     :REM 1 (i.e. 1 E0)
PRINT VAL("1000000000000")         :REM 1E 23
PRINT VAL(LEFT$(TI$,2))            :REM 0 TO 24 (i.e. the hour)
PRINT VAL("150+200")               :REM 150
PRINT VAL("1.2.3")                 :REM 1.2
PRINT VAL("")                      :REM 0
PRINT VAL("123" + CHR$(0) + "456") :REM 123

405 J=VAL(IN$): IF J<82 OR J>90 THEN MS$="WRONG YEAR":GOSUB300:GOTO400
```

Any string, including the null string, which evaluates as zero, is accepted by this routine, but validation is only implicit, and no error message is printed if a string has non-numeric characters in it. This can be very convenient, but equally may be a source of bugs. For this reason most of the examples are direct mode print statements. But, like ASC and LEN, VAL may be used freely in arithmetic expressions; the validation line from an input routine is an illustration. VAL will accept spaces,+,-,E (unshifted only), the decimal point .,and of course 0-9. The validation process is intricate and flowcharts for it have not been published; it is not particularly important to know the precise method of validation, though. The most significant fact is that the first unacceptable character terminates the VAL. There are three lines in the demonstrations which are actually terminated like this, ending when + and . and CHR$(0) are encountered, respectively. Remember that E refers to a power of 10, so 1.2E2 is the same as $1.2 * 10^2$ or 120.

Abbreviated entry: vA

Token: $C5 (197)

Operation: Most of the processing is carried out by a general routine to scan a string and convert it into a floating-point number in accumulator #1. The remainder is housekeeping: the mode is changed to numeric, and a length of zero causes exit, with VAL assigned zero. If, as is usual, the string has non-zero length, the current CHRGET address is saved, and a pointer to the string put in its place. This is because the conversion routine scans the string using CHRGET. A pointer to the end of the string is set up too. Both accumulators and 10 bytes of additional storage are used by the main routine; this uses CHRGET to ignore spaces and fetch 0-9 (This is signalled by a clear carry bit after CHRGET). E and decimal point are looked for; so are + and -, in both their ASCII form and as tokenised keywords. The routine calls extra routines to perform such tasks as multiply the accumulator by 10, and add the contents of A to the floating point accumulator.

ROM entry points:BASIC1:$D685 (54917) BASIC2:$D687 (54919) BASIC4:$C8E3 (51427)

VARPTR

BASIC arithmetic function unavailable directly in CBM BASIC

PURPOSE: Finds the actual location of any variable in memory, after it has been set up in the normal way on running.

Versions: Although BASIC's routine to seek variables is well known, there have been few attempts to write actual routines embodying it. VARPTR is Tandy's name, and their BASIC is the only popular one with this command. My routine points to the name of the variable, so that the location indicated for AA$(6) is the start of AA$(), and is the same value as that returned by AA$(10). To make the routine usable from within BASIC, the value must be assignable, rather than accessible only by printing, say. It therefore uses not only the variable search routine but part of LET.

The syntax is: SYS start address: sought variable: numeric variable.
The routine is relocatable; typically, SYS 634:AB$:X illustrates the syntax. X holds the value of AB$'s starting point in memory, at the time the routine was run. Array variables of course may move up memory when new simple variables are defined, but this cannot happen with simple variables.

Example:
```
10 L%=15
20 SYS 634:L%:X: REM X NOW EQUALS RAM LOCATION OF L%
30 FOR J = X TO X+6: PRINT PEEK(J);: NEXT
40 END
```

When run, this program prints out:

```
204 128  0  15  0  0  0
(L   _    0  15  holds the name and the value of this integer variable)
```

Machine code:

BASIC 1:		BASIC 2,4:		BASIC 2:-	BASIC 4:-
JSR $00C2		JSR $0070			
JSR $CF7B		JSR SEARCH		$CF6D	$C12B
LDY $AE		LDY $5C			
LDA $AF		LDA $5D			
JSR $D278		JSR FXFLT		$D26D	$C4BC
JSR $00C2		JSR $0070			
JSR $CF7B		JSR SEARCH		$CF6D	$C12B
STA $98		STA $46			
STY $99		STY $47			
LDA $5F		LDA $08			
PHA		PHA			
LDA $5E		LDA $07			
PHA		PHA			
JMP $C8B2		JMP ASSIGN		$C8C2	$B945

Notes:[1] This routine won't find TI,TI$,ST, or (in BASIC 4) DS or DS$, since these do not exist in RAM as ordinary variables do.

[2] A shorter routine can be written which prints the values without the additional assigning; this can be valuable when inspecting variables in direct mode. Replace JSR FXFLT, which converts a 2-byte number into floating point form, by a print routine. In this way only 13 bytes is enough for the routine - everything after the fifth instruction can be ignored. The replacement is:

BASIC 1: JMP $DC9F / BASIC 2: JMP $DCD9 / BASIC 4: JMP $CF83

VERIFY

BASIC system command

PURPOSE: Compares a stored memory dump on disk or tape with the equivalent contents of RAM. If they are not identical, ?VERIFY ERROR results. Usually VERIFY checks BASIC programs which have been SAVEd; but other memory dumps, e.g. machine code routines, may be VERIFY'd.

> *NOTE: VERIFY reads the program or dump specified, and compares it with the contents of RAM, without loading it into RAM. Consequently, VERIFY only applies to programs and other memory dumps: it cannot be used with any form of data file which is output from a buffer.*

Syntax: The syntax is identical to that for LOAD, including all the differences between tape and disk syntax.

Modes: Direct and program modes are both valid. VERIFY from within a program may be used to check a save to tape or disk; a message requesting the tape be rewound is necessary with cassettes. Unlike LOAD, the operation of BASIC is not reset; after 'OK' the program continues normally.

Examples:
```
10 SAVE "THIS PROGRAM": REM TAPE UNIT #1 ASSUMED
20 PRINT "REWIND FOR VERIFICATION - ANY KEY TO CONTINUE"
30 GET X$: IF X$="" GOTO 30
40 VERIFY
50 REM ... REST OF PROGRAM ...

SAVE "0:5TH VERSION",8   :REM TYPICAL DISK SAVE
VERIFY "0:5TH*",8        :REM TYPICAL DISK VERIFY

PRINT#15,"V0": REM THIS IS ANOTHER DISK FORMAT. V IS 'VALIDATE',
     OR 'COLLECT' (q.v.); THIS IS NOT THE SAME AS VERIFY.
```

Notes: [1] If you examine the ROM routines you'll find that VERIFY is largely identical to SAVE. However, if a flag ($9D BASIC>1, $020B BASIC 1) is set non-zero, the program is read but the bytes which are input are compared with RAM contents; if they differ, ST is set to #$10 (16) and ?VERIFY ERROR printed. Otherwise, OK appears. When using ROM routines it is good practice to set the flag, otherwise a 'SAVE' may only be VERIFYing.

[2] Some BASICSs, e.g. Apple, have a verify statement which applies both to data files and programs, but Commodore's doesn't.

[3] VERIFY, like LOAD, defaults to cassette #1, or disk unit 0.

Abbreviated entry: vE

Token: $95 (149)

Operation: See note [1]

ROM entry points: VERIFY is a 'kernel' command; its jump address is $FFDB.

```
BASIC 1: $F4BB (62651)
BASIC 2: $F4B7 (62647)
BASIC 4: $F4F6 (62710)
```

WAIT
BASIC command

PURPOSE: Causes BASIC to wait until the memory location specified by its first parameter has one or more bits configured in a way specified by its other parameter(s). Any combination of bits within the location can be tested; when any of these bits takes the sought value, the wait is over.

Syntax: WAIT arithmetic expression 0-65535, arithmetic expression 0-255 with optional third parameter ,arithmetic expression 0-255. The optional third parameter defaults to zero, as might be guessed. If the expressions do not yield integral results on computation, they are rounded down.

Modes: Direct and program modes are both valid.

Examples: WAIT is intended for such uses as handshaking, where some signal is awaited. If a program WAITs for a ROM location to change, or a RAM location which is not accessible to hardware or not updated by the 1/50th 1/60th second interrupt, then it will either wait indefinitely, or not at all. WAIT should be used only with locations whose contents vary, therefore; the examples show this:

```
WAIT 59410,1,1 : REM WAITS FOR RVS OR RVSOFF
WAIT 59410,4,4 : REM WAITS FOR SPACE OR SHIFT-SPACE
1000 POKE 158,0: WAIT 158,1 : REM CLEAR BUFFER/ AWAIT KEY
WAIT 152,1 : REM WAITS FOR A SHIFT KEY
2210 WAIT 142,1: REM RANDOM DELAY OF 0-8 SECONDS
```

Because of the hardware-related aspect of this command, an instruction which is successful with one hardware configuration may work differently with another. The examples all work for BASIC 2, but the 8032 keyboard causes the first two instructions to respond to different keys, and BASIC 1, with a different zero page allocation, can't run the last three. Numbering bits as usual 7 to 0, this is what these commands do: i. Waits until bit 0 of location 59410 is 0. ii. Waits until bit 2 of location 59410 is zero. iii. Waits until bit 0 of location 158 is 1. iv. Waits until bit 0 of location 152 is 1. v. Waits until bit 0 of location 142 is 1. The first two commands' location is controlled by the keyboard PIA; the next location holds the number of characters in the keyboard buffer; this is updated during interrupts. The fourth again uses the PIA, and the fifth the jiffy counter for the clock.

Notes: [1] WAIT is a little-used command and not a very useful one, except maybe for people with their own hardware add-ons. It is also rather difficult to explain. Consider WAIT address,a,b. WAIT peeks the contents of address, performs exclusive-or with b, then AND with a. If the result is non-zero, BASIC continues; otherwise, the loop goes on. What is the reason for this? To see the answer, let's consider an example: suppose we wish to wait for bit 4 of address to be off, or bit 2 on. In either case we are happy for the program to continue, but otherwise we still wish to wait. The first thing is to use parameter b to switch bit 4, which it does by EOR with %0001 0000. So with b=16, bit 4 is switched: now, when the desired condition occurs, bit 4 will be turned on. If parameter a is %0001 0100, the result of the first bit manipulation is ANDed, leaving a result which can be non-zero only if bit 4 was off, or bit 2 on. So WAIT address,20,16 is the solution. So, WAIT address,8,8 waits until bit 3 is off; WAIT address,48 waits for bit 4 or bit 5 to be set to 1; WAIT address,1 waits for bit 0 to turn on.

[2] WAIT 6502,n is Microsoft's joke in BASIC 2 only. n=0 to 255.

Abbreviated entry: wA   Token: $92 (146)

Operation: The parameters are computed and validated; the optional parameter is checked for; then the address is stored in ($11) in BASIC>1 and the parameters in $46 and $47 respectively. The stop key is not tested for ($FFE1) so the routine cannot be interrupted by pressing STOP.

ROM entry points: BASIC1:$D702 (55042) BASIC2:$D710 (55056) BASIC4:$C963 (51555)

CHAPTER 6: DISK DRIVES

6.1 Hardware

__Disk drives__ The disk drives we shall consider in this chapter use so-called 'floppy disks' as their 'media'. (Like 'data', in computing circles 'media' is optionally singular or plural). Alternative bulk memory-storage devices, notably sealed 'hard disks' or 'Winchester disks', named after an IBM project, are coming into greater use, and CBM have announced and shown a model; nevertheless floppies are by far the most popular storage system apart from tape. Originally introduced as an alternative method of data entry to punched cards, by IBM in the mid-70s, the techniques were taken over and used by microcomputers a few years later. In the process, IBM's carefully thought out standards were modified and to some extent dropped. For those not familiar with the concepts of floppy disks, there is an outline in the next section of this chapter. But the basic idea is similar to that of multi-track tape, arranged radially on a disk, like a gramophone record, so that any track can be selected without the long time delay inevitable when searching tape. The disks are often called 'diskettes', and the units to read to and write from them are called 'diskette drives', though in practice it is usual to talk of the units as 'disks' - 'Have you got disks?' These drives are made by specialist manufactures, for example Shugart and Micropolis, and require fairly careful handling. They are usually packaged by the computer manufacturers, and end up in boxes and machines of widely differing sizes and shapes - Apple disk drives and Commodore's may contain the same units. All these drives, when looked at without their external cases, are quite similar.

Typically, a drive unit has a read/write head mounted on rails, and a stepper motor which positions it opposite tracks on the diskette. The head is usually a ferrite and ceramic mixture bonded in glass; the step size is of the order of 1/40th of an inch. To clean the recorded track there are 'tunnel' or 'straddle' erase heads to delete any recording within a short distance from the track. The actual width of the recorded zone is something like 1/80 th of an inch. When a diskette is inserted into a drive, the clutch mechanism which grips the central hole has to position the diskette consistently to within this sort of tolerance; if the disk is also to be used with other drives, these too must be equally precise, or alignment errors will cause failure to read correctly. The drive's spindle motor rotates the disk, which, because of centrifugal force, loses some of its floppiness and may be read, sandwiched between a pressure pad and the read/write head. The rate of revolution is usually 300 r.p.m. within one or two percent. Presumably it is possible to mount 40 or so heads next to each other, reducing head seek time at the expense of disk wear (and cost), but invariably a head seek mechanism is used. The outside track (track zero) may be fitted with a light sensor and a stop, to give a fixed starting reference point - the stepper motor moving the head out until track 0 is signalled, then stepping in, perhaps to the directory track. Other sensors may detect index holes in the diskette and the presence or absence of the write-protect tab. The head has an 'actuator' which moves the head in contact with the disk when reading/writing, and away otherwise. The door of the drive also moves the head away from the disk, to avoid 'glitching' (i.e. magnetic damage) to the data near the head on power-on or off. This mechanism also clamps or unclamps the disk. Double-sided disk drives have two heads, mounted on opposite sides of the disk; they have to be offset so that each can have a pressure pad. Most microcomputer equipment has drives mounted horizontally, or vertically with front loading, but top loading is used with some desk-style equipment. Drives are often paired so backup copying (from one drive to the other) is easy; a whole disk can be duplicated onto another, for security purposes. This is not *necessary*, although it's very convenient. Single-drive copying can be done by loading a part of a disk into memory, copying this by switching disks, re-entering the original disk and reading more of it, and so on.

Disk drives are controlled by control circuits, usually controller boards which include a special disk-controller chip, with functions to turn the motor on or off, read a specified sector of a specified track, seek a track, and many more. These chips have RAM and ROM and perform a lot of error checking, returning bit patterns indicating what (if anything) is amiss. The translation of magnetic patterns into bytes is a hardware function, relying on assorted crossover detectors, amplifiers, and pulse shapers.

Some chips are programmable to return not only bytes on the data bus, but also sync marks and address markers and other housekeeping paraphernalia. Commodore have preferred to use their own chips to control the drives. Information sheets, supplied by disk drive manufacturers, provide interesting information on timing and on expected reliability of their products. This is important when writing controlling programs, but otherwise can be assumed to be taken account of in the disk operating system. For example, the time for the motor to reach a stable running speed is usually about 1 second, and the average time to move to another track about ½ second, depending on the number of tracks on a disk; in either case the operating system software ought to take these delays into consideration. Other 'soft' errors - wrong track found, or byte misread, but re-readable - should be incorporated in DOS. This is done by checking the status indication from the controller, and re-reading data, perhaps by moving the head to track zero and retrying. This may be done ten to fifty times before the error is considered 'hard'. The data sheets provided by controller chip makers include flowcharts of recommended practice, in the hope of preventing the more subtle mistakes. Commodore fell foul of one typical mistake when designing the write-protect software, where the write-gate is kept on when it should be off.

The diagram shows some of the components of a Commodore drive as it appears unscrewed, with the lid propped up. Note the position of DOS ROM chips (most of them), and that drive 'zero' is on the right, drive 'one' on the left, with the heat sink and printed circuit board above. Drive 1 tends to run hotter than drive 0.

Both the on/off switch and IEEE connecting cable are at the back of the machine. Units vary in small details; the 2040 controller board has different ROM slots and is not compatible with the 3040. The 3040 and 4040 models are very similar, with upgradable ROMs available for the 3040. It is worth noting, with your unit, which way the on/off rocker switch operates, so that you can check whether it's actually on. Many units work like this:

OFF: ▱   ON: ▱

But the 4040 has the opposite arrangement:

ON: ▱   OFF: ▱ .

Because of the nature of the IEEE bus, several PETs can be connected to the same disk unit (and printer too), but the users will have to be careful not to use the disks together; if they aren't sure enough to be able to guarantee this, commercial products of the 'Mu-Pet' and 'Regent' type are available. (I suggest that anyone considering the purchase of such a system should first ask the opinion of a current user). CBM disks are not the only storage units around. Compu/think have sold many units; Novapac, I believe, sold fewer; 'Byte' magazine of June '81 and following editions has articles on controlling disks; R. Freeman (Kbaud-Microcomputing, Jan.'80) explained how he added an S-100 disk system to an early PET. Other systems continue to arrive on the scene.

The final point I want to make in this section - it is repeated here and there in the next chapter on BASIC disk commands - is that the CBM disk units are largely autonomous and independent of the PET/CBM which controls them. DOS is held in the *disk ROM*, not in BASIC. So changing the disk unit, or swapping the ROMs in it, will cause it to act differently - for example, to be able to process relative files, where before it couldn't. The PET/CBM can drive any disk unit; BASIC 2 for example can run an 8050 or 4040 disk unit, although the BASIC 4 commands have to be transposed into their more ancestral form. See Chapter 6 for examples.
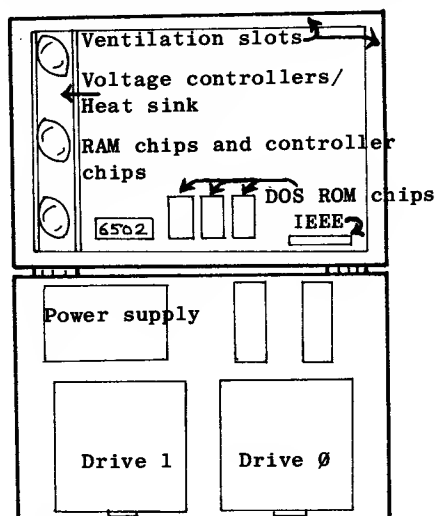
Diskettes ('floppy disks.') The diagram, which is approximately to scale, shows the typical features of a diskette. Bold lines indicate the outline of the envelope and the notches and windows in it; thinner lines mark the position of the magnetic surface itself. The diskette is square (like the sleeve of a record). It has a write-protect notch; if cut out, the disk can be written to; if it is not cut out, or if an adhesive tab is stuck over it, a disk is write-protected, provided the disk drive is designed to sense the notch and process the resulting message. Most drives have this feature; the idea is to prevent inadvertent erasure of important programs or data. Two stress-relieving notches are cut near the read-write window, the elongated slot along which the read-write head moves. The entire disk is spun within the protective envelope; part of it is visible as an annular region, which is gripped by the clamp and rotated by the spindle motor. The small hole nearby enables index markers and sector markers to be sensed; the diskette may be perforated by one or more small holes, which are inside the region used for reading/writing, and through which light can be detected. The

physical orientation of the disk, at least as regards CBM drives, is as shown, when the disk is inserted into the drive. The label side is uppermost, and the read-write slot forward. The label is deliberately positioned away from the sensitive recording surface; this reduces the chance of fingerprinting, and also enables the diskette to be put into its outer dustcover with the label visible and the read-write slot hidden. CBM equipment, and much other, uses disks which are 5 1/4" square (the disk surface is 5 1/8" in diameter). These represent a fair compromise between the size of the complete unit and quantity of storable data.
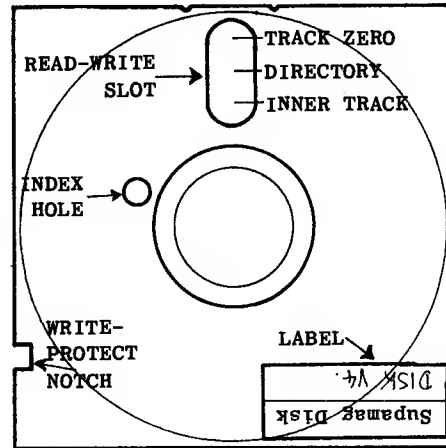
The recording surface is usually a polyethylene derivative, coated on both sides with magnetic recording emulsion. Single-sided disks are tested (I'm told) on one side only, or, if the test fails, flipped over and tested on the other. This process also tests double-sided disks, which are otherwise similar or identical to those labelled 'single-sided'. When a production run has made its quota of double-sided disks, some of the remainder may still be usable as double-sided disks, in spite of their labelling. The magazines regularly have 'new' articles explaining how to double your disk capacity by cutting new index holes and write-protect notches, so that the other side of each disk is usable. The standard argument against is that small dirt particles, trapped by the self-cleaning wiper lining the diskette, become dislodged and spread across the disk surface when the direction of rotation of the disk is reversed in this way. Of course, double-sided drives don't have this problem, as the direction of rotation is constant. The lining of a diskette depends on its quality, but is often a slippery plastic (e.g. PTFE) woven in a loose texture, like a small-scale string vest. Small contaminating particles, smoke, dust, and so on, become trapped there and don't interfere with the read-write head, or scratch the medium. A track's useful life is typically quoted as 3×10^6 passes per track. This sounds a colossal figure, but in fact, at 300 revolutions per minute, represents about 7 days' *continual* running. Disks containing important data should of course be copied and the master disks replaced at intervals. It's difficult to make useful remarks on diskette quality: it is impossible for anyone outside the manufacturing business to know whether the labels represent genuine differences, or whether the same item is repacked/ relabelled and/or mixed with other batches. The magnetic properties of retention and sensitivity alone are very complex. In practice people rely on advertising and on price as criteria. Some brands (Dysan, Scotch) advertise their reliability; Verbatim more recently has done the same thing, perhaps in response to criticism; others (3M, BASF, CDC) seem to rely on their general reputation. In any case, a programmer producing systems for anything approaching a serious use must have a rigorous program to test diskettes by writing and reading to the entire disk surface.

How is all this actually implemented on the PET/CBM's systems? Much, but not all, is standard practice. The outer track, usually called track zero, and the inner track are arranged as the diagram shows, with the directory held on a central track

(or two tracks with the 8050, because of its larger capacity). The directory track(s) contain (see 6.4) the directory, with associated pointers and flags, and the block allocation map, or BAM, which lists the available sectors for future use, so that new data can be stored in unallocated sectors, and scratched data can be redefined as free for use. The general principle is fairly standard. Apple disks for example have a 'volume table of contents' or VTOC and a 'track bit map' which have similar functions to the directory and BAM. The actual diskette may be either hard-sectored or soft-sectored, despite CBM's claim that only soft-sectored disks should be used. The light sensing system appears to be absent, so the index marker and sector holes are not relevant. (A hard-sectored disk can be identified by careful manual rotation of the disk in its envelope; if there is one hole only, the disk is soft-sectored; if there are many, for example eight, the disk is hard sectored. Hard sectors rely on light sensing to position and read individual sectors; soft sectoring uses software). We shall see in 6.5 how to program CBM drives to read or write to any track, and to any sector within that track. But first let's consider the rationale behind sectoring tracks.

The point is that small variations in the spindle motor speed cause data being written at a constant rate to vary in its physical length. Sectors are separated by gaps, and these gaps allow for speed differences between machines; a slower machine writes longer sectors, and vice versa. (There is a curious passage in Osborne-Donahue on this subject). Sectors usually hold 256 or 512 bytes of data; there may be from about 8 to about 30 sectors per track, depending on the recording method. For example, double and quad density recording stores respectively twice and four times the normal amount of data, by doubling the number of sectors or doubling each sector's contents or both. (The number of tracks varies too, of course). A diskette's total storage capacity is tracks x sectors per track x bytes per sector ... usually. Commodore uses an unorthodox and rather horrendous system in which the number of sectors increases as a track is further from the centre. This allows about 20% more data to be stored than would otherwise be available with their single-sided, single-density systems (double-density with the 8050). It means that the *rate* at which data is written is faster at the edge of the disk than the centre. Usually the rate is fixed, so all sectors occupy the same angular distance. Commodore's technique takes advantage of the fact that greater resolution is possible at the edge of a disk. (For the same reason, records reproduce sound better at the start than the end, and have large labels in the middle).

The diagram that follows illustrates the way in which data is stored on these disks. It is recovered by a decoding process, and synchronisation fields and clock pulses are detected by their bit patterns, which are not data bit patterns. Much of the reading and checking is on time, and not on counting. The 'cyclic redundancy check' is a form of hashtotal which follows the data. It is read from the disk only

------ ONE SECTOR OF A TRACK ------------------

	Gap	Sync. Field	I.D. (e.g 4 bytes)	CRC	Gap	Sync. Field	Address Marker	256 bytes of data CBM includes pointers	CRC	Gap	

after the 256 bytes or so of data are in their RAM buffer. Incomplete use of the error detecting software here, and in many other cases, may permit spurious data to enter the system.

Soft errors may be caused by physical contaminants, and by electrical noise, static electricity, defects in the disk surface, speed variations and so on; these are curable by repeat re-reading of the disk. Hard errors can be minimised by care of the system, and also by careful backup procedures. The error-checking mechanisms, if they are used, are pretty formidable, and correctly-adjusted hardware with well-designed software should be extremely reliable. Hard error rates of 1 bit in 10^{11} give an idea of the reliability attainable; this figure is quoted in a disk drive's specification sheet. Section 6.8 of this chapter summarises the care and maintenance which it is prudent to apply to drives and diskettes. However, it is worth remembering that the reliability of data transfer between a computer and disks is nowhere near that of data transfer within RAM and ROM, where several hours' running time may reach this figure (10^{11} bits transferred).

6.2 Software

What is a file?   This concept is quite difficult to grasp; only with practice and exper-
ience does it appear a self-evident and obvious idea. Data when stored in some device
external to the computer (tape, disk, another computer, etc) has to be arranged in
some sort of logical way, to be accessible again; any collection of accessible data may
be called a file. The operating system usually enables files to be named, so that they
are easily identifiable, and adds housekeeping features to the data which, by standard-
ising the input and output routines, allow the use of relatively easy commands, like
INPUT# and PRINT#. Housekeeping with CBM disks includes such features as the
special treatment of [Return] as a record separator, the special treatment of the comma
and colon as 'field' separators, and the automatic generation of header or directory
records in which the file's name, type, starting address or buffer position are stored
for later recovery. When the file is read back, this information is used to process it
correctly. For example, programs and data are held rather similarly, as bytes in
blocks on tape or disk, but a program has a different value in one of its header bytes
which causes the operating system to carry out the load routine into RAM, rather than
prepare a data buffer for reading, which is what happens with a file. At first sight,
this seems extremely involved, but with practice and thought it soon becomes fairly
easy to guess what parameters will be needed to make a file system operable. Large
computers use a variety of file organisations, many of them unavailable on small mach-
ines. (In fact, people who have worked for years with mainframes or large minicomp-
uters are often unable to understand the difficulties of working with the more restrict-
ed operating systems of microcomputers). The summary that follows describes these
file organisation methods. All of them can be implemented with CBM machines in prin-
ciple, but learners will be well advised to use only those systems that are supplied
by Commodore, or which are available as extras with the use of other manufacturers'
firmware. The names of the methods aren't standard, so I've referred to CBM file
types in block lettering in the hope of reducing confusion.

Sequential files.   These files are one of the simplest types in concept. The only file
construction which is simpler is (I suppose) simply a file containing consecutive bytes
of data with no special characters, a long list of data with a method to indicate where
it ends. Most microcomputers, apart from the very cheapest with no file-handling at
all, implement this system. It is very suitable for tape storage, because tape reading
and writing is almost invariably linear because of the difficulty of winding tape at high
speed to read different records. It is also suitable for records of variable length, be-
cause there is no need, as there is with some other methods, to ensure that all the
records are the same length. CBM, and most other micros, use the system in which
the [Return] character acts as a record separator. A 'record' may be made up of
'fields'; the record is a complete entity, perhaps name, address, and several other
details, in which case there is a 'field' corresponding to name, address, and so on.
In practice each record is usually designed with the same number of fields to each
record, so that every record can be read in the same way. COBOL is particularly
well-adapted to explicit handling of fields and records and file definitions like this one
appear at the start of all COBOL programs:

```
01    LOGFILE-HEADER.
      03    DOCUMENT-NUMBER     PIC 9(4).
      03    VDU-NUMBER          PIC 99.
      03    VDU-RETURN          PIC 9(4).
      03    TRANS-STATUS        PIC S9(4).
      03    TRANSACTION.
            05 TX-NAME          PIC X(10).
            05 TX-ID            PIC X(2).
```

This means that the record called 'LOGFILE-HEADER' is 27 bytes long, made up of the
fields named as above, and with the format specified, where 9 means a numeral, S a
sign, and A an alphanumeric character. (This notation has been used in CBM printers
with little change). BASIC has carried over from FORTRAN the habit of not defining
files at all rigorously. Rather than use the formatted layout of the type above, BASIC
separates records by [Return] and fields, where the distinction is kept, with commas
or colons. This is because PRINT# always sends a [Return] at the end of its current

output string. This is sent to the tape or disk, and stored there, so it may as well be put to use as a separator. Similarly, INPUT# is designed to take in a set of characters up to the next [Return]. If this set of characters includes one or more commas or colons, an input statement of the form INPUT#x, X$,Y$,Z$ will assign X$, Y$ and so on to fields within the record. All this is quite straightforward (when you've grasped the idea!) and section 6.3 has demonstration programs. There is one complication peculiar to Commodore: PRINT and PRINT# each follow [Return] with linefeed (this is ASCII character 10 decimal), originally so that the next line on the screen would be moved to, whenever Return was pressed. This character is filtered out of cassette tape files but, with BASIC<4, left to print to disk. To get rid of it a construction like

```
        PRINT#8,X$;CHR$(13);   or   PRINT#8,X$CR$;
```

had to be used. BASIC 4 has a patch in its print routine which deletes this character if the file number is less than 128. For chapter and verse, see PRINT and PRINT# in Chapter 5.

The diagrams which follow are an attempt to explain the layout of sequential files. I have used [R] to mean Return, which is #0D (13 decimal). The first shows a file of this type in which only records have been written (i.e. there are no subdividing fields within a record).

```
                           1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3
Byte: 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
File: J O H N [R] S M I T H [R] A L A N [R] C A R T E R [R] E V E [R] B A R O N [R]
```

In this case, a program to read the file will have input statements of the following type within it at some stage:-

```
        100 INPUT CN$: INPUT SN$
        200 PRINT "FULL NAME IS "; CN$; " "; SN$
```

This format is obviously dictated by the structure of the file, where Christian names and surnames alternate. The second example shows a file where separate fields have been used, a technique which calls for the use of statements like PRINT#x,I$ "," N$ "," M:-

```
      <— Record #1 —>    <— Record # 2 —————>    <—Record # 3.——>
File: C . , B R I S K , 5 5 [R] D . F . , B R O A D , 1 5 8 [R] M . , B R O D Y , 7 [R]
```

Here, because of the separators, we have the option of using either of the following types of input statement to read the data from the file. Note that again the file is written in a regular way, with equal numbers of fields in a record, so that any record may be processed in the same way as all the other records. This is not necessary, but it does simplify programming. Alternative techniques include the use of a number in the first field which counts the total number of fields in that record.

```
        100 INPUT I$, SN$, M : PRINT "NAME, SCORE = " I$ SN$ M
or      100 INPUT I$: INPUT SN$: INPUT M
```

CBM equipment is designed so that PRINT and PRINT# send to tape or disk the same characters as they would have sent to the screen, so that if PRINT and INPUT match there should be no problem. The last example, where two strings are followed by a number, illustrates this. As long as I$, N$ and M are written to the file, they can be read back by the same variables. There is a passage in Osborne-Donahue (p.300 approx.) which seems to suggest, erroneously, that there is some difference between number and strings in this respect.

Relative files. These are sometimes called 'random access' files. Each record is the same length; any record can be called by number, without, as with a sequential file, having to wait while the entire file is read from the start. The organisation of the records is identical to that for sequential files, except that each record is the same length, or at least not longer than the predefined length of a full record. (Some may be exceptionally short, but as long as the number of fields is correct, an early [Return] character will not disturb the file). However, in addition to this file, there must also be a subsidiary file which enables the position of any record to be computed. CBM equipment with DOS 2+ uses a chain of so-called 'side-sectors' for this purpose. DOS 1+ has a very long program to achieve the same effect with 'User files'. The algorithm which determines the data position may in fact be external: using the sector-writing capability of many disk-drives makes possible the construction of files in which the nth. record is simply the nth. sector of the disk. Record number 0 might be track 0, sector 0; record 1 in sector 1 of track 0; record 2 in sector 2 of track 0; and so on.

<u>Direct Access Files</u>. This file access system, like the last, is sometimes called random access, because a file on the disk can be read or written 'at random'. It is a version of relative access in which records are not called by number, but by some key. This key is encoded, converting it into a record number, and the resulting record read. To clarify this, suppose we have the facility to write relative files (e.g. via DOS 2+), but we wish to be able to retrieve records using telephone numbers (or names, or part numbers ...) as the key. One method might be as follows: store in RAM a table of each phone number, in order. Then when an enquiry is to be made, this table could be searched, using perhaps a binary chop search, to convert the phone number into a number within the range of record numbers on file. This would be quite fast. The drawback is that new records could only be inserted into the file by rewriting all the file above the new record, moving it up one place. An alternative approach, direct access codes the key using a 'randomising algorithm', positioning the record according to the resulting value. Records are held in a completely jumbled sequence, but the point is that the algorithm enables any of them to be quickly located from the key, without any disk overhead. A file of this type must be larger than the anticipated number of records; at least 30% more space must be provided so that new records can be placed without too much difficulty. Suppose we open a direct access file with space for 1500 records of length 50 each, anticipating about 1000 records in the complete file. We devise an algorithm which converts any telephone number into an integer in the range 1-1400. (The extra 100 allows for 'consecutive spill' forward). A good algorithm will spread the resulting numbers evenly. We may be able to improve on this using known facts about distribution of such numbers; if for example the final digits are evenly spread, 1400/9 * final digit will ensure that ten equal chunks of data are produced by the algorithm. Another expression evaluating to 0-150 or so must be added to give the randomising formula. Other methods include: taking the remainder after division by a prime number; using RND(key) after RND(-1) to generate repeatable random decimals from 0-1; splitting the key into parts and adding.

The outcome of this will be the sort of situation shown in the diagram. Three telephone numbers have been processed by our algorithm, and yielded the values shown. The records are therefore written into the file:

765-4321 becomes 752;   741-0123 becomes 53;   300-3000 becomes 297:

FILE:	Ø	53	297		752		1500

Now, when we wish to read the record corresponding to 765-4321, we perform the same encoding process, and read record 53. What happens if an algorithm generates synonyms (strictly, the original keys are 'synonyms'). We store it further up the file, as near as possible to its originally computed position. This implies that each record must store its key as well as the associated data, or have some other means of distinguishing a 'home key' from a 'synonym'.

If all the records are stored in the file as they are processed, and the algorithm is truly 'random', the proportion of synonyms expected is half the packing density. In our example, 1000 of 1400 records, about 71%, will be utilised. So about 35% of keys will be synonyms. If synonyms aren't entered immediately, but are stored in another file for later entry, the proportion of synonyms drops by about 25%. So, in this case, about 25% of keys will be synonyms; but the number of records which need to be read when a synonymous record is read is higher.

This is a fascinating system on which to organise files, and is quite an easy one to implement. It has the serious drawback of making a sorted sequential read of the file difficult, because of the randomised order. To do this, another file, holding all the keys to date, is required. This will need to be sorted or merged at intervals. Then it can be read sequentially, and the corresponding records calculated and read. Another drawback is the wasted space, a necessary concomitant of the technique.

To summarise: if this look-up method appeals to you, follow these steps:
(i) Decide whether it will, in fact, do what you want.
(ii) Experiment with 'randomising' techniques, taking account of regularities in the key field. Find one with a good spread.
(iii) Make an estimate of the optimum file size.
(iv) Enter the most frequently used records first; many will become 'home' records, faster to retrieve and rewrite.
(v) Don't write synonyms until the second pass, to maximise the number of 'home' records which the file will contain.

Indexed sequential files. IS or ISAM files are one of the most popular types of file in commercial use. They are unobtainable on most microcomputers as standards, although some firmware (e.g. EPROMs) has been produced and advertised with this type of feature on it. These files are readable sequentially in sorted order - for example by name or by customer number, but in any event by a range of possible keys. Any individual record is also readable or writeable at will. This can be accomplished only with subsidiary files, holding keys and pointers to their records. Any new record needs to be merged and sorted into this index, perhaps on five to ten different keys. A self-maintaining system of this kind is not easy to write. Some small computers - the IBM 5100 series, and 'Microstar' - do have this facility or something like it.

The example which follows is a small-scale version of the real thing, but shows the general principle. I've assumed a rather short file is set up, which is accessed by a single key, expressed as two letters. The index file, which has the same number of records, stores each key used so far, together with a pointer. Now, when the file is to be read in sequence, this index file is read first: record AA is to be found in the main file as record 1, then record AB as record 2, and so on. If a record is to be read at random, say DG, the index is searched, perhaps by a binary chop, since it's in sequence, and when DG is found, its pointer is read and the corresponding main file record read. This of course is slower than any non-indexed method which reads its main record directly. Relative records for example are faster to read, but can only be regarded as 'indexed' if for example a client number equals his record number.

	Record #1	Record #2	Record #3	Record #4
MAIN FILE:	AA detail	AB detail	DG detail	QR detail

INDEX FILE: | AA1 | AB2 | DG3 | QR4 |

Suppose two new records, ZZ detail and FF detail are to be entered into the indexed sequential system. First, we append the new records onto the main file as it exists at present. This expands the main file like this:

	Record #1	Record #2	Record #3	Record #4	Record #5	Record #6
MAIN FILE:	AA detail	AB detail	DG detail	QR detail	ZZ detail	FF detail

The index file needs two records added to it, viz. ZZ5 and FF6. These, however, cannot simply be appended to the index file, since then the sequence of records is no longer maintained. Instead, they are sorted, and merged into the index file. (The sort-merge is a universal technique for incorporating new data into an already sorted file: after sorting, each record is compared with the file records until a position is found into which it fits the correct order). The resulting index file is:

INDEX FILE: | AA1 | AB2 | DG3 | FF6 | QR4 | ZZ5 |

This, of course, is only an outline of one method of many by which this system can be put into operation.

Inverted files. This file organisation method is used in data-base processing, where a file of data is read and the entire file is potentially a source of information; the data base is interrogated using keys, which typically may be combined using AND, OR, NOT and other Boolean operators. The object is *not* to read an entire file, checking on each single record, but instead to cut down the number of possible records as rapidly as possible. This means the construction of a number of key files. Every new record is entered in the main file, and its record number has to be appended, or preferably merged, into each key file for which its details are relevant. This process is likely to be quite slow, but increased efficiency subsequently may make it worthwhile. The example, I hope, speaks for itself.

MAIN FILE: Record # | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 ...

PRICE RANGE 1: | 1 2 3 8 15 16 20 22 33 54 56 ...
PRICE RANGE 2: | 4 6 7 12 13 17 18 24 26 27 29 ...
PRICE RANGE 3: | 5 9 10 11 14 15 19 21 25 28 30 ...
AREA 1   : | 5 6 10 11 13 15 18 24 25 26 ...
AREA 2   : | 3 4 7 8 9 12 20 21 22 23 33 ...
AREA 3   : | 1 2 9 14 16 17 19 27 28 30 ...
ROOMS TYPE 1 : | 4 5 7 8 12 14 16 17 18 20 ..

Direct track-and-sector access methods. Many very efficient systems for the use of disk data are hybrid systems in which data is loaded into a buffer and processed there. An entire track of data may be loaded into memory and searched while in RAM, for example, A sector may correspond to several records, which can be poked into the sector and written back to disk. Searches for Boolean matches can be carried out with machine-code routines on data in RAM. It is possible to save the entire set of BASIC variables on disk and reload them later; when used with integer arrays this can be a very powerful method for storing a great deal of arithmetic information in RAM, where recall is fast.

'Opening' and 'closing' files   The following statements are taken from a variety of computers. They all open one file. Through the idiosyncrasies of syntax a few principles underlying this process may, I hope, shine through:-

```
OPEN FILE FL2,'D40',3,'MASTIND',IN,KEY
OPEN 1,1,1,"TAPE SEQ.RECORDS"
PRINT D$"OPEN MAIL LIST,L200,DO"
SELECT LOGFILE    ASSIGN TO EDS    ACCESS IS SEQUENTIAL
DOPEN#6,"REL DATAFILE",D1,
```

All these include:
(i) Some future reference for the file - a name or number.
(ii) Request to use the file for writing, or reading, or both.
(iii) Reference to the device or drive to be used, unless this is implicitly given.
(iv) Description of the type of file, at least if the file is a new one.
(v) Some internal system to assign a buffer for storage of data, which has been read from the device but is currently stored awaiting processing; 'or which is to be used to store data before it is written to disk; or both.
(vi) An implicit requirement that the file be closed at some future time, whereupon the final buffer of a write file can be processed, and the directory or header details of the file updated.

6.3 Commodore disk drives and file handling

The 2040, 3040, 4040 and 8050 drives   At the time of writing, these drives are the most widely available CBM drives. The 8060, using 8 inch floppy disks and IBM format, and a winchester unit with a single drive, have been announced, but are not widely available. A single, shoe-box sized drive for the VIC, called the 4020, also exists. (Note: CBM's disk units are assigned numbers ending with 0, except the 8061 and 8062 where single and double sided disks are distinguished. The -0 suffix contrasts with the CBM machines, where -08, -16, or -32 records the RAM installed at the time the computer is sold. Most printers end with -2. Clear? It all started with 2001...)

The four drives are similar in appearance.* The 2040-4040 sequence represents continued improvement within the limitations imposed by single-sided, single-density 5 1/4 inch disks. Several physical changes have been introduced: the 2040, of which there are a number still around, was notoriously prone to heating and other problems. It is not particularly easy to upgrade, as its internal main printed circuit board has to be changed. The 3040 avoided some of the problems of its ancestor, while retaining others: for example the diskette clamp seems to lack the normal precision centring mechanism, so that it is widely recommended to close the disk drive door only when the disk is revolving. Initialisation of the disk (see INITIALISE in Chapter 7) is still obligatory in this model. Finally, the 3040 used DOS 1.2, which, while an improvement on DOS 1 in its error-trapping, still only provided sequential files! All these objections were removed by DOS 2.1 and the 4040 drives. The remaining difficulties are summarised in section 6.8 at the end of this chapter. The 8050 has more than twice as many tracks, and about 40% more sectors, on the same size disk. The diskettes therefore are entirely non-interchangeable. The disk doors and disk retaining mechanism are different, presumably better, and the central warning LED signals green and red - not just red! No doubt there are production-line changes of a highly technical type which are not widely known. This then is the evolutionary history of CBM's disk drives to date.

---

*The steel casings are made by Canada's largest barbeque-equipment factory. If there is any symbolic significance in this I've been unable to find it.

The drives were issued contemporaneously with the 2000, 3000, 4000, and 8000 models of the PET/CBM respectively, and are therefore often seen together, because they were bought that way. As mentioned before, the capabilities of the disk drive lie within the disk unit itself, so it is more important to select the correct disk unit - or to be precise the correct ROMs - than the appropriate version of BASIC. The single most important feature of a system is the storable amount of data; if a user wants a program to access 2 million bytes, obviously the programmer will be in trouble with a small capacity disk unit unless space-saving data-packing techniques are used.

How can the different drives be distinguished? The obvious way is to read the Commodore labels on them, if they aren't obscured by dealers' logos, but this is not always reliable, because of the possibility of updating the ROM sets in the 3040 and 8050 series. For example, a 3040 unit may have been upgraded from DOS 1.2 to DOS 2.1 . The infallible way is to peek the disk ROMs themselves; a location which has the required power of discriminating between disk ROMs is $FFFF in the IRQ vector. Thus

```
OPEN 15,8,15: PRINT#15,"M-R" CHR$(255)CHR$(255): GET#15,X$: CLOSE 15:
PRINT ASC(X$)
```

Prints the decimal value of this location, which = 213 with 4040 drives, and 242 with 8050 drives. At the time of writing, there are at least two DOSs for the 8050; these can be identified by CHR$(195)CHR$(251), which returns 32 with DOS 2.5 and 170 with the newer DOS 2.7 .

And this table summarises the main characteristics of CBM drives:

DISK UNIT	Tracks	- - - CAPACITY - - - Total bytes	Free bytes	DOS fitted	Upgrade-able?	Relative files?	Initialise needed?	Diskettes compatible?
[2040	35	2×176 640	2×171 520	1	Yes (with pcb change)	No°	Yes*	Yes][2] ↑
3040	35	2×176 540	2×171 520	1.2	Yes (to 4040)	No°	Yes*	Yes read only ↕ ↑
4040	35	2×174 748	2×169 728	2.1	No	Yes	No	Yes, but read only[3]
8050	77	2×533 248	2×518 400	2.5 +	No	Yes	No	No

°A very long demonstration program may be used to construct files like these; or user written direct-access techniques may be used. Neither comes with DOS.

*When initialising a disk, if the disk doors aren't closed while the disk is spinning, to help centralise the disk, the directory may not read, and you will have to try again. The message 20,READ ERROR,17,0 is typical.

[2]This drive is the oldest and least reliable.

[3]When (say) a 4040 writes to a disk formatted by DOS 1.2, an error message like this will occur:

       73,CBM DOS,19,07

and the disk will become unreadable. The same thing happens if DOS 1 writes to DOS2. Recovery techniques are known (e.g. Harry Broomhall has programs called 'Lazarus') but they are hazardous and success cannot be guaranteed. Therefore, *take care not to write with DOS 2+ onto DOS 1+ disks, or vice versa*. Sometimes this happens out of the blue. A CBM games disk has a concealed game, which only appears if a score on one other game exceeds a certain value. To cause this to happen, the directory is modified on the disk, possibly damaging it.

CBM file types: Sequential, program, relative, and user A directory or catalog of any CBM diskette (see Chapter 7, CATALOG) lists files on disk with a three-letter code to indicate the file type. The possible codes are SEQ, PRG, REL, and USR. Occasionally DEL (deleted) makes an appearance, and sometimes *EL or other anomalous code. These latter are connected with the problems of scratching unclosed files, which are discussed elsewhere, and with bugs in the COPY command as applied to relative files. Apart from malfunctions of this sort, the four file types (or three in DOS 1, which has no REL files) are recorded in the directory at the time the file is written, as a flag: if the flag is #$00, a scratched file is (or was) present; if it is #$81, #$82, #$83, or #$84, the directory translates the value into SEQ, PRG, USR or REL respectively. A file opened as a sequential file lists on the directory with SEQ; a BASIC program saved to disk lists with PRG, not surprisingly. Machine-code also lists with PRG, so there is no way to tell from the directory whether a PRG is BASIC or machine-code. Saving with names like 'OLD.033A' is the usually recommended practice for the meticulous programmer. As we'll see, program files can be opened for reading and writing as though they were sequential. This is useful in the compilation of certain types of utilities for programs, such as cross-referencers for variables.

   User files (USR) are, so far as I'm aware, identical to sequential files. Their sole purpose seems to be to give the impression that the subsidiary files which are opened by the relative file demonstration program in DOS 1, are notably different from the main data file, and from other sequential files on the disk.

   Section 6.4 of this chapter explains how each file type is stored on the disk. It is not necessary for comparative beginners at programming to understand minutiae of this sort, and the remainder of this section explains file-handling from BASIC.

Files and BASIC: (i) Formatting new diskettes A box of diskettes which haven't yet been used ought to be dealt with in a standard way, so the status of any diskette is fairly self-evident. CBM disks may be formatted with a two-character i.d. It is a good idea, in principle, to ensure that each disk has a different i.d., so there will be no chance of DOS garbaging the data on a disk by confusing it with another of identical i.d. This will only be possible with backup disks. Typically, the labels supplied by the disks' manufacturers will be stuck to the disks and filled in (with a felt-tip pen!). The process of writing a name onto a blank disk, and recording i.d. markers on all its tracks and sectors, is usually called 'formatting'. Without it, a diskette cannot be written to or read from. The pattern is characteristic of a particular disk unit; most disk drives can't read disks written by other brands of machine, because the number and position of the tracks and sectors is not the same. When carried out on a diskette which holds data, the rewriting process can be considered to erase all the previous information, so it is rather important to take care with it. Just to be confusing, this is often called 'initialising', the name CBM use to refer to the process of reading a disk's directory and BAM into memory, a non-destructive operation. If you switch from one computer to another, it may be necessary to remember this fact; 'initialise' may delete all your data if tried on another machine. Conversely, in CBM BASIC, HEADER or the non-BASIC 4 command Disk NEW, format disks. These are discussed in Chapter 7 under HEADER, but here are four examples of the commands.

   The two first examples have the effect of formatting an entire disk, giving it the name DISK RW and the i.d. V6. HEADER can only be used by BASIC 4-earlier BASICs don't recognise the word. But Disk NEW can be run with any BASIC.

```
HEADER DØ,IV6,"DISK RW"          :REM DISK ASSUMED TO BE IN DRIVE Ø
OPEN 15,8,15: PRINT#15,"NEWØ:DISK RW,V6"
```

The next examples retain the previous i.d. and simply reformat the directory. This does *not* format the entire disk.

```
HEADER DØ,"DISK RW"          :REM DOESN'T FORMAT THE WHOLE DISKETTE
OPEN 15,8,15: PRINT#15,"NØ:DISK RW"
```

Files and BASIC: (ii) Channel 15, the 'error' channel Channel 15 - secondary address 15 in an OPEN statement - is specially reserved for use by IEEE equipment, of which CBM disk drives are an important example. Its function is to act as a buffer for DS and DS$ messages about the disk status. (See Chapter 7 on DS and DS$ for more on these reserved variables). Most of these messages aren't really 'errors', but the name is a convenient one to use. It also enables commands to be transferred from BASIC, such as PRINT#15,"NEWØ:DISK RW", as we've just seen. With BASICs before BASIC 4

programmers had no option but to get in the habit of opening this channel and print-
ing to it or reading messages from it; BASIC 4 has automated this process, so that it
needs to be done only at the start of a session. Chapter 7 lists all the BASIC 4 comm-
ands which perform disk file operations, along with their channel 15 equivalents, all
of which, incidentally, work in BASIC 4 too, so that it is possible to write disk-file
handling programs which will run on any version of CBM BASIC. When reading Chap-
ter 7, remember that the statement OPEN 15,8,15 is assumed to have been executed,
to open file 15 to the error channel. Other file numbers are often used in the literat-
re, for example OPEN 1,8,15 which is followed by PRINT#1 or INPUT#1. It makes no
difference, except that consistency is helpful, and 15 is mnemonically good.

Files and BASIC: (iii) DOS support and DUM

These programs are supplied on the
test disks which Commodore issue for use with their machines. The first is machine-
code, with a BASIC loader. Its full title is 'Universal DOS Support' but this is too long
for the disk directory, so it appears instead as 'Universal Wedge'. Its purpose is to
extend BASIC's direct mode to include several disk handling commands. These include
initialisation, sending a directory directly to the screen (leaving BASIC intact), and
loading disk programs. BASIC 4 needs this program far less than BASIC<4, because
many of its commands already handle PRINT#15 automatically. Chapter 7 notes when
DOS Support is useful, for example when reading a directory with BASIC<4. The
symbols @,>,↑, and \ are intercepted by DOS Support from BASIC. Since some of them
are also valid in BASIC expressions, DOS Support has an elaborate built-in routine to
ensure that direct-mode commands are accepted, but program-mode is rejected. (Old
versions may have this test missing). Although I haven't repeated a comment on DOS
Support in every command in Chapter 7, every command listed there of the form
PRINT#15,"..." may be simplified with DOS support, in direct mode. For example,

        `@NØ:DISK RW,V6`

formats a new disk with the name 'DISK RW' and i.d. 'V6'.

    DUM is a BASIC program by R Leon of Prominico Ltd., Vancouver, which carries
out disk maintenance for people who haven't puzzled out the operation of CBM disks,
or who like a program to run things. This again was more necessary with BASIC<4
than it now is with BASIC 4, which has easier commands. Nevertheless this utility, or
others like it, remains valuable because of the effortlessness it brings to disk handling.
It operates in direct mode only, and is not a file-handling utility. Instead, it prompts
the user, with a menu, to choose options like 'Copy', 'Backup', and 'New', which carry
out these operations only after asking the operator to check that the relevant disks
are correctly in place. This, of course, reduces the chances of a blunder. The pro-
gram includes a special feature, a 'history file', which is a sequential file called 'DISK
DATA' or something similar, and which stores several dates, for example the date of
the last backup, and comments. 'Filemaster', by L Sasso, is a newer disk utility.

Files and BASIC: (iv) PRINT# and INPUT# and GET#

These BASIC commands send
output to a file and read it back, either as a batch or characters (INPUT#) or as
individual characters (GET#). We shall see in the specimen programs how these comm-
ands operate with each type of file. Meanwhile, in outline, the important features of
them are as follows (more detail is given in Chapter 5 about each of these BASIC key-
words).

        PRINT# outputs strings, variables, expressions and literals to the file in the
            same way that the output is sent to the screen. For example, PRINT#8,
            "HELLO"; X; Y$; 23+34 sends HELLO, the current value of X, the string
            Y$, and the number 57 to the same line of the screen. However, it also
            sends a carriage return + line feed at the end of the line, which is why
            the cursor is now positioned at the beginning of the next line. This is the
            major tricky point about PRINT#. BASIC 4 contains a patch which avoids
            sending a linefeed character if the file number is < 128. If the file number
            is 128 or more, BASIC 4 behaves like BASIC<4, and the resulting records
            on file will begin with linefeed characters. This is not a disaster; it means
            only that the records will mysteriously print one line below their expected
            place, and will be one character longer than expected. The cure is to use

        `PRINT#8,"HELLO";X;Y$;23+34;CHR$(13);`
or   `PRINT#8,"HELLO";X;Y$;23+34;CR$;`   `:REM WHERE CR$=CHR$(13)`

when using BASIC<4. The same trick may also be used with BASIC 4.

INPUT# behaves in a very similar way to the screen input statement. It is a simple command to use, provided only that the programmer remembers to match the format of PRINT# with that of INPUT#. Often, of course, this happens automatically, because a programmer will naturally tend to use identical variable names when writing to a file with PRINT#, as when reading back the same data. I have explained this before in section 6.2, with reference to sequential files; it is also explained in Chapter 5 under INPUT and INPUT#. For those new to programming, the point to understand is that PRINT# and INPUT# are mirror-image commands - what one of them writes, the other will read. Generally, it is not necessary to know about the special characters which make this possible, beyond being careful with, or avoiding altogether, the use of commas and colons.

GET# reads individual characters from a file, including all the special characters like quote marks (ASCII 34), carriage returns (ASCII 13), linefeed characters (ASCII 10), plus all the punctuation symbols and screen editing characters which CBM machines have at their disposal. This makes it far more versatile than INPUT#, *if* you are interested in the complete contents of a file. If you're not, the 'intelligence' of INPUT#, which assigns variables for you, is a better command.

**Files and BASIC: (v) The status variables ST, DS and DS$**   This last subsection of our summarising trot provides a brief revision (or prevision) of the functions of the status variable ST and the disk status variables, given the names, in BASIC 4, of DS and DS$. The method of operation of ST is outlined in Chapter 5 under the heading ST. (Strictly, it's not a reserved word, but that chapter seemed the best place for it). DS and DS$ are described in Chapter 7; they are not, strictly speaking, reserved words either. What is the purpose of these variables? The difference is quite subtle. ST is concerned with input/output processing from the PET/CBM's point of view, so if a device isn't there, or doesn't respond correctly, ST becomes changed from its initial value of zero to 1,2,4,8,16, ... depending on the error condition. Thus, eight different conditions at most can be signalled by ST. The most used in practice is probably the end-of-file condition. ST = 64 signals that the computer has not received a byte from the peripheral, so the end-of-file flag in ST is set, on the theory that the programmer will check this and do something about it. It is always possible to write one's own end-of-file markers. In commercial computing, terminal records containing say ****T are used. When this record is read, the file is closed without attempting to read further. However, because of the possibility that a file isn't correctly closed, in which case the marker will be absent, the use of ST is still useful, particularly with other peoples' files.

    DS and DS$ are generated internally by the disk DOS, and are only available to the computer when specially read. In BASIC 4 this is easy: commands of this type

```
      10 INPUT#5,X$: IF DS>19 GOTO 50000: REM 50000 PRINTS ERROR MESSAGE, AWAITS ACTION
or   100 INPUT#8,X$: PRINT DS$: REM CHECK UTILITY BY PRINTING DISK STATUS EACH TIME
```

mean that the status of the disk unit after performing its operation can be readily checked. Note that DS, the error number, which equals the first numeral in DS$, can equal 0-19 without being counted as an 'error' - see the table under DS$. BASIC<4 is more trouble; in program mode, a subroutine of this sort must be used:

```
10000 OPEN 15,8,15: INPUT#15,X: IF X<20 THEN CLOSE 15: RETURN
10010 INPUT#15, Y,ER$,Z: PRINT X "," Y "," ER$ "," Z
10020 PRINT "DISK ERROR***": END
```

and in direct mode the subroutine may be called, or this line entered:

```
oP 15,8,15: iN15,e,e$: ?e,e$
```

where I've used standard abbreviated forms of the commands to ease the effort of typing them in at the keyboard.

    This checklist of points which are relevant to CBM disk files may seem rather daunting, and I suppose actually *is* rather daunting! However, it is a fact that these disk drives are no more difficult to program than many others. The short demonstration programs in the next section should enable anyone with enthusiasm to get the feel of these various commands and practical requirements. Longer demonstration routines are available on Commodore's demonstration disks; I have tried to keep these short so

that they may be keyed in without too much effort.

Demonstration programs: (i) **Sequential files**

DEMONSTRATION OF SEQUENTIAL FILE - WRITING TO DISK. (BASIC 4)

```
 5 SCRATCH "SEQ FILE",D1
10 DOPEN#1,"SEQ FILE",D1,W
20 FOR J = 1 TO 10
30 X$ = "RECORD NUMBER" + STR$(J)
40 PRINT#1,X$
45 PRINT X$ DS$ ST
50 NEXT J
60 DCLOSE
```

This specimen program writes 10 records, which consist of 'RECORD NUMBER 1' through 'RECORD NUMBER 10'. They are held in a file called 'SEQ FILE' which is on drive 1; I've assumed a test diskette is loaded into that drive. Drive 0 is, of course, just as good! The program works in this way:

Line 5 erases the previous file (if any) of the same name. The program can thus be run repeatedly without ?file exists error. Alternatively, line 10's file name in quotes can be preceded by '@', which opens the file, replacing any previous file as though it were scratched. This construction may be risky to use.

Line 10 opens a disk file, on drive 1, for writing. It is a sequential file called "SEQ FILE". How is the Disk Operating System able to know the file is not relative? As we'll see, a newly created relative file has a length-of-record parameter, which is absent here. So a sequential file is assumed, and 'W' tells the system that it is open for writing. So the necessary buffers are opened in the disk's internal RAM, the name is recorded in the directory, and pointers are set which will enable the new file to be PRINTed to, in sequential order.

Lines 20 - 50: the loop, with its variable J, controls the disk write operation. The figures in the example cause 10 records only to be written. Line 30 assembles an individual record, X$. It's exact form is not important to the demonstration, but I've made each record different from the others, so that on reading the file it's easy to check whether the records are, in fact, in the right sequence.

Line 40 prints X$ to file number 1, which was the number assigned in line 10 to our file 'SEQ FILE'. Just as though the record were printed to the screen of the CBM, a carriage return follows PRINT#1,X$, so the records are correctly separated. (If line 40 is rewritten PRINT#1,X$; with a semi-colon, carriage return is not sent, and the records will be concatenated in a long string. The result will be too long for INPUT# to cope with; but GET# will successfully read the string character by character).

Line 45 is part of the demonstration, and would not normally appear in a finished program, except perhaps a utility routine to check the operations involved in file handling. It prints the record, the disk status string DS$, and the CBM status variable ST to the screen, where they appear in ten rows. These rows should be practically identical, showing DS$ as 00,OK,00,00 and ST as 0, only X$ varying between its limits of RECORD NUMBER 1 and RECORD NUMBER 10.
NOTE: Channel 15, to read DS$, is opened by the system, and need not be explicitly used in a program run by BASIC 4.

Line 60 closes file(s). DCLOSE#1 in this example has the same effect.

DEMONSTRATION OF SEQUENTIAL FILE - READING FROM DISK. (BASIC 4)

```
100 DOPEN#1,"SEQ FILE",D1
110 FOR J = 1 TO 11
120 INPUT#1,X$
125 PRINT X$ DS$ ST
130 NEXT J: DCLOSE
```

Line 100 opens "SEQ FILE", on drive 1, for read, as 'W' and 'L' are both absent. Lines 110-130 perform a loop which inputs records. Each is printed to the screen, with both status variables. Note the effect on DS$ and ST of reading an '11th record'.

DEMONSTRATION OF SEQUENTIAL FILE - WRITING TO DISK. (BASIC<4)

```
 5 OPEN 15,8,15: PRINT#15,"SCRATCH1:SEQ FILE"
10 OPEN 1,8,2,"1:SEQ FILE,SEQ,WRITE"
20 FOR J = 1 TO 10
30 X$ = "RECORD NUMBER" + STR$(J)
40 PRINT#1,X$; CHR$(13);
43 S=ST
44 INPUT#15,E1,ER$,E2,E3
45 PRINT X$ E1 "," ER$"," E2 "," E3; S
50 NEXT J
60 CLOSE 1: CLOSE 15
```

This program has the same effect as the earlier (BASIC 4) version to write to disk, and sends the same information to the screen. BASIC 4 can run this program, or the other, rather simpler, version, but BASIC<4 cannot run that version because it is not equipped with the disk command keywords. The line-numbering in each program is similar; lines 43-44 above are concerned with (i) saving ST, (ii) reading the four messages which correspond to the parts of DS$. ST could be printed in line 43; the sole reason for preserving it till later is to format line 45 in the identical way to that of the other line 45.

Line 5 opens the command channel, and sends a 'scratch' command to delete 'SEQ FILE' from drive 1. The abbreviation PRINT#15,"S1:SEQ FILE" is equally correct.

Line 10 opens 'SEQ FILE' on drive 1 for write. Note that the secondary address may be any value from 2-14 which isn't yet allocated. Again, the abbreviated form OPEN 1,8,2,"1:SEQ FILE,S,W" is as good (and corresponds more accurately to what is sent on the IEEE bus to the disk). BASIC 4 sends the same messages as BASIC<4; the syntax is easier because some of the operations, like opening the command/error channel, and finding an unused secondary address, are built into BASIC 4.

Line 40 illustrates the anti-linefeed manoeuvre necessary with BASIC<4. The character with ASCII value 13 is, of course, carriage return.

Lines 43-45 have the same effect as PRINT X$ DS$ ST in spite of their more formidable appearance. The four 'error' parameters are read from the 'error channel'.

Line 60 No DCLOSE exists in BASIC<4, so all the files must be separately closed.

DEMONSTRATION OF SEQUENTIAL FILE - READING FROM DISK. (BASIC<4)

```
100 OPEN 15,8,15:  OPEN 1,8,2,"1:SEQ FILE,SEQ,READ"
110 FOR J = 1 TO 11
120 INPUT#1,X$
123 S=ST
124 INPUT#15,E1,ER$,E2,E3
125 PRINT X$ E1 "," ER$ "," E2 "," E3; S
130 NEXT J:  CLOSE 1: CLOSE 15
```

Again, this program is identical in its effect to the BASIC 4 version.

Line 100 opens the command channel and also opens file number 1 to the sequential file 'SEQ FILE' on drive 1 for reading. The secondary address, and device number, are chosen subject to the same restrictions as outlined above in the paragraph on line 10. Device number 8 has been assumed throughout.

The screen appearance of the file as it's read should, as in BASIC 4's version, consist of 11 lines, the first ten made up of 'RECORD NUMBER1' to 'RECORD NUMBER 10', each followed by disk status values 0,OK,0,0 and CBM status of 0. The very last record (10) has ST set to 64, which shows that a record is the last record in the file. The attempt to read beyond the end of file has effects which vary slightly with the system in use; BASIC 1, for example, appears to return the last record, whereas later BASICs return the carriage return character instead.

Demonstration programs: (ii) Relative files

DEMONSTRATION OF RELATIVE FILE - WRITING TO DISK. (BASIC 4 AND DOS 2+)

```
1 REM NOTE "W" FOR SEQUENTIAL WRITE ONLY; GIVES ?SYNTAX ERROR WITH REL
2 REM NOTE OPEN FOR BOTH READ AND WRITE IF RELATIVE FILE
3 REM NOTE USE OF 'RECORD#', AND ITS SYNTAX
4 REM NOTE GET ERROR 50 DURING WRITE AS EACH BLOCK OF 256 BYTES IS USED
5 REM NOTE LENGTH OF 21; RECORDS ARE 20 LONG + CARRIAGE RETURN
6 REM NOTE 'RANDOM' ORDER OF READBACK IN LINE 210
7 REM NOTE IF LINE 220 IS OMITTED, FILE WILL BE READ SEQUENTIALLY
10 DOPEN#2,"REL FILE",D1,L21
20 FOR J = 1 TO 30
30 X$ = "RECORD NUMBER"+STR$(J)
40 X$ = X$ + LEFT$("**************",20-LEN(X$))
50 RECORD#2,(J),1
60 PRINT#2,X$
70 PRINT X$ DS$ ST
80 NEXT
90 DCLOSE
```

The program is quite similar to those which write sequential files: the file is created, thirty records (in this case) are written to it, and the records, together with the disk status variables DS$ and ST, are printed to the screen.

Line 10 opens a file called 'REL FILE' on drive 1. Its record length is specified as 21. The file is opened for write, but 'W' as a parameter generates ?SYNTAX ERROR because of the implicit confusion between a relative file (signalled by L..) and a sequential file.

Lines 20 - 80 comprise the loop which controls the way records are written to disk:

Lines 30 - 40 generate a string variable X$ of length exactly 20 bytes. (The twenty-first is a carriage return). Leading asterisks pad X$ to the correct length, like this: ****RECORD NUMBER 1 . (It is not necessary to fill the record space in this way; shorter - but not longer - records may be used).

Line 50 uses RECORD# to postion the relative file pointer of file number 2 to the Jth record's first byte. J is bracketed, as required by BASIC 4 syntax. Thus as the loop executes, RECORD#2,1,1 then RECORD#2,2,1 then RECORD#2,3,1 ... set the pointer to records 1,2,3,... and so on.

Line 60: in this way, record number J is printed into the space allocated for it by DOS.

Line 70 prints the record, the disk status variable DS$, and ST to the screen in 30 rows (the first few will be lost as the screen scrolls). This behaves almost identically to the sequential demonstration. However, at intervals, error 50 is signalled in DS$. This is not a serious error, but means only that the relative file is being expanded to incorporate its new data. (See the entry in Chapter 7 under DS$). Since one sector stores 254 bytes, and our records occupy 21 bytes, message #50 is generated about every 254/21 = 12 or so records.

Line 90 closes the file.

DEMONSTRATION OF RELATIVE FILE - READING FROM DISK. (BASIC 4 AND DOS 2+)
```
200 DOPEN#2,"REL FILE",D1
210 FOR J = 30 TO 1 STEP -1
220 RECORD#2,(J),1
230 INPUT#2,X$
240 PRINT X$ DS$ ST
250 NEXT
260 DCLOSE
```
This example should be self-explanatory. However, note the non-sequential order in which records are retrieved. If line 210 is replaced by 210 INPUT "RECORD NUMBER"; J and line 250 by 250 GOTO 210 true relative or 'random' access can be demonstrated.

DEMONSTRATION OF RELATIVE FILE - WRITING TO DISK. (BASIC<4 AND DOS 2+)

The following example duplicates the effects of the BASIC 4 programs which we have just looked at. The only difference lies in the fact that the version of BASIC in use doesn't have the special disk-controlling keywords of BASIC 4. Apart from this, things are much the same: DOS still has to be version 2 (or presumably later versions when these arrive on the scene), not DOS 1.x which hasn't the required relative file capacities.

```
10 OPEN 2,8,2,"1:REL FILE,L" + CHR$(21):    OPEN 15,8,15
20 FOR J = 1 TO 30
30 X$ = "RECORD NUMBER" + STR$(J)
40 X$ = X$ + LEFT$("***************",20-LEN(X$))
50 PRINT#15,"P" + CHR$(2) + CHR$(J) + CHR$(0) + CHR$(1)
60 PRINT#2,X$: S = ST
65 INPUT#15,E1,ER$,E2,E3
70 PRINT X$ E1 "," ER$ "," E2 "," E3 , S
80 NEXT
90 CLOSE 2: CLOSE 15
```

Each line duplicates the corresponding line of BASIC 4's version, with the exception of the additional line, 65. This is interpolated purely to fetch the messages from the command channel which correspond to those of DS$.

Line 10 may need some explanation: OPEN, with the format listed here, opens for relative access with DOS 2+. BASIC 4 sends exactly the same string to the IEEE bus, in spite of the apparent differences in syntax. The same thing is true of line 50, which is equivalent to RECORD. The secondary address of the relative file (i.e. 2, here), the low and high bytes of the record number, and the byte position, are understood by DOS to be the four bytes after P.

DEMONSTRATION OF RELATIVE FILE - READING FROM DISK. (BASIC<4 AND DOS 2+)

```
200 OPEN 2,8,2,"1:REL FILE":    OPEN 15,8,15
210 FOR J = 30 TO 1 STEP -1
220 PRINT#15,"P" + CHR$(2) + CHR$(J) + CHR$(0) + CHR$(1)
230 INPUT#2,X$: S = ST
235 INPUT#15,E1,ER$,E2,E3
240 PRINT X$ E1 "," ER$ "," E2 "," E3 , S
250 NEXT
260 CLOSE 2: CLOSE 15
```

This program reads back the records in the reverse order to that in which they were written, to demonstrate the 'random access' permitted by this type of file structure. The record numbers vary between 1 and 30, so there is no possibility of attempting to read non-existent records. However, as suggested in the last example, modifying line 210 to 210 INPUT "RECORD NUMBER"; J and line 250 by 250 GOTO 210 enables the user to call up any of the records, in the same manner that a record is retrievable by a database system. Because of the structure of line 220, the maximum value of the record number is 255. To increase this to the allowable maximum of 65535, lines like these need to be introduced, so that the RECORD statement in line 200 has both low and high bytes programmable:

```
215 JH = J/256:   JL = J AND 255: REM JH=HIGH, JL=LOW, BYTES OF J. [JH IS ROUNDED
220 PRINT#15,"P" + CHR$(2) + CHR$(JL) + CHR$(JH) + CHR$(1):   REM IN THIS LINE]
```

Demonstration programs: (iii) Program files

Program files are not data storage files in the same way that sequential and relative files are: they do not hold potentially enormous amounts of data in a convenient form for reading, processing, and output. Instead they store consecutive bytes directly from memory, together with the information required to reload the same part of RAM with the identical data. In the CBM system this is accomplished by storing two bytes at the start of the file which hold the load address. Subsequent bytes are read and stored into this address and its following locations, until the file ends. The very last byte is not stored into RAM. (VERIFY uses exactly the same procedure, except that the bytes are compared, not stored, with RAM locations). Both BASIC programs and machine-code programs and routines are stored like this. Consequently,

```
DSAVE "BASIC TEST PROG",D1      :BASIC 4
SAVE "1:BASIC TEST PROG",8      :BASIC<4
```

write a program file to disk called 'BASIC TEST PROG'. The program written is the one which is currently in memory. (If the start-of-BASIC and/or end-of-BASIC pointers are altered, other consecutive RAM will be saved under that name. See SAVE in Chapter 5 and DSAVE in Chapter 7 for more on this subject. As a further example, the SUPERMON and EXTRAMON programs look like BASIC, but include a long chunk of machine-code, which the program causes to relocate into the high end of memory. It is only possible to SAVE or DSAVE such a composite program by altering the end-of-BASIC pointers so that they also include the machine-code).

```
.S "1:M/C PROG",08,033A,037F
```

is a typical command to save a program-file called 'M/C PROG' onto drive 1 of disk 8. All the code between $033A and $037E is saved.

Files of this sort can be read and written almost like sequential files. In fact, some proprietary software (e.g. 'Wordpro')stores its files as program files, and these can be examined and written or rewritten in this way. Similarly BASIC programs and machine-code routines are readable and writeable at will. The OPEN command must be the BASIC<4 type, since BASIC 4's file-handling hasn't concerned itself with these comparatively advanced techniques. All that is required is the use of 'P' as a parameter, when opening the file for reading or writing. Let's look at a few examples of the kind of thing that can be done.

(a) Finding the load address of program files. This is helpful with some types of machine code, and can be useful with unusual BASICs where the normal $0401 start has been overridden. All that is needed is something like this:

```
10 INPUT "FILE NAME, DRIVE NUMBER"; N$,D$ : REM E.G. M/CODE#1  ON   DRIVE 0
20 OPEN 1,8,2,D$ + ":" + N$ + ",P,R"      : REM E.G. "0:M/CODE,P,R"
30 GET#1,X$: IF X$="" THEN X$=CHR$(0)
35 X=ASC(X$)                               : REM X=LOW BYTE OF LOAD ADDRESS ...
40 GET#1,Y$: IF Y$="" THEN Y$=CHR$(0)
45 Y=ASC(Y$)                               : REM ... AND Y=HIGH BYTE
50 PRINT "LOAD ADDRESS IS " X + 256*Y
```

(b) Writing loadable machine-code or other routines directly onto disk. An assembler, for example, might be required to assemble code into an area of RAM already occupied by code. The normal process of putting the code into RAM, then saving the result, is in this case unworkable. However, by writing the load address to disk, followed by bytes of machine-code, any area of RAM can be made the subject of a loadable file, even tricky areas like the zero-page (well, up to a point!) and also into areas like screen RAM.

```
OPEN 1,8,2,"0:CODE,P,W"
PRINT#1,CHR$(1)CHR$(4);   :REM LOAD ADDRESS IS $0401
PRINT#1,CHR$(162)CHR$(0)CHR$(138)CHR$(157)CHR$(0)CHR$(128)CHR$(232)CHR$(208)
        CHR$(249)CHR$(96)
CLOSE 1
```

This prints a simple machine-code string to a file called 'CODE'. If this is LOADed, SYS 1025 will cause the code to execute; it prints 256 different characters on the screen.

(c) Analysing BASIC programs. As we've seen in Chapter 2, BASIC is a complex structure, and a program to deal with it needs to take account of link addresses and linenumbers, and within the program itself, tokens, variables, and special characters (punctuation, quotes, REM, DATA, and meaningless spaces). Examples may be found in Kilobaud-Microcomputing (R W Baker, Sept.'80) and CPUCN (Jim Butterfield, Vol.2 #8). When using a program file, in addition the two leading bytes, which almost always are 1 and 4, need to be read and subsequently ignored. CATALOG in Chapter 7 has a BASIC program, designed to read a disk's directory, which shows the sort of thing.

(d) Processing BASIC (or other) programs. By opening a program, reading it byte by byte, and rewriting the result to another file, a number of editing manoeuvres are possible; for example, merging may be accomplished by writing one program up to a certain linenumber, then writing in turn whichever linenumber is next. The link add- resses have to be preserved. Machine-code may be processed in the same way; for example, one could read it, replacing predefined combinations of characters by others. This may make it possible to painlessly search for identifying messages and so forth. The example shows how two programs (BASIC) may be appended, giving a third. Note the handling of the final zero terminating byte of the first program, which has to be replaced by the first byte of the appending program. I haven't included commands to input file names, format the screen, or check DS$, to save space:

```
100 OPEN 2,8,2,"0:FIRST PROG,P,R"
110 OPEN 3,8,3,"0:BOTH PROGS,P,W" :REM THIS IS THE NEW COMPOSITE PROGRAM
120 GET#2,X$
130 Y$=X$: GET#2,X$: IF ST <> 0 GOTO 200
140 IF Y$="" THEN Y$=CHR$(0)
150 PRINT#3,Y$;                    :REM PRINT SINGLE CHARACTERS TO THE NEW FILE
160 GOTO 130
200 CLOSE 2
210 OPEN 4,8,4,"0:SECOND PROG,P,R"
220 GET#4,Y$: GET#4,Y$
230 GET#4,Y$: IF ST <> 0 GOTO 300
240 PRINT#3,Y$;                    :REM PRINT SECOND PROG TO NEW FILE
250 GOTO 230
300 PRINT#3,CHR$(0);
310 CLOSE 3: CLOSE 4
```

The point of the coding in lines 120-160 is to copy the whole contents of the program file called 'FIRST PROG' into the file 'BOTH PROGS', *except* for the very last byte, which is the zero terminating byte. Hence line 130, which continually reads the next character X$, testing for end-of-file with ST, while writing only the previous charac- ter. By line 200 file number 2 is finished with; file number 4 is opened, and the entire contents of 'SECOND PROG' written to the end of 'BOTH PROGS', including the end zero byte. LOADing 'BOTH PROGS' will reveal a correct append of the programs. Note that line 220 throws away the load address of SECOND PROG; this is now subsumed under the original program's load address, and is not needed. Note also lines 140 and 240. These are needed to cure a small bug caused by the CBM's difficulites with the null character "" and CHR$(0). The same conversion has to be done when construct- ions like GET#1,X$: PRINT ASC(X$) are being used.

Another highly interesting application is in modifying BASIC; the example that follows is based on a routine called 'LOCKSMITH', which adds code to the start of a BASIC program (not BASIC 1, however) so that on LOAD, the stop key is disabled by the usual interrupt address + 3 method (which also turns off the clock, TI), and screen-clear, RUN, and carriage return are forced into the keyboard buffer. The program is thus made to RUN simply on LOAD. (The process is more complex*than I've made it seem here). The program 'protected' in this way is comparatively invulnerable to listing, but of course given the correct approach is rather easy to break. The exam- ple takes a program on drive 0 called 'BASIC PROG' and rewrites it to drive 0 as 'BASIC PROG AUTO'. These names are used for convenience only; obviously proper input and formatting routines enable the program to operate in a user-friendly way. The program itself can be locked. Note that the load address is $0100, the bottom of the stack.

---

*Not only the stack is overwritten, but the IRQ vector is changed (twice), the reset vector called on Stop, NMI is used and a routine updates all the BASIC pointers.

```
100 OPEN 2,8,2,"0:BASIC PROG,P,R"
110 OPEN 3,8,3,"0:BASIC PROG AUTO,P,W"
120 FOR J = 0 TO 1: PRINT#3,CHR$(J);: NEXT
130 FOR J = 0 TO 255: PRINT#3,CHR$(2);: NEXT
140 FOR J = 1 TO 3: PRINT#3,CHR$(0);: NEXT
150 READ J: IF J = 999 GOTO 200
160 N = N + 1: PRINT#3,CHR$(J);: NEXT
200 FOR J = 1 TO 510-N: PRINT#3,CHR$(0);: NEXT
210 GET#2,X$: S = ST: IF X$ = "" THEN X$ = CHR$(0)
220 PRINT#3,X$;: IF S = 0 GOTO 210
300 CLOSE 2: CLOSE 3
500 DATA 165,144,164,145,16,12,24,105,3,144,1,200,141,          130,2,140,131,2
510 DATA 162,18,189,84,2,157,111,2,202,16,247,154,169,1,72,72,72,72,72,169,122
520 DATA 160,2,120,133,144,132,145,88,169,4,133,158,165,40,133,42,165,41,133
530 DATA 43,160,0,162,3,177,42,230,42,208,2,230,43,201,0,208,242,202,208,241
540 DATA 108,148,0,147,82,213,13,0,0,0,0,0,0,0,32,234,255,169,255,133,155,76
550 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,108,252,255,999
```

6.4 CBM diskette formats

Overview of data storage on diskette   CBM disk drives contain their DOS in ROM chips, and so the entire capacity of the diskette is available for files. Except, that is, all the housekeeping details which are necessary to a file-handling system. These are held on a single track (track 18) in 2040/3040/4040 diskettes, and on two tracks in the larger-capacity 8050 (tracks 38 and 39). This is a very standard arrangement. The central position is selected to cut down on track seeking time. Note that Commodore documentation numbers its tracks starting from 1, so the first track is track 1. This is not universal; it seems to be used because 'track 0' is used as a special end-of-file indicator. The sectors are numbered from 0, on the other hand.

There are really only three distinct types of housekeeping information on disk, and programmers who are interested in delving into the niceties of disk programming can write utilities to examine them. The first, on track 18, sector 0 in the -40 range, and track 38 sectors 0 and 3 in the -50 machines, holds mainly the block availability map, or BAM, of the disk, and the 'directory header', which is the title of the disk and its i.d., as written by HEADER or Disk NEW. A BASIC program, 'VIEW BAM'. is a utility program on most demonstration disks which reads this area and translates the bits (which are on/off) into sector (=block) availability (available/used). It reads the disk's name and i.d. too, and calculates the number of free blocks on the disk by a running calculation.

The second region is the remainder of the directory; this is a chain of sectors, which holds a record of each file, its type, its name, and its position on disk. So far as I know, Commodore don't supply a utility to read these sectors. Most of the information in any case is visible on the directory. The exception is the pointers to the first track and sector of the file, which is the first of what may be a very long chain of sectors. With one's own utility program, these details may be displayed on the screen or in hardcopy. 'DISPLAY T&S' prints sectors, without following the chain.

Thirdly, the majority of the disk is occupied by its files. These are generally chained together, with a final terminating block. In principle any file can be traced from its directory entry through all its sectors. This can often be a valuable exercise, and has practical applications in several types of error-correction, error-recovery, and disk revival routines.

Before examining each of these three subdivisions in greater detail, we will look at the arrangement of sectors on CBM disks. The arrangement (more sectors at the outer tracks,   fewer at the inner) is unique to Commodore, to the best of my knowledge. This table summarises the current situation:

TRACKS:	SECTORS:	
	2040 & 3040	4040
1-17	0-20	0-20
18-24*	0-19	0-18
25-30	0-17	0-17
31-35	0-16	0-16

TRACKS:	SECTORS:
	8050
1-39*	0-28
40-53	0-26
54-64	0-24
65-77	0-22

*Includes directory track(s).

The Directory Header and Block Availability Map (BAM).

BYTES:	2040	3040	4040
	Track 18, sector 0:		
0-1	Pointer to directory		
2	Format: 1	1	A
4-143	BAM: 140 bytes in total. Each of 35 tracks has 4 bytes		
144-161	Name of diskette + shift spaces to make length 16 characters		
162-163	Diskette's 2 character i.d.		
165-166	---	---	2A (version)
171-255	Not used		
[180-191	'BLOCKS FREE' may appear here]		

BYTES:	8050
	Track 39, sector 0
0-1	Pointer to BAM1
2	Format: C
6-21	Name of diskette + shift spaces
24-25	Diskette's i.d.
27-28	2C (version)
33-255	Not used

BYTES:	Track 38, sector 0
0-1	Pointer to BAM2
2	Format: C
4-5	Tracks 1 & 51
6-255	BAM1: 250 bytes. Each of 50 tracks has a 5 byte entry.

BYTES:	Track 38, sector 3
0-1	Pointer to directory
2	Format: C
4-5	Tracks 52 & 77
6-140	BAM2: 135 bytes. Each of 27 tracks has 5 byte entry.
141ff	Not used

The left-hand diagram above shows the main features of the BAM and disk identification in the smaller drive units (2040/3040/4040). The diagrams on the right all obtain to the 8050 drives; because of the larger storage capacity, and probably also to allow for future expansion, 3 sectors are used for the data which could be held in 1 sector only on the smaller diskettes. Most of the first sector is unused; BAM is divided into two parts by track, and the first of these sectors holds pointers, a disk format byte, the range of tracks for which it holds the BAM, and 250 bytes devoted to BAM. This formula may be repeated indefinitely; this makes future expansion possible with (some) compatibility. Not all the small detail (e.g. features like shifted spaces, as opposed to null characters) appears in these diagrams. For any disk drive, these less important features can be checked fairly easily with 'DISPLAY T&S' or some other similar utility program.

What is the structure of entries in BAM? As the tables show, the shorter tracks of the smaller disk units have a 4 byte entry in their BAM, while the 8050 uses 5 bytes to map its tracks. The principle in each case is the same. The BAM is split into 4s or 5s, so that the 10th track's map starts at the 37th byte or the 46th byte, and takes up 4 or 5 bytes. The first byte stores the free sectors in the track. This parmeter is used when the directory computes the total number of blocks free. The 3 or 4 remaining bytes, naturally, have 24 or 32 bits in total. Each of these reflects the status of the corresponding sector in that track. Since the largest number of sectors per track in the 4040 type drive is 21, while the corresponding figure for the 8050 is 29, it is clear why the 8050 needs the extra byte in each track's map. Extracting the relevant bit in a program is a bit tricky. The BASIC below shows the method: suppose we are looking at sector number S in a track. We can find the byte which holds the relevant bit, by counting the correct number of 4 or 5 byte units, then picking the 2nd, 3rd, 4th, or, with 8050, perhaps 5th byte. If this byte is B,

$2\uparrow$(S AND 7) AND B finds the bit value; if this is zero, the sector is allocated; if not, it is a free sector.

The Disk Directory

The directory - the list of files stored on a diskette - is contained in a single track, following the header details, which occupy 1 sector. Models 2040/3040/4040 use track 18; the 8050 uses track 39. These tracks respectively contain 20,20,19, and 29 sectors, leaving 19,19,18, and 28 after subtracting the header's sector. Each sector has space for 8 files, so the maximum file storage of these devices is 152,152,144, and 224 files in that order. A diskette can be entirely filled, therefore, if its files average about 1K bytes with the smaller units, or 2K bytes with the 8050. If the average file size is smaller than this, the directory will run out of space before the diskette.

Directory blocks are chained in the usual way, the first two bytes pointing to the track and sector holding the next directory block. The track pointed to is always 18 or 39 of course, except for the final sector, which has a zero terminator. The order of sectors, as with data storage, is not sequential, but, to cut down the time spent waiting for the disk to rotate beneath the head, distributed around the disk at about 180° intervals. Each directory block is divided into eight subdivisions of 32 bytes. The first two bytes are unused, except in the very first such subdivision, where they are used as the linking pointer. This table shows the overall structure:

BYTES:	2040 and 3040 Track 18, sectors 1-20	4040 Track 18, sectors 1-19	8050 Track 39, sectors 1-28
0-31	Linking track and sector pointer + file entry 1 in sector		
32-63	File entry 2 in sector		
64-95	File entry 3 in sector		
96-127	File entry 4 in sector		
128-159	File entry 5 in sector		
160-191	File entry 6 in sector		
192-223	File entry 7 in sector		
224-255	File entry 8 in sector		

Each file entry is formatted as in the following table. Note that the relative file bytes are used only in DOS 2+, and do not appear in earlier DOSes.

BYTES:	CONTENTS OF A DIRECTORY ENTRY:
0-1	Track and sector pointer in first entry. Otherwise unused.
2	FILE TYPE. #0=Scratched/ Not yet used. #80=DELeted #81=SEQuential file #82=PRG, program file #83=USR, user file #84=RELative file #1 - #4 signals an unclosed file. Such files are removed by COLLECT. #80 is a scratched unclosed file, a type to be avoided.
3-4	Track and sector pointer to first block of file.
5-20	File name + shifted spaces (#A0 characters).
21-22	Track and sector pointer to relative file's first side sector.
23	Record size of relative file (i.e. parameter following L on opening file).
24-27	Unused
28-29	Replacement track and sector pointer for OPEN@
30-31	Low and high byte of no. of blocks in file, as shown on the directory.

On the next page we have some actual examples, produced with the utility called 'DISPLAY T&S', but output to a CBM printer instead of the screen. The features in these tables are marked.

Sector 0 of track 18 on a DOS 1+ diskette is displayed below. The diskette is nearly full; the BAM shows that the outer tracks have 21 + 21 + 5 free sectors, and the inner 14 + 17 + 17 + 17 sectors. Note that most of the BAM shows as zero bytes. This is because bit 0 is used to indicate that a sector is allocated; and a further byte gives the number of free sectors. All four bytes are therefore zero. The directory track's BAM entry is visible in the middle of BAM; 11 sectors are free, so the diskette could hold another 88 files, although they would have to be rather short. Note also the title of the disk and its two-character identifier.

```
TYPICAL TRACK/SECTOR CONTENTS OF THE DIRECTORY

00 : (12 01)(01) 00 (15 FF FF 1F :      ππ  ←Next directory sector.
08 : 15 FF FF 1F 05 80 82 0A :    ππ  ■■ ←Version number.
10 : 00 00 00 00 00 00 00 00 :
18 : 00 00 00 00 00 00 00 00 :
20 : 00 00 00 00 00 00 00 00 : Block Availability Map.
28 : 00 00 00 00 00 00 00 00 :
30 : 00 00 00 00 00 00 00 00 :
38 : 00 00 00 00 00 00 00 00 :
40 : 00 00 00 00 00 00 00 00 :
48 : (0B 68 DB 06) 00 00 00 00 :   <+ ←— Directory BAM entry.
50 : 00 00 00 00 00 00 00 00 :
58 : 00 00 00 00 00 00 00 00 :
60 : 00 00 00 00 00 00 00 00 :
68 : 00 00 00 00 00 00 00 00 :
70 : 00 00 00 00 00 00 00 00 :
78 : 00 00 00 00 00 00 00 00 :
80 : 0E F5 F7 01 11 FF FF 01 :  I ‾ ππ
88 : 11 FF FF 01 11 FF FF 01 :  ππ  ππ
90 : 55 4E 49 56 45 52 53 41 : UNIVERSA  ←Disk Name
98 : 4C 20 44 45 4D 4F 20 20 : L DEMO      + i.d.
A0 : A0 A0 56 31 20 20 A0 A0 :    V1
A8 : A0 A0 A0 00 00 00 00 00 :
B0 : 00 00 00 00 12 4C 4F 43 :        LOC
B8 : 4B 53 20 46 52 45 45 2E : KS FREE.
C0 : 20 20 20 20 20 20 20 20 :
C8 : 20 20 20 20 20 20 20 00 :
D0 : 00 00 00 00 00 00 00 00 :
D8 : 00 00 00 00 00 00 00 00 :    ←Unused.
E0 : 00 00 00 00 00 00 00 00 :
E8 : 00 00 00 00 00 00 00 00 :
F0 : 00 00 00 00 00 00 00 00 :
F8 : 00 00 00 00 00 00 00 00 :
```

A typical directory sector is listed on the next page. Note that there are exactly eight file entries. The first two bytes point to another sector on the same track. When a directory is read and printed to the screen, these sectors are read in order, and the type of file, file name, and number of sectors occupied are all read from each of these eight entries. and converted into readable form. The file-type entry, for example, is converted from #81 into SEQ, which is more meaningful to the user. For this reason, unless a special array-sorting process is used, the sequence of items as displayed by a directory tends to be immutable, so that sometimes it is worthwhile to plan the order in which files are recorded.

Because of CBM graphics conventions, the file-type is displayed as a reverse character on the sector listing - the reverse heart is a sequential file, the remaining files are all #82 = program files.

The first entry in this sector has been marked with 5 boxes; these mean:

(i) The next directory sector is track 18, sector 10. (18 = hexadecimal 12).
(ii) The file-type, #82, shows that this is a program file.
(iii) The program begins at track 22, sector 9.
(iv) 'DISK COMM2 ' is the program's name.
(v) The program occupies 2 sectors only (i.e. is less than 509 bytes in length).

```
         (i).    (ii).   (iii).        (iv).
00 : (12  0A)(82)(16  09)/44  49  53 ):  ▌▌ DIS
08 :/4B  20  43  4F  4D  4D  32  A0/:  K COMM2
10 :(A0  A0  A0  A0  A0/00  00  00 :
18 : 00  00  00  00  00  00 (02  00):  (v).
20 : 00  00  82  16  10  44  49  53 :  ▌▌ DIS
28 : 4B  20  43  4F  4D  4D  33  A0 :  K COMM3
30 : A0  A0  A0  A0  A0  00  00  00 :
38 : 00  00  00  00  00  00  03  00 :
40 : 00  00  82  0D  00  44  49  53 :  ▌▌ DIS
48 : 4B  20  57  52  49  54  45  A0 :  K WRITE
50 : A0  A0  A0  A0  A0  00  00  00 :
58 : 00  00  00  00  00  00  04  00 :
60 : 00  00  82  0D  01  44  49  53 :  ▌▌ DIS
68 : 4B  20  52  45  41  44  A0  A0 :  K READ
70 : A0  A0  A0  A0  A0  00  00  00 :
78 : 00  00  00  00  00  00  04  00 :
80 : 00  00  82  0D  03  44  49  53 :  ▌▌ DIS
88 : 4B  20  4F  56  45  52  4C  41 :  K OVERLA
90 : 59  53  A0  A0  A0  00  00  00 :  YS
98 : 00  00  00  00  00  00  02  00 :
A0 : 00  00  82  0D  04  44  49  53 :  ▌▌ DIS
A8 : 4B  20  44  49  52  A0  A0  A0 :  K DIR
B0 : A0  A0  A0  A0  A0  00  00  00 :
B8 : 00  00  00  00  00  00  05  00 :
C0 : 00  00  81  0D  09  20  20  50 :  ▌ P
C8 : 45  54  20  44  41  54  41  20 :  ET DATA
D0 : 20  A0  A0  A0  A0  00  00  00 :
D8 : 00  00  00  00  00  00  38  00 :       8
E0 : 00  00  82  17  00  52  41  4E :  ▌▌ RAN
E8 : 44  4F  4D  20  31  2E  30  30 :  DOM 1.00
F0 : A0  A0  A0  A0  A0  00  00  00 :
F8 : 00  00  00  00  00  00  22  00 :       "
```

A relative file entry is slightly more complex than the other types of file. The single specimen below shows the extra features of

(vi) Pointer to side-sector chain, which here starts in track 15, sector 2.
(vii) Length-of-record parameter. The records in the example are 21 bytes long.

```
C0 : 00  00 (84)(10  0E)/52  45  4C ): ▌REL
C8 :/20  46  49  4C  45  A0  A0  A0/:  FILE
D0 :\A0  A0  A0  A0  A0/(0F  02)(15): ←—(vi),(vii)
D8 : 00  00  00  00  00  00 (04  00):
```

Finally, we leave our tour of the CBM disk system with a few examples of data storage on disk. The next page has examples of sequential file storage, BASIC storage, and machine-code. From the diskette's point of view, these are all stored in a similar way, in chains of sectors in which the first two bytes either point to the next track and sector, or contain track number zero, to indicate end-of-file, with the second byte holding the number of valid bytes in this final sector.

Program files are stored with an introductory load address pointer, so BASIC begins 01 04 followed by the RAM dump of the program. This consists of lines linked by pointers, each line containing a link address, a linenumber, and BASIC, terminated by a zero byte. The exception is the very last line, which has a link address of zero to show that the program is finished. This pattern can be traced in the BASIC dump on the next page. Note that much of BASIC is readable, although the tokens are in an unfamiliar form. The program includes machine-code (it is DOS Support, which is a BASIC loader for machine-code). Typically, this is rather amorphous. On the other hand, files, which use ASCII storage, are usually entirely readable. Note the carriage

return characters, which are the record separators in the CBM system.

```
00 : (11 0A)(01 04)(17 04)(05 00) :
08 : 41 B2 31 32 AC 31 36 AE : A┬12 ∎16┐
10 : 33 3A 8F 20 24 43 30 30 : 3:∎ $C00
18 : 30 00 34 04 0A 00 8B C2 : 0 4    ∎│
20 : 28 41 29 B3 B1 37 36 A7 : (A)┤┴76 │
28 : 9E 31 36 33 39 3A 8F 20 : ∎1639:∎
30 : 42 41 53 49 43 32 00 50 : BASIC2 P
38 : 04 0F 00 8B C2 28 41 29 : ∎│(A)
40 : B2 37 36 A7 9E 32 31 35 : ┬76 ∎215
48 : 31 3A 8F 20 42 41 53 49 : 1:∎ BASI
50 : 43 34 00 8F 04 14 00 99 : C4 ∎    ∎
58 : 22 93 11 11 11 11 11 11 : "�‖
60 : 11 11 11 11 11 11 20 20 :
68 : 20 20 20 20 55 4E 49 56 :     UNIV
70 : 45 52 53 41 4C 20 44 4F : ERSAL DO
78 : 53 20 53 55 50 50 4F 52 : S SUPPOR
80 : 54 20 4C 4F 41 44 45 44 : T LOADED
88 : 11 11 11 11 11 11 11 11 :
90 : 22 00 95 04 1E 00 A2 00 : "  ▮   ▬
98 : (00 00)(AA AA AA AA AA AA : ‖‖‖‖‖‖
A0 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
A8 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
B0 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
B8 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
C0 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
C8 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
D0 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
D8 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
E0 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
E8 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
F0 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖
F8 : AA AA AA AA AA AA AA AA : ‖‖‖‖‖‖‖‖

00 : (11 14) AA AA AA EA E6 77 : ‖‖‖ ▮7
08 : D0 02 E6 78 86 B3 BA BD : ┐ ▩8▩┘┘
10 : 01 01 C9 9B D0 3A BD 02 : ↘∎┌:┘
18 : 01 C9 C3 D0 33 A5 77 D0 : ↘─┐│ 7┐
20 : 2C A5 78 C9 02 D0 26 A0 : ,∣ 8↘ ┐&
28 : 00 84 B3 B1 77 C9 3E F0 : ▮┤┴7↘> ┌
30 : 11 C9 40 F0 0D C8 85 B3 : ↘@┌ ‖▦┤
```

←**BASIC program.**

←End of BASIC program.

←Garbage before machine-code section.

←machine-code.

```
00 : (0D 13) 30 30 30 31 2C 50 :  0001,P
08 : (0D) 30 30 30 32 2C 50 45 : 0002,PE
10 : (0D) 30 30 30 33 2C 50 45 : 0003,PE
18 : 54 (0D) 30 30 30 34 2C 50 : T 0004,P
20 : 45 54 20 (0D) 30 30 30 35 : ET  0005
28 : 2C 50 45 54 20 44 (0D) 30 : ,PET D 0
30 : 30 30 36 2C 50 45 54 20 : 006,PET
38 : 44 49 (0D) 30 30 30 37 2C : DI 0007,
40 : 50 45 54 20 44 49 53 (0D) : PET DIS
48 : 30 30 30 38 2C 50 45 54 : 0008,PET
50 : 20 44 49 53 4B (0D) 30 30 :  DISK 00
58 : 30 39 2C 50 45 54 20 44 : 09,PET D
60 : 49 53 4B 20 (0D) 30 30 31 : ISK  001
68 : 30 2C 50 45 54 20 44 49 : 0,PET DI
70 : 53 4B 20 44 (0D) 30 30 31 : SK D 001
78 : 31 2C 50 45 54 20 44 49 : 1,PET DI
80 : 53 4B 20 44 41 (0D) 30 30 : SK DA 00
88 : 31 32 2C 50 45 54 20 44 : 12,PET D
90 : 49 53 4B 20 44 41 54 (0D) : ISK DAT
98 : 30 30 31 33 2C 50 45 54 : 0013,PET
A0 : 20 44 49 53 4B 20 44 41 :  DISK DA
A8 : 54 41 (0D) 30 30 31 34 2C : TA 0014,
```

Relative files are more complex than the other types, and are stored in a more elaborate manner. Each file is held in two parts: the first is the main data storage, which resembles a sequential file. The other part is the chain of side sectors. The starting-point of each chain is recorded in the directory entry. Let's first look at the data storage. The important point to note is that the file is divided, conceptually at least, into equal chunks of data. The size of each chunk is determined by the 'L' parameter when the file is opened for the first time. It is convenient to refer to these subdivisions as 'records', although by using the byte pointer of the RECORD statement, it's possible to write several 'records' into the allocated space. A record may be divided, by commas or colons, into 'fields': typically, a name and several lines of address make up the fields in a record. In a sense, within each record, the data is read and written sequentially, in a statement like PRINT#8,A",,"B$",,"C$. Good file-design takes account of the speed of access possible with relative files, and also of the different ways in which data may be arranged on file. As we have seen, a sequential file is straightforward to write to: data is written, and finished off with a Return; and subsequent data is added to the end of this, so the result is a long file punctuated by Return characters.

Writing to a relative file isn't quite as simple, because of the fact that at any time a record can be accessed and written to. Suppose record #250 holds a name and address at present; then the program decides that record #250 should be some other name and address, so it calls up this record, and writes the new data into the record. If the new address is shorter than the older one, there is a risk that garbage may be left in the record. Consequently, any PRINT# statement to a relative file not only writes its data, but also fills the remainder of that record, from its position at the end of its data to the record end, with null characters (zero bytes). This must be borne in mind if several PRINT# statements are made to the same record, using constructions like RECORD#1,250,10 to write from byte position 10 in the record. Apart from this subtlety, PRINT# is usable exactly like a sequential record's PRINT#, and a Return character is written in the same way. For these reasons, the easiest way to use relative files is to follow these two rules:

(i) Test the length of your data before printing it to disk, to check that it'll fit the record size; remember the carriage return character.

(ii) Use a single PRINT# statement for each record, for example: PRINT#8,N;",,";N$;",,";M;M$ which writes 3 fields to a record, consisting of a numeral N, a string N$, and a composite field holding numeral M then string M$.

DATA SECTOR CHAIN:

SIDE SECTOR CHAIN:

← Pointers linking data sectors.

← Pointers to sectors of data.

← Pointers between side-sectors.
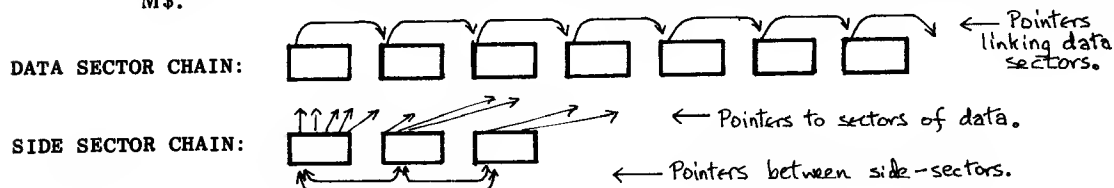
[NOTE: The chained sectors are not arranged linearly, but scattered about the diskette]

As the diagram shows, relative files have a chain of 'side sectors', which point to the sectors holding the actual data. For example, sector 250 has its own pointer, which is in the third side sector, and consists of its track and sector number. When a record is accessed, the record length is used to calculate which sector(s) hold the record; the maximum length of a relative record is 254, so two sectors at most hold it. The appropriate side-sector is loaded into its buffer, and the pointers for two sectors read, so the actual data can be loaded next. This is quite an efficient process. To take an actual example, suppose we wish to read record #100, and the record length is stored as 100. Also suppose that the file is open, and one of its buffers holds a side-sector, and the other holds a data sector. (The third holds data for input or output). The record starts at the 99*84 = 8316th byte in the relative file of data. In a sequential system, we'd have to read consecutive sectors to find this. In our relative file system, though, DOS calculates that the 8316th byte is to be found in sector 32 (i.e. 8316 divided by 254). So sector 32 (and perhaps 33) must be read. Where is sector 32 of the data file? It's pointers are held in side sector number 32/240, i.e. 0. So the disk is searched for this sector, which takes a single disk read unless this side sector is in a buffer already, in which case it can perform the next step immediately, which is

to calculate the position in the side sector which stores sector 32's pointers, read the track and sector from this position, and finally read the disk again. Where necessary, the next data sector can be read either from the side sector, or from the prior data sector. Three disk reads are therefore the maximum required by this method.

Each side sector is formatted like this:

BYTES:	CONTENTS:
0-1	Track and sector pointer to next side sector.
2	Side sector number, 0-5.
3	Record length of relative file.
4-15	6 pairs of pointers to *every* side sector.
16-255	120 pairs of pointers to consecutive sectors of data.

so that a new side sector can be found with a single disk-read only. The first pair of pointers is the usual DOS maintenance link pointer set, so that the file can be COPYed (in principle) and not COLLECTed. An extra channel needs to be kept open by DOS for this type of file; one for the side sector, one for a data sector, and the third for the data itself. (A sequential file needs only two, holding a sector and data respectively). Since ten channels is the maximum allowed by the system, apart from channels 0 and 1, the following combinations of *open* files are the most obtainable (more files may be used in a program by closing some while others are open):

	3 Relative files + 0 Sequential files
or	2 Relative files + 2 Sequential files
or	1 Relative file + 3 Sequential files
or	0 Relative files + 5 Sequential files.

The first issues of DOS 2.5 (for 8050 drives) permit only a maximum of 6 side sectors to exists. This is the same number as is available with the 4040 drives, and is something of a restriction; 'The 8050 thinks it is three 4040s', as I've heard it put. A single relative file can't fill the whole of an 8050 disk; three can. This restriction will be removed with the third set of ROMs for the 8050.*

Let's look at the restrictions implicit in the relative files' handling. First of all, the number of records is held as two bytes, and can't exceed 65535. Secondly, the length of a record can't exceed 254. Thirdly, the maximum number of sectors which the file can occupy is limited, by the side sector restriction, to 120*6=720 sectors. So the maximum data storage capacity of one of these files is 720*254 = 182 880 bytes. This is not a great deal, so users of the 8050 may need to separate what could have been a single file into several of shorter record-length. Data compression techniques may be used, particularly with numerals, and repetitive information should be left out; for example, the demonstration programs in section 6.3 of this chapter all write records of the form 'RECORD NUMBER x'; in practice, only the x need be stored, as is perhaps obvious. To calculate the longest available record-length, when the number of records is known, divide 182 880 by that number and subtract 1 (for a carriage return); this gives the 8050 figure. Commodore documentation puts the length of a 4040 relative file as 167 132 bytes maximum (and implies that this is a diskette's maximum, not a file's maximum, which is confusing). Thus, 1827 records of length 100 can be fitted into an 8050 relative file.

Another approach is the use of a large number of small files. Section 6.2 explained how 'inverted files' make a suitable structure for a database; a practical illustration is the OZZ and the later Silicon Office retrieval system, in which relative records are recoverable by a key such as 'SMITH218', which includes both a relative record number (218) and its own internal system, in which a series of short files hold pointers based on the initial of the field. Thus, a relative file holds the Ss in sorted order, from which the corresponding data can be recovered. (Each new entry is added to the relative file, and also merged into its index file).

---

*As it may be important to have the facility to store large relative files, a program like this one may be useful. It is one way (of many) to test the upper limit of DOS with relative files. Put a formatted empty disk in drive 0:

```
10 DOPEN#1,"TEST",L100,D0
20 FOR J = 1 TO 1E9 : RECORD#1,(J*100): IF DS<>52 THEN NEXT
30 PRINT "MAX.REL.FILE APPROX.=" 100*J "BYTES": DCLOSE
```

6.5 Direct access programming to disk

CBM direct access commands   Commodore's motives for introducing these commands, whilst supplying very little documentation, seem to have been mixed. Possibly they were a temporary measure, designed to suggest that relative files and other convenient file-handling techniques were available with DOS 1+, when in fact implementation was rather difficult. DOS 2+ does not appear to have been correctly updated, so that the 'Block-Allocate' function may not work correctly. There were at the time of writing persistent rumours (or to be precise, rumors) that new CBM disks will drop these commands, switching to others and also being 'more supportive'. This remains to be seen. The following summary applies to DOS 1+ and DOS 2+, and consequently to the range of 2040,3040,4040, and 8050 drives; it may not apply to later versions of DOS.

These drives contain two microprocessors; one of these processes the incoming data on the IEEE bus, including the command channel strings and the input and output of bytes of data. This shares RAM with the other processor, which in effect is a disk controller chip, operating the read/write head, the motors, the encoding and decoding of bytes, the error handling, and the housekeeping, including such matters as checking clock pulses, and testing cyclic redundancy checks. This processor is less accessible than the other; there's a well-known Butterfield program (see e.g. IPUG,Jan.'80) which in effect enables either chip's ROM to be disassembled. In its original form it is written for the 2040 drive. Some programs make use of these facilities to provide a high degree of copy protection. For example, OZZ and 'Silicon Office' have their own 'U' routines to read and write sectors, which are reputed to be different from the normal ones, so that the resulting disks are truly uncopyable with the normal CBM instructions. This sort of thing is rather unusual, and tends to require co-operation from Commodore to be workable.

How are direct access commands sent? A special character in the 'open' statement signals that this type of processing is to be used, and DOS allocates a buffer. This is numbered with a 'channel number' which is identical to the secondary address used in the 'open' statement. The syntax is:

```
OPEN 1,8,2,"#"   :REM ASSIGNS A DISK BUFFER TO CHANNEL 2 AND LOGICAL FILE NO. 1
OPEN 7,8,5,"#6"  :REM ASSIGNS BUFFER 6 TO CHANNEL 5 (OR ERROR 70,NO CHANNEL)
```

DOPEN includes an irrelevant drive number. GET#1,X$ or GET#7,X$ returns the buffer number (3-12), in our examples. The first format is less trouble, since it searches for a free buffer itself. The channel number occurs in all the commands in which sectors are read or written; the BAM instructions (Block-Allocate and Block-Free) don't use it, neither do the DOS memory commands. However, the 'U' commands, which jump to DOS RAM (and, in the 2040 only, to an expansion ROM socket), also require the channel number. Note that secondary addresses 0 and 1 are reserved by DOS for reading and writing.

The examples that follow assume, for consistency in exposition, that

```
OPEN 15,8,15
OPEN 1,8,2,"#"
```

have been performed, opening the command channel as file #15 and a direct-access channel as file #1, channel number 2 – the different numbers to emphasise which parameter is in use. Some of these commands, those beginning 'B', resemble disk commands as sent by BASIC<4, in that alternative spellings are accepted; the hyphen is the separator between the two parts of the command name. Mnemonically, this can be a help. A colon marks the command name's end. The 'M', or 'memory', commands are less forgiving in this respect. They are also intolerant of the use of numerals or variables in place of strings, while 'B' and 'U' commands are able to deal with numbers as they are received. After the alphabetic list, with its short illustrations, I've included fuller examples of the use of these direct access commands; looking at examples is probably the easiest way to get the feel of them.

I should perhaps add that these files, like all others, can be closed when they are finished with. This may cause the block availability map to be written to the disk, to reflect its updated status, and thus make permanent the changes in sectors which these routines may have carried out. Conversely, if COLLECT (Disk VALIDATE) is carried out at a future time, user-written sectors, even though they've been allocated in BAM, will be de-allocated, unless an elaborate system of pointers has been included to mimic one of the acceptable file types. Such sectors will then be liable to be over-written if files are subsequently stored on the disk by DOS. For this reason, direct

access files must normally be segregated from files which are maintained by DOS, so that an entire disk (or side of a disk, with double-sided drives) may be reserved for this form of data storage.

B-R

BLOCK-READ command. Replaced by U1.

This command enables any block on a standard-format disk to be read. The disk must be formatted in the same way as the drive which is reading; in particular, the number of tracks must match, so that the stepper-motor intervals are equal. 8050 disks cannot be read by -40 drives, and -40 disks cannot be read by 8050 drives. U1 is the more reliable instruction, and is recommended in place of B-R, which it closely resembles. If a sector cannot be read, because of some flaw in the recording process, Block-Write can sometimes reconstruct the sector, so (for example) disks which will no longer initialise may be recoverable.

Syntax: PRINT#15,"U1"; channel number; drive; track; sector   or:
         PRINT#15,"U1: string of four characters" where the four bytes are taken to be channel, drive, track, and sector.
         Note that commas, instead of semi-colons, may be usable as separators in the first form, though semi-colons are the recommended character.
         On executing this command, the sector is read into the channel's buffer, and the buffer-pointer set to the start, so that characters input from the buffer will read from the beginning. If bytes 144-145 only, say, are wanted, Buffer-Pointer can be used to shift this pointer.

Examples: The first example is rather rudimentary; it prints 256 characters from any sector directly to the screen. If these include screen editing characters, the screen will clear, shift, etc. 'DISPLAY T&S' works like this, using a decimal to hexadecimal conversion routine on ASC(X$). Remember that X$ must be tested for equality with the null string "", and converted to CHR$(0) when it is null.

```
10 OPEN 1,8,2,"#": OPEN 15,8,15
20 INPUT "DRIVE, TRACK, SECTOR"; D,T,S
30 PRINT#15,"U1";2;D;T;S          :REM 2 IS SECONDARY ADDRESS OF FILE NO.1
40 GET#1,X$: IF ST=64 GOTO 20     :REM END OF BUFFER
50 PRINT X$;: GOTO 40             :REM PRINT CHARACTER FROM SECTOR
```

The next example shows how a chain of sectors can be followed; all that's needed is to read the first two bytes of a sector, and read the corresponding track and sector, until eventually the track is recorded as zero, showing that the file has come to an end. The routine prints each track-sector pair:

```
10 OPEN 1,8,2,"#": OPEN 15,8,15
20 INPUT "FIRST SECTOR'S DRIVE, TRACK, AND SECTOR"; D,T,S
30 PRINT "TRACK =" T " SECTOR = "S
40 PRINT#15,"U1";2;D;T;S
50 GET#1,T$: GET#1,S$
60 IF T$="" THEN PRINT "TRACK = ZERO": CLOSE 1: CLOSE 15: END
70 IF S$="" THEN S$=CHR$(0)
80 T=ASC(T$): S=ASC(S$): GOTO 30
```

Further examples occur in RANDOM 1.00, the 8K program to process relative files with DOS 1. See linenumbers 410ff. for typical read, update and write disk processing.
         The alternative form of syntax of this command, and of the remaining Block commands, is less convenient to use, so I shall only briefly mention it here. (I am talking about BASIC programming; in machine-code it is easier to send a command string on the IEEE bus in the alternative form). It consists of a string of 7 characters:

```
1000 C$ = "U1:" + CHR$(CH) + CHR$(DR) + CHR$(TR) + CHR$(SE)
1010 PRINT#15,C$

3000 PRINT#15,"U1:" CHR$(CH) CHR$(DR) CHR$(TR) CHR$(SE)
```

B-W

BLOCK-WRITE command. Replaced by U2.

This command writes the current contents of a specified buffer to any track and sector of a disk. Data may be put in the buffer by PRINT#1, in combination with Buffer-Pointer if the data is wanted in mid-buffer. It may be loaded from another sector, or a combination of the two methods may be used to load/ update/ rewrite any sector. U2 is recommended in place of B-W.

Syntax: PRINT#15,"U2"; channel number; drive; track; sector   or:
      PRINT#15,"U2: string of four characters" where the four bytes are taken to be channel, drive, track, and sector numbers.
      The position of the Buffer-Pointer is not relevant to this commmand; it is used by INPUT#1, GET#1, and PRINT#1, when reading from or writing to the buffer, but the entire sector is written irrespective of the pointer's position. After this command, the pointer is left at the start of the buffer (byte 1).

Examples: The first example writes to every sector in track 1. This may be used as the basis of a test timing program; varying the order in which they are written will cause variations in time taken. CBM disks use an algorithm to calculate the position of the 'next' sector in a track; the -40 series for example adds about 10 to the previous sector.

```
10 OPEN 1,8,2,"#":  OPEN 15,8,15
20 DATA 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
21 REM ORDER IS VARIABLE. NOTE THAT 8050 UNITS HAVE SECTORS 0-28 IN THIS TRACK
30 FOR S = 0 TO 20
40 READ SE
50 PRINT#15,"B-P";2,1    :REM CHANNEL 2 IS SECONDARY ADDRESS. BYTE POSITION=1
60 PRINT#1,"MESSAGE" + STR$(S) :REM THIS WILL BE WRITTEN TO THE SECTOR
70 PRINT#15,"U2";2,D,1,SE       :REM D = DRIVE NUMBER, 1=TRACK, SE=SECTOR
80 NEXT: CLOSE 1: CLOSE 15
```

Note that BASIC<4 will add an extra line feed, CHR$(10), on the end of the message in line 60. The next example shows a sector being partly rewritten with PRINT#. Note that in BASIC this is accompanied by a carriage return character; this is unlikely to be wanted if the sector is part of BASIC or machine-code, but is acceptable with files. To avoid the character being sent, simply finish the PRINT# statement with a semi-colon.

```
240 INPUT "INSERT MESSAGE FROM WHICH STARTING-POINT (1=SECTOR START)";P
250 INPUT "MESSAGE";M$
260 PRINT#15,"B-P";2;P          :REM 2 IS SECONDARY ADDRESS = CHANNEL
270 PRINT#1,M$;                  :REM WRITE MESSAGE (COULD ALSO CHECK THAT
                                     P + LEN(M$) < 257).
```

B-E

BLOCK-EXECUTE command.

Block-Execute is analogous to a program LOAD then RUN. It loads the specified sector into its buffer, then jumps, in machine-language, to the start of the buffer. On encountering RTS it returns to the BASIC program using it. This command is not often used, since there is little advantage in its use unless the programmer has detailed knowledge of the DOS ROM in the disk unit. The Memory-Execute command is similar, but its machine-code is not kept on a disk sector, but directly PRINTed to memory, so it is more widely used in BASIC utilities.

Syntax: PRINT#15,"B-E"; channel number; drive; track; sector   or:
      PRINT#15,"B-E: string of four characters" where the four bytes are taken to be channel, drive, track, and sector numbers.

Examples: Any of the Memory-Execute type of command met with in utilities can be written to a sector on disk, but will be vulnerable to overwriting if COLLECTed, so that the disk cannot be an ordinary DOS disk. For example, a pair of routines in Transactor (reprinted in CCN, July '81) give the whereabouts of DOS' subroutines to

compute the number of free sectors in a diskette (the routine is part of the directory, and adds the free sectors as they appear in BAM). The information for DOS 2 and DOS 2.5 is:

DOS 2: Put disk drive number (0 or 1) in location $12,
       execute routine $DB34,
       then blocks free are contained in $4377 (low byte) and $4378 (high byte).

DOS 2.5: Put drive number (0 or 1) into location $12,
       execute routine $D3E7,
       then blocks free are contained in $43AF (low byte) and $43B0 (high byte).

Since the B-E buffer is not fixed, unused RAM locations of DOS are needed; one of these is Memory-Written with the drive number, and B-E called, holding this:

LDA DRIVE/ JSR DB34/ LDA 4377/ STA RAMLO/ LDA 4378/ STA RAMHI/ RTS

B-A

BLOCK-ALLOCATE command.

This is a BAM command which sets a bit in the BAM low, corresponding to a track and sector which the parameters specify. This prevents the sector from being overwritten, unless COLLECTed, when the BAM bit will be set high again. If the requested block is already in use, error 65,NO BLOCK is signalled in DS$ or by GET#15,E. Some DOS versions return the next available track and sector with the error string. This makes the allocation of new sectors very easy, since the same sector can always be requested (e.g. the directory header) which is known to be used, and the next parameters can simply be read out. This works with DOS 1, but is reported to be unreliable with DOS 2; this of course makes DOS 2 more unworkable than DOS 1 in this respect. The way to get round this bug is to try tracks and sectors in some increasing pattern, until error 65 no longer appears. Ideally the pattern should be that of the order of sectors used by DOS. The 'next' track and sector is evaluated by DOS according to its own algorithms.* The syntax is identical to that of Block-Free.

Examples: With DOS 1, this type of subroutine will allocate a new T and S:
```
1000 PRINT#15,"B-A";D;T;S
1010 INPUT#15,E,E$,ET,ES
1020 IF E=0 THEN RETURN      :REM T AND S ALLOCATED SATISFACTORILY
1030 IF E<>65 THEN GOTO ...:REM ERROR-HANDLING ROUTINE
1040 T=ET:  S=ES:  IF T=18 THEN T=19: REM AVOID DIRECTORY TRACK
1050 GOTO 1000              :REM ALLOCATE THE NEXT TRACK & SECTOR AS RETURNED
```

DOS 2 requires a more tedious routine; the following is an outline, omitting the detail required to test sectors for validity. See linenumbers 800ff. in 'RANDOM 1' for an example of the type of test that's needed. Since this routine relies on incrementation of the track number, a full diskette can only be ensured by writing from track 0, sector 0, rather than starting at the directory track's neighbourhood.
```
1000 GOSUB ...              :REM INCREMENT SECTOR/TRACK; AVOID DIRECTORY
1010 PRINT#15,"B-A";D,T,S   :REM TEST THE BAM
1020 IF DS=0 THEN RETURN    :REM T AND S ALLOCATED SATISFACTORILY
1030 IF DS<>65 THEN GOTO ... :REM ERROR-HANDLING ROUTINE
1040 GOTO 1000              :REM CONTINUE LOOP
```

---

*The algorithm is required to generate a series of sectors, covering the entire range without repetition, with about half a disk's separation between consecutive sectors. Typically, it generates even sectors in ascending pairs followed by odd sectors in descending pairs. Example: (i) Set a constant = ½ the number of sectors, to the nearest integer. (ii) Select two even constants, one greater and one less than the number of sectors. (iii) Start with sector 0, adding the constant, subtracting the larger value when the sector is impossibly high: when zero, set the sector to 1 and repeat, but subtracting the smaller constant on overflow.
    Consider track 1 of the -40 range, with sectors 0-20. Setting our increment to 10, and our high and low constants to 22 and 18, the following sequence is generated:
    0,10,20,8, (i.e. 30-22),16,4,14,2,12,1, (i.e. in place of 0),11,3,(i.e. 21-18), 13,5,15,7,17,9,19, exit (as sequence exhausted).

B-F

BLOCK-FREE command.

This command is the converse of Block-Allocate; it is a BAM command which sets the specified track and sector bit in BAM high, so the sector is de-allocated. The sector still exists, but is liable to be overwritten.

Syntax: PRINT#15,"B-F" drive; track; sector
The syntax is identical to that of 'Block-Allocate'. Note that a channel number is not required.

Example: The routine de-allocates all the sectors of a diskette. The BAM at the end of this process consists of 1s only. Routines of this sort may be useful in experiments with free-format disks without a directory. Note however that de-allocating sectors with this command is not *necessary;* allocated sectors can be written over, ignoring the warning of error 65 that the sector has previously been allocated.

```
1000 FOR T = 1 TO 35     :REM 2040,3040 OR 4040
1010 DATA 20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,19,19,19,19
1012 DATA 19,19,19,17,17,17,17,17,17,16,16,16,16: REM 4040 HAS 18, NOT 19
1020 READ X: FOR S = 0 TO X
1030 PRINT#15,"B-F";D;T;S
1040 NEXT S,T
```

B-P

BUFFER-POINTER command

Buffer-Pointer controls the pointer to a buffer used by the 'Block' commands. It is used both when PRINTing data into a buffer and when reading it out. The rationale is similar to RECORD when relative files are used from BASIC.

Syntax: B-P has only two parameters, the channel number and the byte position. The latter normally takes values in the range 1- 255.
        PRINT#15,"B-P"; channel number; byte position

Examples: Block-Write includes several examples of this command. To illustrate how Buffer-Pointer can be used with read, I've written this short example program; it reads a sector from the directory, then reads the requested file details from the buffer, using the fact that each file's data occupies 32 bytes.

```
10 REM DIRECTORY TRACK = 18 OR 39; SECTORS = 1-20, 1-19, 1-28; DEPENDING ON MODEL
20 OPEN 1,8,2,"#": OPEN 15,8,15
30 INPUT "DRIVE, SECTOR"; D,S
40 PRINT#15,"U1",2,D,39,S         :REM OR TRACK = 18 FOR 2040, 3040, 4040
50 INPUT "WHICH FILE"; F          :REM F FROM 1-8
60 PRINT#15,"B-P";2;32*F - 31     :REM POINTER MUST BE 1,33,65,97, ETC
70 PRINT CHR$(34);                :REM QUOTE MARK
80 FOR J=1 TO 32
90 GET#1,X$: PRINT X$;            :REM PRINT DETAILS OF ONE FILE ENTRY ONLY
100 NEXT J
110 GOTO 50
```

M-R

MEMORY-READ command.

M-R enables the RAM and ROM in DOS to be read, one byte at a time, by the CBM. This has applications in disassembling DOS, peeking RAM and zero-page, looking at BAM as it is held in memory, and so on. A new M-R command must be issued for each new byte.

Syntax: PRINT#15,"M-R" followed by two bytes only. These constructions are acceptable: PRINT#15,"M-R";CHR$(8)CHR$(0) and PRINT#15,"M-R"CHR$(123)CHR$(7) and PRINT#15,"M-R" + "q" + "%" ,among others. The two parameters which follow "M-R" are the low and high bytes respectively of the DOS address to be peeked. Note that the form "M-R" is the only accepted form in which this command can be issued. The block commands allow "BLOCK-ALLOCATE" for example , or other words with the initials B-A.

Examples: The first example returns the value of any byte in any location in the IEEE part of DOS (i.e. not the disk controller's locations). I've written it as a subroutine, so that it can be added to a BASIC disassembler and used to disassemble DOS ROM in hardcopy form. (This is easier than modifying a machine-code disassembler, and not all that much slower with a printer). Assuming channel 15 is open, the byte held by DOS in location QQ is returned in QQ. So, QQ=63030 returns from this routine with QQ=PEEK(DOS location 63030), for example. QQ doesn't signify anything particular; I chose it only because it's unlikely to be a variable used elsewhere in a program.

```
10000 REM ON ENTRY, QQ = LOCATION (0-65535); QQ RETURNS AS THE DOS PEEK
10001 REM
10010 PRINT#15,"M-R" CHR$(QQ - INT(QQ/256)*256) CHR$(QQ/256):REM LO THEN HI BYTE
10020 GET#15,QQ$:  IF QQ$="" THEN QQ$=CHR$(0)
10030 QQ = ASC(QQ$):  RETURN
```

The second example peeks into a region of RAM in which the current disks' directory entries are held. (For a memory map, see a few pages forward). This area will of course be peekable by the disassembler too, but this program prints it in readable form:

```
100 INPUT "WHICH BUFFER? (1=$4200, 2=$4300); B    :REM B SHOULD BE 1 OR 2
110 INPUT "WHICH DRIVE?   (1= -40 , 2=8050) ; T    :REM T SHOULD BE 1 OR 2
120 IF T=1 THEN T=144                              :REM STARTING-POINT OF
130 IF T=2 THEN T=6                                :REM   NAMES AND IDS
140 FOR J=0 TO 19
150 PRINT#15,"M-R";CHR$(T+J);CHR$(65+B)            :REM FETCHES AND PRINTS DISK
160 GET#15,X$                                      :REM NAME AND I.D.
170 PRINT X$;
180 NEXT J
```

M-W

MEMORY-WRITE command.

M-W enables data to be placed into the disk unit's RAM. This means machine-code programs can be written to reside within the disk unit. In principle this has many uses, including hardware control and alternative software to operate the disk, but in practice its use is rather restricted, since detailed knowledge of the working of the disk units is not widespread. Additionally, the different ROMs are largely incompatible, so that general-purpose machine-code is made more difficult to write. And there is not a great deal of RAM. Each 'M-W' command can write 34 bytes at most.

Syntax: PRINT#15 followed by a character string of this form:
M - W start address (low byte) start address (high byte) number of bytes data.
For example, the following formats are accepted:

```
X$="M-R" + CHR$(0) + CHR$(16) + CHR$(1) + CHR$(0): PRINT#15,X$
PRINT#15,"M-R"CHR$(18)CHR$(0)CHR$(1)CHR$(1)
```

where the first puts a null byte into $1000, and the second puts CHR$(1) into the zero-page location $12. (This in fact represents a drive number). Like M-R, this command has no expanded form.

Examples: The first example converts an 8050 disk unit number into a new value, by poking two zero-page locations. Typically, this is used to copy from a 4040 drive to an 8050, by DLOAD "BASIC PROG",D1,U8: DSAVE "BASIC PROG",D0,U9 and other similar commands.

```
PRINT#15,"M-W"CHR$(12)CHR$(0)CHR$(2)CHR$(9+32)CHR$(9+64) :REM NEEDN'T BE 9.
```

The next example is more complicated; it shows how machine-code can be stored in the disk buffers. Buffer number 2 is used (0 and 1 are used by DOS) and the code is executed by the internal (not IEEE) processor, which peeks 16 bytes from that processor's addressable memory. The result is stored in the RAM shared between both processors. It occurs in 'DISK MEMORY DISPLAY' by Jim Butterfield.

```
110 DATA 77,45,87,0,18,16,162,0,189    :REM MACHINE-CODE BEFORE THE ADDRESS...
120 DATA 157,64,6,232,224,16,208,245,108,252,255 :REM ... AND AFTER
130 FOR J = 1 TO 9: READ X: C$ = C$ + CHR$(X): NEXT
140 FOR J = 1 TO 11:READ X: D$ = D$ + CHR$(X): NEXT
315 REM U=LOW BYTE, V=HIGH BYTE, OF DISK PROCESSOR LOCATION TO BE PEEKED
320 PRINT#15, C$;CHR$(U);CHR$(V);D$                :REM CODE NOW IN PLACE
```

The string C$ contains M-W followed by 0 and #$12, which is address $1200 stored with its bytes in reverse order. The next byte is 16 decimal; this is the number of bytes to be written to memory. The data statements have only 14 further bytes; the remaining two are supplied in program-line 320 by CHR$(U) CHR$(V). The effect of the code is as follows:

```
$1200 LDX #$00
$1202 LDA ADDRESS,X
$1205 STA $0640,X
$1208 INX
$1209 CPX #$10
$120B BNE $1202
$120D JMP ($FFFC)
```
     and this can be checked by looking at the decimal equivalents of the hexadecimal listing. This routine can also be written into a BASIC disassembler, so the workings of the disk processor can be examined.

M-E

MEMORY-EXECUTE

Jumps to the specified location in the IEEE processor's address space, and executes the code it finds there. This may be a standard ROM routine, or user-written code which has been stored in RAM with M-W. ROM routines can be found by disassembling DOS or by reading other peoples' programs.

Syntax: M-E has only two parameters, the low and high bytes of the execute location. PRINT#15,"M-E" CHR$(231) CHR$(211) is a typical example. There is no alternative expanded form of M-E.

Examples: This command calls the hardware reset address in DOS:

     PRINT#15,"M-W" CHR$(0) CHR$(18) CHR$(3) CHR$(108) CHR$(252) CHR$(255)
     PRINT#15,"M-E" CHR$(0) CHR$(18) : REM PERFORMS INDIRECT JUMP TO ($FFFC)
As we shall see this is equivalent to PRINT#15,"UJ".

The error LED and DS$ are operated by a routine at $D925 (4040) and $EEB3 (8050). So   PRINT#15,"M-E" CHR$(179) CHR$(238)   :REM PROCESS DS$ FOR 8050

The routine in the directory processing which computes the total number of blocks free from BAM, and stores the result in the appropriate position in the directory, is at $DB34 (4040) and $D3E7 (8050). So again a routine like the following may be used from BASIC to call the subroutine:

     PRINT#15,"M-E" CHR$(231) CHR$(211)
This may be followed by the routine in M-R to print the directory, which should be updated to reflect the correct total of free blocks.

     The memory-execute command leads naturally to the last (documented) special DOS commands, of which there are 10, all beginning 'U', and referred to as user-commands or user-defined commands. Their function is exactly the same as M-E, except that no address is specified; it is implicit instead in the command. For example,

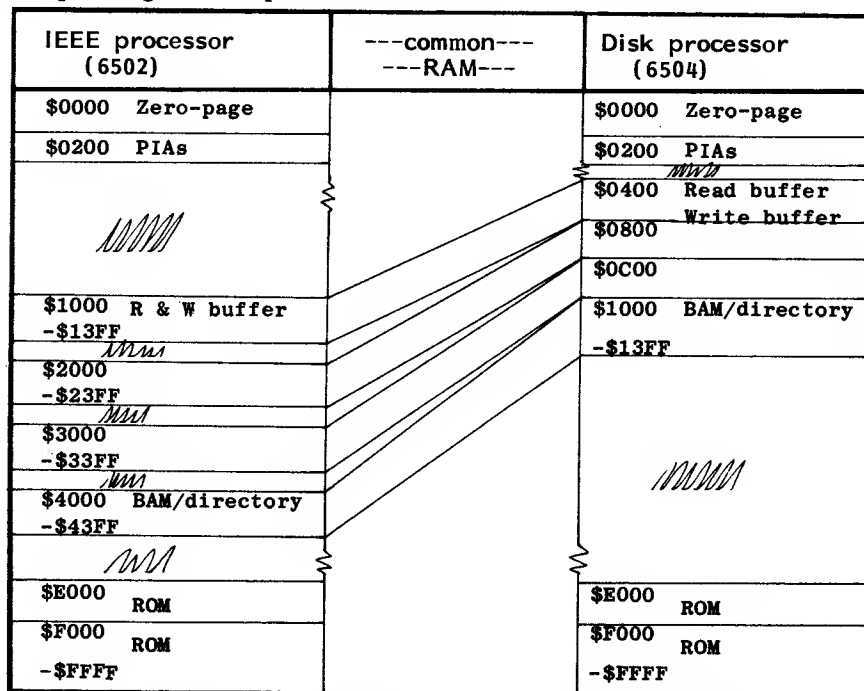   PRINT#15,"U3"   and   PRINT#15,"M-E" CHR$(0) CHR$(19)

are identical in their effects; each jumps to $1300 in DOS RAM and executes whatever code has been written there. As the table indicates, each address is separated from its neighbour by three bytes, so the intention is to use the commands with a jump table - e.g. $1300 JMP (FFFC) / $1303 JMP $EEB3/ $1306 JMP $1200 and so on. As we've seen already, U1 and U2 are exceptional - they are used in place of B-R and B-W, and have channel, drive, track and sector parameters in their command strings. Note that some of the jump addresses, in the early disk drives, are to non-existent addresses in expansion ROM. This is an error, since these addresses, if required, can always be accessed from the RAM jump table, e.g. by $1309 JMP $D008, which keeps more options open. Later disk units have a uniform jump table in the third buffer, perhaps because the earliest buffer which isn't allocated by DOS, that from $1200-$12FF, is a popular location for machine-code routines. Obviously, considerable knowledge of machine-code, and of the workings of CBM disk units, is necessary to use these commands fruitfully. Some commercial software, for example, has several routines of this sort to read and write sectors in non-standard ways to disk, as a security measure.

UA – UJ

User-defined jump addresses + NMI and RESET vectors.

COMMAND	FUNCTION	
U1 or UA U2 or UB	B-R (BLOCK-READ) B-W (BLOCK-WRITE)	
	JUMP ADDRESS TABLE	
	2040/3040	4040/8050
U3 or UC	$1300	$1300
U4 or UD	$1303	$1303
U5 or UE	$1306	$1306
U6 or UF	$D008	$1309
U7 or UG	$D00B	$130C
U8 or UH	$D00E	$130F
U9 or UI	$D0D5	$10F0 (=NMI)
U: or UJ	Power-on	Power-on

Notes on direct-access programming The memory-map of CBM disk units is made more
complicated by the presence of two processors, a 6502 which handles the IEEE comm-
ands, and a 6504 to control the disk. These have RAM in common, as might be expect-
ed, consisting of 4K in total, or 16 'pages' of 256 bytes. Most of these are buffers for
input and output to the disk; several hold BAMs and directory information, although,
because of the varying storage capacities between the units, the number and arrange-
ment of these buffers differs. This diagram shows the main features of the memory
map, including the differently-numbered RAM as between the two processors. Note the
way that the same RAM shows itself differently to each of the processors; if you
refer to the example machine-code under M-W, you'll see that 16 bytes of the disk
processor's ROM or RAM are put into $640 and the following locations. The memory-map
shows that $1200, from the disk processor's point of view, is $600, so that the piece
of code and the transferred bytes both occupy the same buffer of $1200-$12FF or $600-
$6FF depending on the processor.

IEEE processor (6502)	---common--- ---RAM---	Disk processor (6504)
$0000 Zero-page		$0000 Zero-page
$0200 PIAs		$0200 PIAs
		$0400 Read buffer
		$0800 Write buffer
		$0C00
$1000 R & W buffer -$13FF		$1000 BAM/directory -$13FF
$2000 -$23FF		
$3000 -$33FF		
$4000 BAM/directory -$43FF		
$E000 ROM		$E000 ROM
$F000 ROM -$FFFF		$F000 ROM -$FFFF

This diagram is adapted from some comments by Jim Butterfield on the early disk units.
It is not intended to be to a detailed map, but to give the general layout of the system.
It is not to scale.

The IEEE processor, as we've seen (see M-R), can have its accessible memory disassembled very easily; the easiest method is to patch a BASIC disassembler and take a hard-copy printout. When exploring RAM, a test program can be used to poke and read back memory locations; if the new value is retained, the location must be RAM. This is thorough, but painfully slow in BASIC. The Disk processor is less accessible, and requires knowledge of the working of the read buffer to extract the information. (It is, of course, also possible to disassemble ROM by taking the chip from its socket and using alternative hardware). The key to this is location $1004 ($0404 to the disk processor itself). If this location has its high bit set, the disk processor goes into action, either reading/ writing the disk or executing code. When the location is reset the IEEE part of the operation knows the operation is over, and collects its results from RAM or performs the next operation. The following short piece of BASIC, inserted into the routine in M-R, causes BASIC to disassemble the internal, disk, machine code.

```
10004 PRINT#15,C$;CHR$(QQ - INT(QQ/256)*256);CHR$(QQ/256);D$
10006 PRINT#15,"M-W";CHR$(4)CHR$(16)CHR$(1)CHR$(224)
10008 PRINT#15,"M-R";CHR$(4)CHR$(16): GET#15,QQ$: IF QQ$=CHR$(224) GOTO 10008
10009 QQ = 4672
```

Where C$ and D$ are the machine-code strings in M-W (q.v.). Line 10004 puts the machine-code routine into the common RAM; Line 10006 pokes 224 into $1004; line 10008 waits until that location indicates that the routine has been executed. The resulting bytes are deposited in $640ff of the disk processor's RAM; this is the same as $1240 ff or the IEEE processor. Hence line 10009 sets address $1240 (=4672) and peeks it using the ordinary M-R command. One character only is taken from the buffer so the code in C$ and D$ is over-complicated for this routine. Unfortunately, because of ROM variations, this routine mayn't operate without some changes: the 2040 unit for instance uses JMP $FEC1 in place of JMP (FFFC) at the end of D$, i.e. the data statements end ...,76,193,254. At the time of writing I don't have a complete list of corresponding addresses for all disk ROMs.

We've seen, in M-R, how to find the buffers which contain the BAMs. Some of RAM is mappable as it is with BASIC ROM and RAM; for example location $0282 controls the LEDs on the disk units, bits 3,4, and 5 determining whether drive 0/ drive 1 / central LEDs are lit. $12 holds the drive number, $2B the track, and so on.

Utilities Some disk utilities are obtainable; most of them are rather disappointing. Instead of, say, general-purpose disk de-corrupters or index sequential files, they tend to perform comparatively trivial operations like reading disks' i.d.s or changing names of disks. As an example of the kind of thing that could be done, there is a BASIC routine in Compute! (Mar.'81) by D L Cone, printed in that journal's rather peculiar typographic style, which has several CBM disk recovery routines, usable even if the directory has been erased, relying on the track and sector links. Other magazines (e.g. Liverpool Software Gazette) have had similar things. Track and sector routines which interpret what they find aren't hard to write. Routines to find unreadable sectors, and rewrite them so as to de-corrupt a disk, are possible, and can be useful if for example a diskette won't initialise. There are considerable possibilities along these lines; at the most advanced, routines to report the format of non-standard disks could be valuable.

There are a number of utilities designed to facilitate operations which ought to be automatic with the system, but which have residual bugs or problems, or are simply not very easy to carry out. The programs 'COPY ALL' and 'COPY/ALL' for instance,* available from user groups and some Commodore dealers, are intended as convenient alternatives to BASIC 4's COPY command. The latter is designed for all file types, including relative files. As new DOS ROMs are issued, such utilities should become outdated, and there is a chance that they may cease to work with different configurations.

The comparatively autonomous 'intelligence' of these units should be borne in mind. For example, with several different disk units connected to the same computer, copying between units becomes fairly straightforward; the units are reconfigured by software so that their unit numbers are different (see M-W for an 8050 example), so transfer of programs between disks, followed by backups within the units, can be performed. The disk drives can be disconnected while a backup is taking place.

---

*COPY/ALL (see CCN vol.3 #10, Nov.'81 for listing) replaces the earlier COPY.ALL

6.6 Machine-code programming with CBM disk drives*

Files: opening, reading, and closing Let's start with a fairly simple example, which reads sequential files and displays the result by poking it into the screen.

Suppose drive 0 has a sequential file called 'DATA' on its diskette.

After OPEN 2,8,3,"0:DATA,S" logical file #2 is open for reading. (Different numbers for the logical file and secondary address have been chosen to make the machine-code's operation clear).

```
$028C       LDX #$02    ;LOGICAL FILE NUMBER
$028E       JSR $FFC6   ;SET INPUT DEVICE - ANY BASIC ROM
$0291       LDY #$00
$0293  L1   JSR $FFCF   ;GET CHARACTER FROM DISK - ANY BASIC ROM
$0296       STA $8000,Y ;POKE CHARACTER TO SCREEN TOP
$0299       INY
$029A       BEQ OUT     ;256 CHARACTERS DISPLAYED
$029C       LDA $96     ;TEST STATUS BYTE (ST)
$029E       BEQ L1      ;NOT END-OF-FILE
$02A0  OUT  JMP $FFCC   ;RESTORE DEFAULT DEVICES
```

```
.:  028A        A2 02 20 C6 FF A0
.:  0292  00 20 CF FF 99 00 80 C8
.:  029A  F0 04 A5 96 F0 F3 4C CC
.:  02A2  FF
```

SYS 652 reads and displays 256 characters from the file on the top of the screen, or fewer if end-of-file is found (it is signalled by ST, the contents of $96). Because the bytes are poked, carriage return appears as 'm' (the 13th. letter of the alphabet).

(1) The routine prints ?FILE NOT OPEN ERROR if the file (logical file #2) isn't open.

(2) Each SYS 652 reads and displays the following 256 characters of the file.

(3) CLOSE 2 finally turns off the LED and deallocates the channel.

(4) DS is not checked. (We'll see later how this is done).

Note that on branching to OUT, Y holds the character-count; this is usually zero, and pokes to the screen as '@'. A holds ST. Either or both these figures can be printed as numerals, for example using the routine that prints a line-number, which can be found at the end of the reset sequence when it prints '31743 bytes free' or whatever figure is its RAM. In BASIC 4, X holds the value, LDA #00/ JSR CF83 prints it in decimal.

Many cosmetic improvements can be made to the output; [CLEAR], CHR$(147), can be printed to the screen, for instance. The file may be checked for carriage return characters, and output one record at a time. The output routine ($FFD2) can be substituted for screen pokes. The stop key can be tested for, and so on.[2]

Can a file be OPENed using machine-code? Obviously this must be possible, since BASIC itself operates exclusively with machine-code. This example is one way of doing it:

SYS 634 "0:DATA,S" with

```
$027A  LDA #$02
$027C  STA $D2     ;LOGICAL FILE NUMBER
$027E  LDA #$03
$0280  STA $D3     ;SECONDARY ADDRESS
$0282  LDA #$08
$0284  STA $D4     ;DEVICE NUMBER
$0286  JSR $F53C   ;CHECK COMMAND STRING & PUT IN BUFFER ($F4FD BASIC 2)
$0289  JSR $F563   ;OPENS FILE WITH THESE PARAMETERS ($F524 BASIC 2)
$028C  LDX #$02    ETC. AS ROUTINE ABOVE
```

This routine now opens the file and displays the first 256 characters; subsequent SYS calls to 652 continue to read the file.

```
.:  027A  A9 02 85 D2 A9 03 85 D3
.:  0282  A9 08 85 D4 20 3C F5 20
.:  028A  63 F5
```

---

*It seemed best to me to put this section in Chapter 6, although strictly it is out of sequence. Programmers not familiar with 6502 code should skip to the next section.
[2]There's a good example, using the screen-scroll routine, by R Davis in CCN,Vol.3, #5.

A file can be closed from BASIC using the address in FFC3 ('CLOSE') in association with a routine to fetch parameters from BASIC. Usually it is easier to avoid the input, simply loading the accumulator with the logical file number and entering CLOSE 5 bytes further on. In BASIC 4, for example, LDA #02/ JSR $F2E2/ continue closes logical file #2. BASIC 4 also has DCLOSE, which closes all files without needing file numbers. A slightly tedious piece of code can make CLOSE transferrable between BASICs, by computing the indirect address of CLOSE and adding 5, as in the following example, in which A is assumed to hold the file number:

```
        TAY             ;SAVE FILE NUMBER
        CLC
        LDA  FFC4       ;LOW BYTE OF ADDRESS OF CLOSE
        ADC  #05
        STA  FCL+1
        LDA  FFC5       ;HIGH BYTE OF ADDRESS OF CLOSE
        ADC  #00
        STA  FCL+2
        TYA             ;RECOVER FILE NUMBER
FCL JMP FCLOSE          ;JUMP TO ADDRESS + 5
```

Programs and blocks of RAM: loading and saving from machine-code   As we have seen in the section on program files, BASIC programs and RAM dumps are held in the same way on disk (and on tape), namely with the text preceded by two bytes which hold the load address. The ROM routines to load and save naturally use the true values of these parameters, but the machine-code programmer has the further option of putting in alternative addresses, so that routines may be relocated. This is not a facility that is much used, but remains an interesting possibility. In BASIC 4, DLOAD checks its parameters before entering LOAD. We'll thus consider only LOAD ($FFD5). This is a BASIC keyword, and assumes a BASIC program; this means that variables will be CLRed and so on. The monitor's .L command does not assume this. In fact the main part of LOAD is a subroutine used by both these commands, at $F322 (BASIC 2) or $F356 (BASIC 4). To use this routine, the following parameters must be correctly set; the named file will load in the normal place, but no pointers will be reset.

```
$D1 holds length of filename
($DA) points to start of filename (e.g. to 'DATA' or 'F1' or 'PRG*' in RAM)
$D4 holds device number - usually #8
$9D holds 0. This is the LOAD/ VERIFY flag; 1 means verify
$96 holds 0. This is the status byte (ST).
```

After these preliminaries, LOAD's subroutine is called. ST can be used to test for a successful load; both ROM loads AND ST with #$10 to test for a load error; the bit should be low, so ST AND #$10 <> 0 signals an error.

In order to load a block of data held as a program file to any part of memory, it is only necessary to simulate that part of LOAD which fetches the 2-byte load address from the file. Supposing that the list of parameters above has been set, we need:

```
        LDA  #60
        STA  D3         ;SECONDARY ADDRESS 0
        JSR  F4A5       ;SEND NAME TO IEEE.  (F466 IN BASIC 2)
        JSR  F0D2       ;SEND 'TALK'.       (F0B6 IN BASIC 2)
        LDA  D3
        JSR  F193       ;SEND SECONDARY ADDRESS    (F128 IN BASIC 2)
        JSR  F1C0       ;GET LOW BYTE OF ADDRESS   (F18C IN BASIC 2)
        LDA  $96        ;CHECK ST'S BYTE 1, I.E. 2ND FROM RIGHT
        LSR  A
        LSR  A
        BCC  CONT       ;ST OK
        JMP  F3C1       ;ABORT FILES, PRINT ?FILE NOT FOUND ERROR (F56E IN BASIC 2)
CONT JSR  F1C0          ;GET HIGH BYTE OF ADDRESS. BOTH BYTES ARE IGNORED.
     JSR  F391          ;REJOIN LOAD (WITHOUT PRINTING 'LOADING ...').  (F355 IN BASIC 2)
```

The named file will now be loaded from ($FB) onward in memory. This address is normally set from the two bytes which are thrown away by the machine-code routine. So the contents of $FB and $FC must also be set by the introductory routines.
    The code may be treated as a subroutine (e.g. followed by RTS) or used in-line.

SAVE's address in the kernel jump table is FFD8. It's construction is similar to LOAD, but easier to follow, since the BASIC version doesn't require to be processed, as it is after LOAD. Its general layout is:

```
S1    JSR GETPAR; GETS NAME, LENGTH OF NAME, DEVICE, SEC. ADDRESS FROM BASIC
S2    JSR STTEND; SETS (C9) AND (FB) TO END OF BASIC / START OF BASIC
S3    LDA D4    ; CHECKS DEVICE NUMBER, AND BRANCHES ACCORDINGLY:

                ; IEEE SAVE - THIS INCLUDES CBM DISKS

                ; CASSETTE TAPE SAVE
```

The subroutine at entry-point S1 fetches the same parameters as LOAD, and sets the same default values, with the exception of $9D, the load/verify flag, which it ignores. The subroutine at S2 simply stores the start of BASIC pointer in (FB) and the end of BASIC pointer in (C9). Obviously, the monitor routine .S "0:HELLO",08,1234,2345 bypasses these, storing its name pointers and address pointers in the appropriate locations, and going straight to S3. It is, in fact, acceptable to enter this routine a little later if the parameters are correctly set, since there's no point in comparing the device number with #3. Once again, we can modify SAVE if we choose. For example, instead of sending the load address as the first two bytes, we might send the horizontal and vertical screen positions at which to load the data; or we might not send any leading values; or we could send an extra code byte with some meaning of our own. In each case, there must be a corresponding LOAD routine to process our non-standard program file. The IEEE SAVE's major portion can be simulated like this:

```
         LDA #61
         STA D3
         JSR F4A5   ;SEND NAME TO BUS  (F466 IN BASIC 2)
         JSR F0D5   ;SEND 'LISTEN'     (F0BA IN BASIC 2)
         LDA D3
         JSR F143   ;SEND SECONDARY ADDRESS (F128 IN BASIC 2)
LOOP     LDA ??
         JSR F19E   ;SEND A CHARACTER TO THE FILE..
         B?? LOOP   ;..IN SOME SORT OF LOOP
END      BIT D3
         BMI RET    ;BRANCH TO RTS
         JSR F0D5   ;SEND 'LISTEN'     (F0BA IN BASIC 2)
         LDA D3
         AND #EF
         ORA #E0
         JSR F143   ;SEND SECONDARY ADDRESS (F128 IN BASIC 2)
         JMP F1B9   ;SEND 'UNLISTEN'        (F183 IN BASIC 2)
```

The piece of code at LOOP sends the data from the accumulator. The ROM save, of course, first sends two address bytes, then consecutive bytes in ascending order from the low to the high address. It is preceded, in effect, by code which sets the length of the name and its pointer, the device number, and ST=0. The secondary address is not needed; the routine sets it to 1 for write.

As the entries in SAVE and DSAVE (Chapters 5 and 7 respectively) show, it's possible to modify SAVE even from BASIC, by poking new 'start' and 'end' addresses; BASIC followed by machine-code (e.g. Supermon) can be saved as a BASIC program by raising the end-of-BASIC pointer to include the machine-code. A screen display can be saved as a program file by temporarily changing the start-of-BASIC to $8000 and the end-of-BASIC to $8400 or $8800, for a 40-column and 80-column screen respectively.

Sending a command string to a disk drive   Usually the command string is put into the input buffer, and sent from there to the disk drive. It is not however necessary that a string be held in a buffer; it can be incorporated into a machine-code routine or subroutine. The sequence of operations is this:

(i) Put #8 into $D4 and #6F into $D3 (device number 8 and sec. address 15).
(ii) Send 'Listen'.
(iii) Send the secondary address.
(iv) Send the command string sequentially. If the string is held in the input buffer, a zero byte will terminate it; this should not be sent.
(v) Send 'Unlisten'. The disk operation will take place now.

Command strings are sent to the disk drive in the older form of disk syntax, which the newer BASIC 4 commands (DLOAD etc.) also use, sending exactly the same strings after checking their newer syntax. The relative file handling is accomplished in the same sort of way, but the strings it sends are not recognised by DOS 1+. For a summary of these commands, see Chapter 15's tables of BASIC ROM; BASIC 4 from D839 lists the format of the commands, including the ,L construction which opens a new relative file, and the P construction for RECORD (see DA31). These commands permit relative files to be read and written from machine-code.

As an example, suppose the input buffer is used to take in the command; this means that the cursor will flash in the usual way, and characters will appear on the screen as they are entered at the keyboard. Carriage return terminates the input, and in fact puts a zero terminal byte as a marker at the end of the string, which is moved to $0200ff. This program sends a command input in this way to disk:

```
        JSR B4E2    ;INPUT TO BUFFER (C46F IN BASIC 2)
        LDA #08
        STA D4
        LDA #6F
        STA D3
        JSR F0D5    ;SEND 'LISTEN'   (F0BA IN BASIC 2)
        LDA D3
        JSR F143    ;SEND SECONDARY ADDRESS OF 15 (F128 IN BASIC 2)
        LDX #00
LOOP    LDA 0200,X
        BEQ END
        JSR F19E    ;SEND BUFFER CHARACTER (F16F IN BASIC 2)
        INX
        BNE LOOP    ;THIS ALWAYS BRANCHES
END     JSR F1B9    ;SEND 'UNLISTEN' (F183 IN BASIC 2)
        CONTINUE ...
```

For example, $1 sent by this code displays the directory of drive 1, and D1=0 performs a backup of drive 0 onto drive 1. S0:PROG* scratches files on drive 0 which begin PROG. And so on.

Disk status messages (DS$ and DS)   These are easier in BASIC 4 than earlier BASICs because routines exist which recognise DS and DS$. For example,

JSR C024 fetches DS and puts it in floating-point accumulator #1.

JSR BFC9 sets up the DS$ string in BASIC RAM, and sets the length parameter in $0D and the pointer to its start in ($0E), so that, for example:

```
        JSR BFC9    ;GET DS$ (BASIC 4 ONLY) AND SET UP STRING
                    ;SET HORIZ. AND VERT. SCREEN PARAMS. IF NECESSARY (SEE HTAB,VTAB)
        LDY #FF
LOOP    INY
        CPY $0D     ;COMPARE OFFSET WITH LENGTH OF STRING
        BEQ EXIT    ;AND EXIT WHEN EQUAL
        LDA ($0E),Y ;LOAD CHARACTER FROM STRING
        JSR FFD2    ;STANDARD ROUTINE TO OUTPUT A CHARACTER
        BNE LOOP    ;BRANCH ALWAYS (ACCUMULATOR LOADED WITH NON ZERO CHARACTER)
EXIT    CONTINUE...
```

DS$ can be tested at any time without printing the string by this sort of routine, used by HEADER to decide whether the newly-formatted disk was a 'bad disk' or not:

```
        JSR D991    ;GET DS$ (BASIC 4 ONLY)
        LDY #00
        LDA (0E),Y
        CMP #32
        BCS ERROR
        CONTINUE...
```

The rationale is that DS$ messages starting with 2 or more may be serious, while those with 0 or 1, i.e. 0,1,10,11,12,...,19   are not. In practice a number of messages are warnings rather than errors. The easiest way to test for messages which are not to be considered 'fatal' is probably to fetch DS as a floating-point number, convert it to an integer with a ROM routine, then test the low byte of the resulting integer.

BASIC<4 can read the error channel, and display DS$, using this routine:

```
        LDA #8
        STA D4    ;DEVICE NUMBER
        JSR F0D2  ;SEND 'TALK'      (F0B6 IN BASIC 2)
        LDA #6F
        STA D3    ;SECONDARY ADDRESS 15, WITH 'TALK'
        JSR F193  ;SEND SEC. ADDRESS (F164 IN BASIC 2) WITH 'TALK'
LOOP    JSR F1C0  ;GET BYTE FROM IEEE (F18C IN BASIC 2)
        CMP #D    ; RETURN?
        BEQ OUT   ; IF SO, GO OUT
        JSR E202  ; PRINT TO SCREEN - CAN USE FFD2 FOR OTHER DEVICES. (E3D8 IN BASIC 2)
        BNE LOOP  ;BRANCH ALWAYS AS ACCUMULATOR DOESN'T HOLD NULL
OUT     JSR E202  ; PRINT FINAL C. RETURN (E3D8 IN BASIC 2)
        JSR F1AE  ;SEND 'UNTALK'    (F17F IN BASIC 2 .. SLIGHTLY DIFFERENT ROUTINE)
        CONTINUE..
```

Routines like this will work with either BASIC 2 or BASIC 4, but in practice they are unlikely to be used with BASIC 4, since they are already built into ROM and can be used more economically than BASIC 2 allows by a direct call.

Throughout this section I have not referred to BASIC 1; in fact this is usable with disks, but to save space I have omitted its ROM entries and other details where the ROM and its RAM allocation differs from later BASICs. The tables in Chapter 15 can be used to help make these interconversions.

6.7 Compu/think disk drives

General   'Plug-compatibility' in the world of large computers refers to peripherals such as disk units or printers which may be substituted for those made by the computer manufacturer, the aim being to save time (if delivery dates are better) or money. This phenomenon has spread to the microcomputer industry. The more successful makers of micros have found innumerable suppliers of extra software, chips, interfaces and so on operating from outside their companies. Sometimes ex-employees help produce the stuff. This puts companies like Commodore in a slightly difficult position; their response is usually to encourage such alternatives unless they themselves have a similar product. Disk units are an important case in point, since they are expensive to buy and costly to maintain. High capacity disk units are quite a bit more expensive than the computers which use them. A number of non-Commodore disks have been produced and are sometimes met with. 'Novapac' disks for example were an early entrant into the field. Kilobaud-Microcomputing magazine ran an article on adding S100 disks to a PET (R Freeman, Jan.'80). Byte ran a series (June '81 ff.) on connecting disk units, using disk controller chips. New units, including hard disks and even modified full-size hard disks, continue to be sold. However, for a long time Compu/think disk drives were the only large-capacity storage system for the PET and CBM, so a separate section on them seems justified. There are also many operational differences from CBM disk units, and the comparison is often instructive. Commodore literature at present scarcely mentions alternative disks or printers to their own. There is little published material on these drives; Printout, and the PET Benelux Exchange magazine (in Dutch!), have run articles.

Physical size, capacity, operating system   Twin drives, typically MPI or PERTEC, are mounted vertically, under a standard three-sided sheet metal case. The appearance is similar to the 8061/ 8062 drives announced by Commodore, but smaller. This small size is achieved largely by filling the CBM with its electronics, consisting of a main board holding EPROM and RAM. There now exist single sided versions, single sided double density versions, and double sided, double density versions, using 5 1/4 inch disks, with storage capacities of 2 x 100 000, 2 x 200 000, and 2 x 400 000 bytes respectively, and a further 8 inch model, which is larger, and stores 2 x 800 000 bytes. These all operate with a disk operating system called Diskmon, which is compatible only with BASIC<4, because all the memory from $9000 to $BFFF is occupied by ROM and RAM with this system, not all of which is free with BASIC 4. There is a further Compu/-think product called BB-DOS which occupies ROM slot $9000-9FFF.

The operating system of these disks is quite portable; not much Commodore ROM is used, and even quite trivial features, such as the rectangular border around the screen are parameterised. In fact the same system is used in a different machine,

distributed in the U.K. by ACT Ltd. ('Applied Computing Techniques').* It is not an IEEE system, and does not have the autonomy of CBM drives; it is more accessible, and can be peeked and disassembled freely. It is initialised in the same way that tool-kits are, by a SYS call, in this case to $B000 (SYS 45056). This puts a wedge into BASIC's CHRGET routine, a widely-used technique to enhance BASIC - see Chapter 14. The function of the wedge is to check for the '$' symbol and interpret subsequent characters in its own way. Only the initial of the subsequent command counts; any following characters are ignored,[2] unless they are either a comma, semi-colon, or end of statement byte. This of course is fairly standard. The operating system shares many of the cassette tape system's locations; the zero page is a little overcrowded, so for example random numbers don't work correctly with the disk drives in operation. The usual simple Stop-key disable prevents the disk system from working. Two commands are designed for use with printers connected via the user port (not IEEE) and cables equipped with Compu/think's own interface; this presumably is an RS232 connection.

The format and organisation of the disks is far simpler than Commodore's, which must help explain its earlier arrival on the scene. The general system is less ambitious and appears to have fewer obscure bugs as a result of this. On the other hand, it has considerable limitations which have to be programmed around.

The drives have 40 tracks with 5 1/4 inch disks, 80 tracks with 8 inch disks. Each track has 10 sectors. Disk handling is by a Western Digital chip; single density drives have 256 bytes per sector, double density drives 512 bytes. The smallest unit which Diskmon uses is the track, i.e. 2560 or 5120 bytes, all of which is saved onto disk or read from disk. The directory is held on track zero; thus there is a maximum of 39 or 79 files and programs on any disk side. When a record from a file is read, the whole of this large buffer is filled, and a pointer used to find the actual record. Compared with CBM's 256 byte buffers, this is wasteful, and one of the serious limit-ations of this system is that only *one* file can be open at a time. For some purposes, for example adding to a file which is partly complete, this is fine. But it makes ordin-ary file updating difficult. The buffer may be updated by poking data in; since the position of this buffer (normally $9000-$A3FF) is known, this is a practical prop-osition, as we shall see. But it is not convenient. It is one reason why software using these drives tends to be slow, if it isn't well written: a file is repeatedly opened and closed, alternately with another (updated) file. As we shall also see, tracks can be loaded and saved into non-standard buffers, notably the top of RAM after the memory top has been lowered by poking. This can be a powerful technique. One buffer can hold an index, the other data, so an indexed sequential system may be implemented without too much trouble. There is an exception to the rule that tracks are the lowest common multiple of the system. Program files, which include machine-code dumps like CBM disks and tapes, are saved, from the starting address, in whole sectors. As long as a programmer knows this, it causes little difficulty. But it makes overlays more difficult, because RAM which is further up memory than the theoretical end of a pro-gram or machine-code routine is overwritten. For example, machine-code saved in the cassette buffers ($027A - $03FF is untouched by Diskmon) will, when loaded, reach from $027A - $0479 if the system is double-density, and if the code started at $027A. It cannot therefore be loaded by a BASIC program unless another program is loaded afterwards to reinstate BASIC in $0400 and the subsequent locations. Similarly, code of the type of Supermon or Extramon, stored at the top of memory, and saved to disk, reprints the top of the screen as it was when the code was saved, because screen RAM is adjacent to the machine-code, and some of it is stored with the code.

There are differences in style between these and CBM disks, which are as much a matter of taste as anything else. These disks save with replace without comment, where CBM disks print ?file exists error. Files are erased ('scratched') without comm-ent. Errors are reported with an error number and track and sector number, and the program crashes (i.e. stops, printing 'READY.'). There is no error message; the man-ual gives a vague indication of what the errors (as returned by the disk controller) mean. CBM's DS and DS$ system means that this need never happen, because an error

*As an example of the parameterisation, try this demonstration routine (BASIC 2) which
.:   033A   A9 00 A0 00 A2 00 4C 9B   repetitively prints different rectangles to the
.:   0342   BE   screen, using the machine-code subroutine:

0 FOR J = 0 TO 40: POKE 827,K: K=(K+1) AND 255: POKE 829,J*.6: POKE 831,J: SYS 826:
  NEXT: GOTO 0

[2]A funny story by Gerry Weinberg tells of the consternation he aroused in an enthusiast
showing off his typing-error-resistant system, by innocently typing something like
'RELETE'. It is possible, but unlikely, that $F in place of $D will erase a disk.

can always be detected with DS, and an appropriate request for action issued, e.g. 'Please put a disk in drive 0'.

Commands   The table lists Diskmon's additional commands with CBM equivalents. To simplify matters I've assumed drive 1 with Compu/think, drive 0 with CBM, although ranges of 1 - 4 and 0 - 1 respectively are normally valid. Most conversions should be fairly easy, but there are likely to be problems if multiple files are open with CBM disks, since Compu/think can't handle this situation. APPEND and some other commands are hard to translate into Compu/think usage. A few complications are omitted.

Compu/think	CBM BASIC 4	CBM BASIC < 4
$F,1 [No title or i.d. is assigned by $FORMAT]	HEADER "TITLE",D0,Iid	PRINT#15,"N0:TITLE,id"
$D,1	DIRECTORY D0	PRINT#15,"$0"
$S,1,"PROGRAM"	DSAVE "PROGRAM",D0	SAVE "0:PROGRAM",8
$S,1,"M/CODE","1234","2345"	.S,"0:M/CODE",08,1234,2345	
$L,1,"PROG"	DLOAD "PROG",D0	LOAD "0:PROG",8

$L;1,"PROG"   [Loads without affecting the running of the current program, unless it is overwritten. There is no easy CBM alternative].

$X,1,"PROG"	DLOAD then RUN	LOAD then RUN

[Loads and executes BASIC or machine-code. The first command with Compu/think should be CLR. Similar CBM instructions include DOS support's up-arrow function, the shift-stop key, and DLOAD or LOAD *in program mode only*].

$X;1,"PROG"   [Loads a new BASIC program and runs it, retaining previous variables' values, subject to the provisos on program length which also apply to CBM programs loaded from program-mode. Because of Compu/think's sector-loading principle, the new program must be sufficiently short that its final sector doesn't overlap the end-of-program pointers, and thus corrupt the stored variables].

$E,1,"FILE"	SCRATCH "FILE",D0	PRINT#15,"S0:FILE"

DATA FILE HANDLING:

Opening a new file:

$O,1,"W","FILE","PARAMS"	DOPEN#1,D0,"FILE",W	OPEN 1,8,8,"0:FILE,S,W"
or:	DOPEN#1,D0,"FILE",L50	OPEN 1,8,8,"0:FILE,L"+L$

[Note that $O has the same form with sequential and relative (or 'direct access') files. This trick is done by storing parameters in the directory entry, for use (with sequential files) as a store of (say) creation date, but, since this can be written as a record anyway, this is less important than its random access interpretation. Its eight bytes, for example I$=CHR$(7)+CHR$(1)+CHR$(55)+CHR$(1)+"XXXX", define the number of records in the complete file, followed by the record length; the rest is ignored. So the example allocates 311 records of length 263. The directory keeps a record of the tracks allocated to a file; records fill tracks and straddle over to the next track].

Opening a file which exists already:

$O,1,"W","SEQ FILE","PARAMS"	DOPEN#1,D0,"SEQ FILE",W	OPEN 1,8,8,"0:SEQ,S,W"
$O,1,"R","SEQ FILE",I$	DOPEN#1,D0,"SEQ FILE"	OPEN 1,8,8,"0:SEQ FILE"
$O,1,"D","REL FILE",I$	DOPEN#1,D0,"REL FILE"	OPEN 1,8,8,"0:REL FILE"

[These are, in order: (i) Sequential file opened for writing, (ii) Sequential file opened for reading, (iii) Direct access file opened for both reading and writing].

Reading records from/ writing records to / files of sequential and direct access type:

R,R	INPUT#1,R$	INPUT#1,R$
$R;N,R$	RECORD#1,(N): INPUT#1,R$	see RECORD# (Chap.7)
W,R	PRINT#1,R$	PRINT#1,R$
$W;N,R$	RECORD#1,(N): PRINT#1,R$	see RECORD# (Chap.7)

Closing a file:

$C	DCLOSE#1	CLOSE 1

Compu/think DOS has five further commands which don't appear in CBM BASIC, and several other features.

$B and $P ('BLIST' and 'PRINT') operate only with printers connected to the user port (not the IEEE port). $B,"NAME" prints a heading, page numbers, and a listing of BASIC with 50 lines to the page. It does not attempt to print screen editing characters in a readable form. $P prints a line, as PRINT# does to an IEEE port printer.

$G is intended for use with machine-code; after loading it, $G is intended to execute it, since the load address of the code is known. The effect should be identical to a SYS call to the first location of the code. However, this command has a bug! The relevant code is:

```
B53D  LDA #4C  ; JUMP OPCODE  (BASIC 2 VERSION)
      STA 27   ; LOCATION PRIOR TO START ADDRESS
      JSR 0028; SHOULD BE 0027
```

With BASIC loaded, the jump goes to $28-$29, holding pointer (low - high) to start of BASIC, and to $2A-$2B, holding the pointer (low-high) to the end of BASIC. So if BASIC is loaded, the code encountered is

```
0028 ORA (04,X)
002A -VARIES WITH END-OF-BASIC-
```

So, for example, if the program's length is varied so that PEEK(42)=96, $G will simply return and print READY. , because RTS has decimal value 96. When trying to run machine-code, the effect depends on the load address of the code, and is far more variable; usually it will crash.

$H clears the memory ('HALT'), having a similar effect to a power-on reset, except that locations below BASIC aren't affected. In this way the wedge to the system is retained.

$M is a memory-displaying command; $M,"1234 displays 190 bytes, in hex and in ASCII equivalent. It is rather slow in action; a fast-screen poke may be desirable. (See Chapter 5, PRINT). The following (completely relocatable) routines supplement this command: > and < step forward and backward to the adjacent section of memory,] and [jump about 1K, so the routine can be used to scan memory:

	BASIC 1									BASIC 2						
.:	033A	20	F7	B9	C9	3E	F0	F9	C9	.:	033A	20	0D	BA	C9	3E	F0	F9	C9
.:	0342	3C	F0	09	C9	5D	F0	14	C9	.:	0342	3C	F0	09	C9	5D	F0	14	C9
.:	034A	5B	F0	1A	60	18	C6	F8	A5	.:	034A	5B	F0	1A	60	18	C6	FC	A5
.:	0352	F7	69	84	85	F7	B0	E1	C6	.:	0352	FB	69	84	85	FB	B0	E1	C6
.:	035A	F8	90	DD	A5	F8	69	0E	18	.:	035A	FC	90	DD	A5	FC	69	0E	18
.:	0362	85	F8	18	90	D3	A5	F8	69	.:	0362	85	FC	18	90	D3	A5	FC	69
.:	036A	EE	85	F8	18	90	CA			.:	036A	EE	85	FC	18	90	CA		

The EPROM contains a serial number, identical to that on the disk drive. There is a security location. This is made possible by the use of the wedge. If you trace the jump address in $B000, you'll find the initialisation routine for the wedge, which gives the address to which CHRGET will now jump - e.g. B43E. The routine starting from here first saves the processor status and other things; the next operation is to check $040A (=1034) for the presence of #2A, which is usually an asterisk.* (It may not be; the following examples happen to include #2A in the link address or linenumber so if either format is accidentally used, the protection will unexpectedly come into play:

```
10 REMX
42 REM ANYTHING. The first line must be 2 tokens long; the second has line 42.

5 REMABC
10 REM THIS LINE MUST CONTAIN 27 TOKENS EXACTLY)
```

If #2A is detected, only $G, $H, or RUN are usable. $H erases BASIC; $G, as we've seen, is liable to crash, but this can be prevented by manipulating the program length and peeking location 42. Thus, only RUN is left. This is Compu/think's equivalent of the Auto-run modifications of the CBM to BASIC. It prevents LIST and also prevents accidental deletion of program lines by users entering numbers. We shall see how to insert the character in the section on reading tracks and using the directory.

---

*Some old versions use POKE 6,100 as their security location.

Bugs in Diskmon   The manuals record several types of bug; these chiefly relate to string handling of the parameters used in file handling. For example, in this situation

```
100 $0,0,"R","SEQ FILE",I$
120 $R,R$
```

a sequential file is opened for read, and a record (R$) is read. I$ *should* now take the value assigned to I$ when the file was originally written, and R$ *should* be the record as it was written to the disk file. In practice, either of these can go wrong; the best preventative is to assign the variables twice:

```
0 I$="": R$=""
122 R$=R$+""
```

These program lines correct the deficiencies which may exist.

Further bugs include the introduction of spurious characters at the end of long program lines, the failure of integer variable (e.g. X%) parameters to operate correctly, and the well-known IF problem, requiring the use of the short IF statement,

```
IF D=1 THEN $D,1    must be replaced by
IF D=1 THEN: $D,1
```

because the first version always performs $D,1 whether or not D=1. This is a problem with many wedge programs; provided it's remembered, it causes no problems. Sometimes the disk drives are left spinning after an error, perhaps a failure to find a file. The validation is not very thorough with these disk units. They can be stopped by a SYS call to the motor-off subroutine, or, more easily, by calling a directory with $D,0 or $D,1 etc. after which the drives stop.

Jump table of Diskmon functions and RAM and ROM memory map

```
B000  45056  Inserts a jump command into CHRGET
B003  45059  Returns from CHRGET - replaces processor status and enters CHRGOT
B006  45062  Reads directory into a short buffer (e.g. 1K. Depends on capacity).
B009  45065  Motor off, may also write updated directory to track 0.
B00C  45068  Clear 25 byte blocks in directory to erase file.
B00F  45071  Allocate track (read).
B012  45074  Allocate track (write).
B015  45077  Erase file - if it exists - from the directory.
B018  45080  Write buffer from $9000-$A3FF to disk track. (AFFB=drive, AFFF=track).
B01B  45083  Read track into  $9000-$A3FF.
B01E  45086  Write buffer from pointers, e.g. (FB) through (C9).
B021  45089  Read track into area defined by low/high pointers, e.g. (FB) through (C9)
B024  45092  Turn motor on.
B027  45095  Turn motor off.
B02A  45098  Save program to disk.
B02D  45101  Print line with $P or $B to parallel port printer.
B030  45104  Load program from disk.
B033  45107  Read directory; set relative track to zero. Then:
B036  45110  Increment relative track number, allocate track, read program track.
B039  45113  ASCII/hex. IF X<> 0, A becomes hex of X & Y; if X=0, X & Y become ASC(A)
B03C  45116  Perform $X (load program and run it).
B03F  45119  Continue with SAVE.
```

Some of these routines are usable from BASIC. This is not an exhaustive list of jump locations; those corresponding to $ commands - $R,$C,$D, and the rest - vary with versions, but can all be found after the initialisation routine to which B000 jumps. In BASIC2 (where peeking 45057=43) the comparisons and jumps start at B4C2; BASIC 1 (with peek of 45057=35) starts at B4B6. The pointers and absolute addresses of the buffers vary with the model; so does the length of buffer allocated for the directory track. The examples quoted above apply to BASIC 2's double-density version of Diskmon. Generally, it is not too difficult to locate subroutines which set pointers and set absolute values of track limits, because they take the form of loading the accumulator with a value/ storing it/ loading with another value/ and so on. Routines like $M and $D are fairly easy, because there's a batch of in-line coding, usually near tables of text like 'DIRECTORY' or 'MEMORY DUMP', and in association with a memory-map it is feasible to decode such routines. The most difficult to understand are those which deal with file opening, reading, writing, and closing.

8000-C000	A400-AFFF	AFD0-AFFF

| 8000 Screen RAM |
| 9000 5K buffer |
| A000 A400 Diskmon RAM |
| B000 Diskmon ROM |
| C000 BASIC ROM |

| A400 Unused RAM |
| A800 |
| AA00 Record area |
| AB00 Directory |
| AC00 area |
| AD00 (1000 bytes) |
| AE00 |
| AF00 Current |
| AFFF variables |

AFD0-AFD1	Block byte counter
AFD2	Switch (1 Read, 2 Write)
AFD3	Record byte count
AFD4	Counter: when 0, read/ write ends
AFD5-AFD6	Pointer to buffer lower limit
AFD7-AFD8	Pointer to buffer higher limit
AFD9	Disk read/ write, motor on/off
AFDA-AFDF	
AFE0-AFEF	Current file name (16 bytes max.)
AFF0	Current relative track number
AFF1-AFF8	Current file data or pointers for load or save
AFF9	Directory write switch (#10 = write)
AFFA	Command mode (#22=command mode)
AFFB	Drive number (1-8)
AFFC	Sector number (1-10)
AFFD	Disk command byte
AFFE	Register save area
AFFF	Track number (0-39)

BASIC programming with Compu/think disk drives   Before presenting programs which demonstrate file-handling with this system, let's look at some general programming features.

(i) Because of the possible bugs in string-handling, programs should begin with the CLR command. This may be combined with a memory-lowering POKE if several buffers are used; POKE 52,255: POKE 53,107: CLR for example reserves 5120 bytes of RAM from $6C00-$7FFF with a 32K system.

(ii) The directory of these disks provides no information about a file type. For this reason the manual suggests that sequential files are given names ending '.SEQ', program files names ending '.BAS', machine code files names with '.GO', and so on. This is entirely optional; programs will run without these codes, but readability is helped if some convention is used.

(iii) When a file is opened, either for reading or writing, the spindle motor is left turning, so that delays due to motor start are minimised. (The LED on the drive will stay on). To save wear, the motor can be turned off (see the jump table for the SYS command), and turned on only when needed. A delay of at least a half-second is advised in the disk manuals before writing or reading is attempted, so that the motor speed has time to stabilise at its correct value.

(iv) Diskettes need formatting before use; this is a write-only operation (which works even with head-cleaning diskettes). All diskettes should be formatted. (Some manuals have a strangely-worded warning which seems to imply the exact opposite). Disks do not need to be 'initialised' in the CBM sense. They have no names or other identification; in practice the label is a sufficiently good reminder.

(v) $C (close file) writes a null character, CHR$(0), onto the end of the file, and this can be used as a marker in the same way that ST=64 may be used with CBM disks. In either case, the alternatives are to record the number of records, perhaps in another file, or to write one's own end of file marker. Because of the arrangement by tracks, unclosed files are less of a potential hazard than with CBM drives.

(vi) Records are terminated by CHR$(13), the carriage return character.

(vii) In program mode, a 'file not found' flag exists, so that this type of construction is possible:

```
1000 POKE 44976,0: $0,0,"R","SEQ FILE"
1010 IF PEEK(44976)=255 GOTO 10000: REM ERROR-HANDLING ROUTINE
```

(vii) A number of routines, built into CBM disk DOS, are only available in this system as BASIC programs on disk. This includes file copy and disk copy utilities and also utilities to read disks recorded with different densities. Some versions of the file-copying program don't work. MONITOR is a long BASIC program which performs similar functions to SUPERMON and EXTRAMON, but more slowly. The tiny assembler has some directives (ORG, EXT, BYT, ADR, TXT, END) of which some are implemented. The screen won't scroll: 1531 MK=0/ 1535 MK=MK+1/ 1590 GOTO 1535/ delete 6512 remedies this.

RANDOM-FORMAT computes the parameters (i.e. low and high bytes of number of records and record length respectively) of a specified 'random-access' file, for those who haven't found out their secret. DISKCOPY provides duplicate or backup disks; the copy process proceeds one track at a time, and these are recorded on the screen. A track which is difficult to read or write causes a noticeable delay; this can be helpful in identifying weak disks. Several test programs are available. They are not exhaustive, and leave much of the diskette suface untested. There are also RAM testing BASlC programs. A few technical test programs (BOARDTEST, HAT,NR1) exist.

Demonstration programs showing file-handling These short programs are similar to those in section 6.3 on CBM disk drives. The commands are not very different from BASlC 4:

```
DEMONSTRATION OF A SEQUENTIAL FILE. WRITING TO DISK.   (COMPU/THINK)

10 $0,1,"W","SEQ FILE","ABCDEFGH"  :REM THE PARAMETER STRING CAN BE CHOSEN TO
                                        MEAN SOMETHING, IF YOU WISH
20 FOR J = 1 TO 10
30 X$ = "RECORD NUMBER" + STR$(J)
40 $W,X$
45 PRINT X$                    :REM REPEAT ON SCREEN
50 NEXT
60 $C                          :REM CLOSES THE SINGLE OBTAINABLE FILE
```

There is no statement corresponding to the CBM's APPEND; without elaborate work in the tracks holding the file data, it is therefore impossible to extend such a file once written. Also it cannot be read and simultaneously written, with corrections, to another file, because of the restriction of one open file only.

```
DEMONSTRATION OF A SEQUENTIAL FILE. READING FROM DISK. (COMPU/THINK)

100 $0,1,"R","SEQ FILE",I$       :REM I$ SHOULD ALREADY HAVE BEEN DEFINED; ...
110 FOR J = 1 TO 11             :REM I$ NOW SHOULD BE "ABCDEFGH" OR OTHER.
120 $R,X$: X$=X$+""             :REM THE PATCH MAY NOT BE NECESSARY
125 PRINT X$: IF X$=CHR$(0) THEN PRINT "END OF FILE FOUND": GOTO 140
130 NEXT J
140 $C                          :REM CLOSE
```

Note that old records aren't deleted; if end-of-file is ignored or not tested, it may be possible to read remaining records left from an earlier, probably longer, file.

DEMONSTRATION OF A 'RANDOM ACCESS' OR RELATIVE FILE: BOTH READING AND WRITING.

The first brief program allocates tracks and directory entries for a file called 'R FILE' which has a maximum of 4000 records of length 80. (This includes the final carriage return, and the record is stored with total length equal to 81 bytes).

```
10 I$ = CHR$(136)+CHR$(19)+CHR$(80)+CHR$(0)+"XXXX" :REM 19*256 + 136 = 5000, &
                                                       0*256 +  80 =    80.
20 $0,1,"W","R FILE",I$         :REM OPENS FOR WRITE LIKE A SEQUENTIAL FILE
30 FOR J = 1 TO 5000
40 $W,BL$                       :REM BL$ MUST BE A STRING OF 80 SPACES HERE
50 NEXT
60 $C
```

The point of writing spaces is to erase previous data from all the records. This can be time-consuming. Unlike the CBM equivalent, these files don't automatically enlarge themselves if asked to read a record beyond the current maximum number. However, like CBM relative files, they can be either written to or read from when open; they are not explicitly opened only for one or other of these operations. Consequently files of this type are usually more often used than sequential files. They can be updated while information is entered from the keyboard, or from other non-Compu/think files, such as tape files, or CBM disk units. But they cannot be updated easily from a file of Compu/think data without some programming effort, such as storing the new data in string arrays in RAM, perhaps rereading a file several times, while the relative file is closed, to read all the records in it. This is one of the penalties of the more simple operating system of Diskmon. The next page has a read and write demonstration program:-

```
5 R$="": I$=""
10 $0,1,"D","R FILE",I$              :REM "D" MEANS 'DIRECT ACCESS'
20 INPUT "READ OR WRITE"; RW$
30 IF RW$="R" THEN GOSUB 100: GOTO 20
40 IF RW$="W" THEN GOSUB 200: GOTO 20
50 $C: END                           :REM ONLY R AND W ACCEPTED

100 REM ** READ RANDOM ACCESS FILE **
110 INPUT "READ WHICH RECORD"; N
120 IF N<1 OR N>5000 GOTO 110
130 $READ;N,R$:  R$=R$+""             :REM $R AND $READ HAVE THE SAME EFFECT
140 PRINT R$                          :REM PRINT RESULT ON THE SCREEN
150 RETURN

200 REM ** WRITE RANDOM ACCESS FILE **
210 INPUT "WRITE WHICH RECORD"; N
220 IF N<1 OR N>5000 GOTO 210
230 INPUT "TYPE IN RECORD"; R$
240 IF LEN(R$)>79 GOTO 230
250 IF LEN(R$)<79 THEN R$=R$+" ": GOTO 250 :REM PAD WITH SPACES THE LAZY WAY
260 $W;N,R$                           :REM WRITE THE RECORD WITH C.RETURN
270 RETURN
```

Reading and writing tracks   The following BASIC routine (which is easily converted into machine-code) reliably loads any track into the normal buffer from $9000-$A3FF:

```
700 REM ** READ TRACK TR OF DRIVE DR INTO $9000-$A3FF (36864-41983)
710 POKE 45051,DR   :REM POKE DRIVE NUMBER INTO AFFB
720 SYS 45062       :REM MOTOR ON (ALSO READS DIRECTORY). B006
730 POKE 45055,TR   :REM POKE TRACK NUMBER INTO AFFF
740 SYS 45083       :REM READ TRACK INTO NORMAL BUFFER. B01B
750 SYS 45095       :REM MOTOR OFF - B027
760 RETURN
```

And the opposite process, of storing a track from $9000-$A3FF onto disk, is carried out using the write routine SYS 45080 (B018) in place of line 740's read routine. So a general read/write routine for this buffer might be:

```
700 REM ** BUFFER READ/WRITE: TR=TRACK, DR=DRIVE, RW$="R" OR "W"
710 POKE 45051,DR: SYS 45062: POKE 45055,TR
720 IF RW$="R" THEN SYS 45083
730 IF RW$="W" THEN SYS 45080       :REM IDEALLY, SIGNAL ERROR IF RW$=OTHER
740 SYS 45095: RETURN
```

It is also straightforward to load tracks from non-standard parts of RAM, and read them back - if required, into different RAM areas. All that's required is to poke the low and high addresses, and use a similar routine to those above, except that the routine which reads or writes is selected to bypass the allocation subroutine for these buffers. (For example, the slight difference on disassembly between $B018 and $B01E is taken up by this allocation routine). The low and high pointers are ($FB) and ($C9) respectively in BASIC>1. For example:

```
800 REM ** BUFFER READ/WRITE: THIS EXAMPLE USES $6C00-$7FFF
810 POKE 45051,DR: SYS 45062: POKE 45055,TR
820 POKE 251,0: POKE 252,108:    REM NOW (FC) HOLDS $6C00   [BASIC 1:247 & 248]
830 POKE 201,255: POKE 202,127:  REM AND (C9) HOLDS $7FFF   [BASIC 1:229 & 230]
840 IF RW$="R" THEN SYS 45089    :REM B021
850 IF RW$="W" THEN SYS 45086    :REM B01E
860 SYS 45095: RETURN
```

There is of course no problem in parameterising this routine further, so that entry of the lower limit of a buffer calculates the upper limit and pokes all four bytes. Note that a buffer need not hold 5120 bytes; the directory for example is loaded into a shorter buffer, typically 1000 bytes long. If these routines are new to you, try experimenting by adding input statements for drive and track numbers, calling a read sub-

routine, and displaying the result with, for example,
       PRINT CHR$(34);:FOR J = 36864 TO 41983: PRINT CHR$(PEEK(J));: NEXT.

As many as 7 buffers can be simultaneously stored in a 32K machine, at the expense of space for BASIC (7 buffers leave 1K, 6 buffers leave 5K, for example). There is considerable scope for machine-code searching of RAM and similar activities. Self-checking hashtotals can be introduced quite easily. Tracks may have their track number, drive number, and a hashtotal stored in three bytes at the end of the track; in this way, an entire disk can be tested for its data's self-consistency with a simple program to consecutively read and check each track. These possibilities suggest that several CBMs may be joined to one disk unit, since reading and writing can be made infrequent, but I suspect that the necessary boards with RAM and EPROM may not be available independently of the drives.

The directory track   Track zero, the outermost track, holds the disk's directory, unless the disk is used in some special way, when it may contain data. Each file has directory entries 25 bytes long. The entries are quite simple, and the following diagram illustrates a typical disk directory. Note that the file names have been printed in ASCII characters for readability, with ƀ representing a single space character, and the remaining values are represented by the PEEK value, again for readability.

-- FILE NAME --	Relative Track No.	Starting Address or Number of Rel. Records		End Address or Record Length		Rest of I$ or Seq.file parameter			
BACKGAMMONƀƀƀƀƀ	1	1	4	88	27	32	32	32	32
BACKGAMMONƀƀƀƀƀ	2	1	4	88	27	32	32	32	32
MERGE.GOƀƀƀƀƀƀƀ	1	0	48	0	61	32	32	32	32
UTILITY.DATƀƀƀƀ	1					32 51 32 52 56 49 32 32 [i.e. 3 481]			
GALLFILE.DIRƀƀƀ	1	236	46	6	0	32	32	32	32
---------------	OTHERS	---------------							
GALLFILE.DIRƀƀƀ	17	236	46	6	0	32	32	32	32
FREEƀTRACKbƀƀƀƀ	255	32	32	32	32	32	32	32	32

Each entry has (i) 16 characters of file name, padded with space characters.
      (ii) A relative track number. 'FREE TRACK' has a symbolic value of 255. All file tracks are numbered consecutively, starting from 1. Note that a file is assigned the earliest empty tracks, so a file may be interwoven with other files once some erasure of files has occurred.
      (iii) 8 bytes of parameter. There is no indication of the type of a file; a program be read as a random access file, or a sequential file loaded as machine-code. The result of such experimentation depends on the 8 parameter bytes, which are interpreted in three different ways, viz. starting and end address for machine-code and BASIC files (BACKGAMMON and MERGE.GO in the example) *or* number of records and record length for random-access files (GALLFILE. DIR) *or* a string of 8 user-assigned characters (UTILITY.DAT has the date of last update = 3 4 81 for instance).

The program BACKGAMMON starts at 4*256 + 1 = 1025, and ends at 27*256 + 88=7000. This is a BASIC program as its starting-point implies, although it could be machine-code. It is about 6000 bytes long, and therefore occupies two tracks. MERGE.GO starts at $3000 and extend to $3D00, small enough for 1 track. UTILITY.DAT is a short sequential file. GALLFILE.DIR is a very long direct access file; its parameters show it to have space for 46*256 + 236 = 12012 records of length 6 each. In fact, the system allocates space for 7 bytes in each of these records; this is a peculiarity of the system. 17 tracks are needed for the resulting file. Finally, we see a FREE TRACK entry; all unused tracks contain this. Very occasionally, the word 'bad' appears in reverse after a file name entry. The directory is printed by a special routine (follow $D to find it - e.g. BB39 in BASIC 2). This lists only filenames with relative track number 1, to avoid repetition. The number of free tracks is calculated simultaneously. It is interesting to note that because of this, files can be recorded in a way which

makes them invisible on the directory: if a file name is longer than 16 characters, the length is not checked, but the name overwrites the relative track number. So long as CHR$(1) isn't written in the 17th position, the file won't appear. This character should not be CHR$(255) or the file will be overwritten. Note that only 16 characters are compared during a load or save operation, so the first 16 characters must differ in some way between different files. (There is no system of default file names, like "*" or "BAS*" with CBM drives). So a program saved as "LENGTH=SEVENTEEN!" is invisible on the directory, but will load by $L,1,"LENGTH=SEVENTEEN".

The short program which follows analyses the directory track, printing the contents of each track, or of one single file name when this is found. For example, a disk has a program called 'PRINT PRICE LIST'; the program comes up with this:

```
NAME: PRINT PRICE LIST
```

ABS. TR#	REL. TR#	-LOW #RECS	PROGRAM R.A.FILE	HIGH- LEN-	SEQ.FILE I$ PARAM
16	1	1025	BASIC	11804	.
17	2	1025	BASIC	11804	.
18	3	1025	BASIC	11804	.

Showing the three tracks which hold the file, which is presumed by its start address to be BASIC, and which extends from 1025 - 11804. I$ is not important.

```
10 REM **DIRECTORY TRACK ANALYSER FOR COMPU/THINK
20 INPUT "WHICH DRIVE"; DR: IF DR<1 OR DR>4 GOTO 20
30 INPUT "FILE NAME"; F$
40 IF LEN(F$)>16 GOTO 30
50 IF LEN(F$)<16 THEN F$=F$+" ": GOTO 50
100 POKE 45051,DR: SYS 45062: POKE 45055,0    :REM TRACK 0
110 SYS 45083: SYS 45095                       :REM READ IT, CLOSE DRIVE

200 PRINT "NAME"; F$: PRINT
210 PRINT"ABS. REL.   -LOW PROGRAM HIGH-   SEQ.FILE";
220 PRINT"TR#  TR#  #RECS R.A.FILE LEN-  I$ PARAM"
230 DIM DE$(39)                                :REM FOR 40 TRACK UNIT

300 IN = ASC(F$)                               :REM INITIAL VALUE FOR FAST TEST
310 FOR DE = 1 TO 39: IF IN<>PEEK(36839 + 25*DE) THEN NEXT: GOTO 500
320 FOR DE = DE TO 39                          :REM BUILD TABLE OF NAMES...
330  K = 36839 + 25*DE
340  IF IN<>PEEK(K) GOTO 390                   :REM ... IGNORING IMPOSSIBLE ONES
350  FOR J = K TO K+15
360   DE$(DE) = DE$(DE) + CHR$(PEEK(J))
370  NEXT J
380 IF F$=DE$(DE) THEN N=1: GOSUB 450
390 NEXT DE
400 IF N=0 GOTO 500                            :REM FILE NOT FOUND
410 END

450 PRINT DE TAB(4) PEEK(J) TAB(10) PEEK(J+1) + 256*PEEK(J+2);
460 IF 1025 = PEEK(J+1) + 256*PEEK(J+2) THEN PRINT "  BASIC";
470 PRINT TAB(25) PEEK(J+3) + 256*PEEK(J+4) TAB(31);
480 FOR K = 1 TO 8: PRINT CHR$(PEEK(J+K));: NEXT K
490 RETURN

500 REM ** FILE NOT FOUND - LIST ENTIRE DIRECTORY
510 PRINT F$; "NOT FOUND": PRINT
520 FOR DE = 1 TO 39: K = 36839 + 25*DE: FOR J = K TO K+15
530  DE$(DE) = DE$(DE) + CHR$(PEEK(J)): NEXT J
540  PRINT " " DE; DE$(DE);
550 NEXT DE
```

A similar program can 'Unlist' programs, by poking an asterisk in the correct place. Or it may be modified to Re-list a protected program. Renaming of files is also easy, and some protection can be achieved by including cursor-control characters in a file name. An 'Unlist' program, too long to be included here, has to (i) search the directory for the specified name, typically by loading track zero into RAM and searching it; (ii) If the name is found, calculating the absolute track of the first relative track of the file, which is the first met with in the directory; (iii) loading this track, poking in 42, and finally saving back on disk. The program should begin with REM, preceded by not more than 4 tokens, or some other arrangement immune from the influence of a foreign CHR$(42). The business part of such a program is something like this:

```
1010 TR=DE        :REM TRACK NUMBER = DIRECTORY ENTRY
1020 RW$="R": GOSUB 700:  REM READ FIRST TRACK OF PROGRAM
1030 POKE 36873,42:       REM EQUIVALENT TO LOCATION 1034 OF BASIC
1040 RW$="W": GOSUB 700:  REM WRITE MODIFIED TRACK
1050 PRINT "UNLIST "; F$; "COMPLETE": END
```

Machine-code and RAM buffers This subsection deals with machine-code processing of data held in RAM buffers. Because of Compu/think's track-handling system, this is principally relevant with Compu/think, but is also usable with CBM disks, in which buffers of data can be loaded and saved by the save command .S and its BASIC equivalents and .L and LOAD.

It may be useful to erase a buffer; this short machine-code routine, which uses no zero-page locations and is consequently not immediately relocatable, puts zero bytes into all locations $9000-$A3FF. It can be modified to erase any set of locations whose lower and upper limits are of the form $xx00-$xxFF. SYS 800 runs it:

```
.:  0320 A9 00 AA 9D 00 90 E8 D0
.:  0328 FA EE 25 03 A9 A4 CD 25
.:  0330 03 D0 ED A9 90 8D 25 03
.:  0338 60
```

The next example is a routine which searches a buffer for a record. A form of indexing may be implemented in this way; a short code or identifier, followed by a record number, enables a two-stage operation to find the record by its code. The search routine uses a flowchart like this:
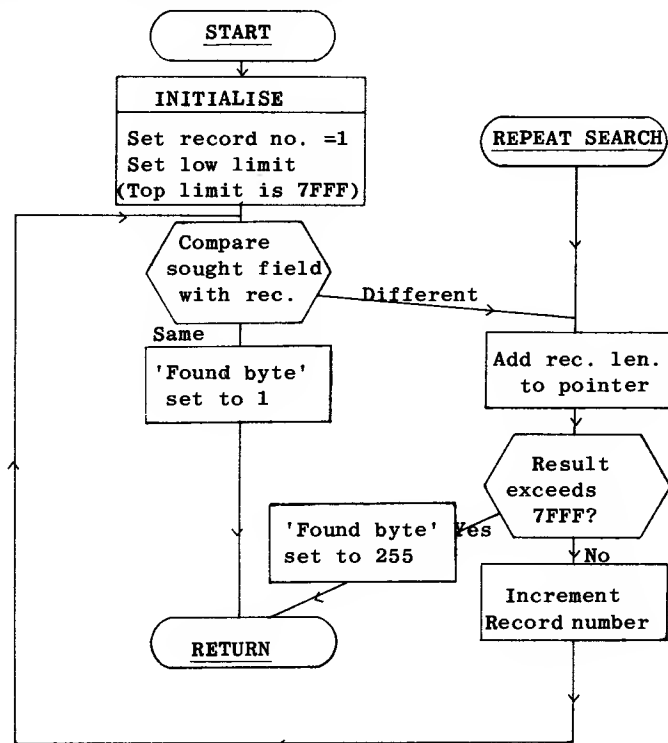
This is a sequential search: records are scanned from one end to the other. When the search is carried out (the routine takes a fraction of a second) a location, 847, is set with a coded value to show whether a match was found between a record and the field stored in RAM. A value of 1 shows it was found; 255 shows the search was not successful. An alternative entry point enables a search to be repeated, if a record exists more than once. The diagram illustrates the rationale behind this type of search. A subfield within a record - perhaps a single character only - can be sought. Given that the records are of uniform length, the important starting position is the point at which the comparison field begins; this need not coincide with the start of the buffer.

The locations used in this example program are:

```
832-846 Sought field. Maximum length is 15 characters.
847     Found byte. 1=Found; 255=Not found.
848-849 Current record number. Starts at 1.
850-851 Start point of search. Not necessarily equal to the start of the buffer.
852     Record length.
853     Length of sought field (1 - 15).
854-855 Current pointer into buffer.
```

```
←— Record length ——————————→←———————————————————→←————————————————————→
        Compared field              ←——————————→            ←———————————→
┌─────────────────────────┬──────────────────────┬──────────────────────┐
│ R E C O R D  N U M B E R 1│ R E C O R D  N U M B E R 2│ R E C O R D  N U M B E R 3│
└─────────────────────────┴──────────────────────┴──────────────────────┘
Start of
Buffer
```
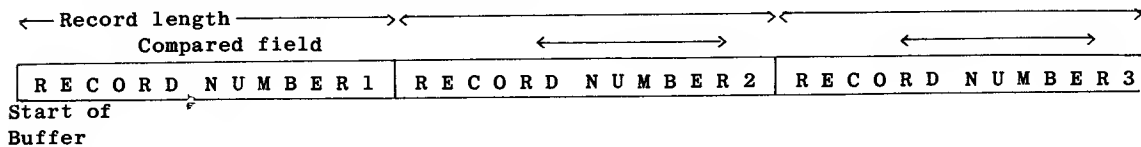
MACHINE-CODE SEARCH ROUTINE TO HUNT STRING IN BUFFER

```
1 FOR L=856 TO 967: READ M: POKE L,M: NEXT: REM SETS UP M/C IN CASS.BUFF.2
2 DATA 169,0,168,141,81,3,169,1,141,80,3,173,82,3,141,86,3,173,83,3,141,87,3
3 DATA 173,86,3,141,124,3,173,87,3,141,125,3
4 DATA 185,0,16,217,64,3,208,9,200,204,85,3,240,52,76,123,3
5 DATA 24,173,84,3,109,86,3,141,86,3,169,0,109,87,3,141,87,3
6 DATA 174,85,3,202,138,24,109,86,3,169,255,109,87,3,201,127,16,18
7 DATA 238,80,3,208,3,238,81,3,160,0,76,111,3
8 DATA 169,1,76,196,3,169,255,141,79,3,96
10 PRINT "[CLR]MACHINE CODE BUFFER SEARCH ROUTINE"
15 PRINT "[DOWN]LOCATIONS:": PRINT "---------"
20 PRINT " ROUTINE IS ENTERED AT $358 (=856)"
22 PRINT " OR AT $38C (=908) TO REPEAT"
25 PRINT "[DOWN] $340-$34E (= 832 - 846) HOLD THE SOUGHT STRING"
30 PRINT "[DOWN] BYTE $34F (847) IS THE CHECK BYTE"
31 PRINT "  AND HOLDS 255 IF NOT FOUND,": PRINT"  1 IF RECORD FOUND."
35 PRINT "[DOWN]ASSUMPTION IS THAT THE 5120 BYTES"
36 PRINT "FROM $6C00 (27648) TO $7FFF (32767)";
37 PRINT " HOLDRECORDS OF EQUAL LENGTH"
1000 REM ** ROUTINE HERE (EG COMPU/THINK TO LOAD TRACK OF DRIVE) **
2000 PRINT "[DOWN]DO YOU WISH TO EXAMINE THE BUFFER? ";
2010 INPUT "IF SOPRESS A KEY DURING PRINTOUT TO STOP"; YN$
2020 IF YN$="N" THEN 3000
2030 FOR L=27648TO32767:GETX$:IFX$=""THENPRINTCHR$(PEEK(L));:NEXT
3000 PRINT: PRINT: INPUT "SOUGHT ITEM:"; CT$
3010 FOR L = 1 TO LEN(CT$): POKE 831+L,ASC(MID$(CT$,L,1)): NEXT
3020 POKE 853,LEN(CT$)
3030 PRINT "ENTER TOTAL RECORD LENGTH (EG. INCLUDING RETURN)"
3040 INPUT "(REC. LENGTH=1 FOR COMPLETE SEARCH)"; R
3050 POKE 852,R
3060 INPUT "START OF SEARCH (EG 27648):";S
3070 POKE 851,INT(S/256): POKE 850,S-INT(S/256)*256
4000 SYS 856
4010 PRINT "[DOWN]CHECKBYTE CONTAINS "; PEEK(847)
4020 PRINT "[DOWN]RECORD NO (START=1) IS:"; PEEK(848) + 256*PEEK(849)
4030 IF PEEK(847)=255 GOTO 4060
4040 PRINT: IF PEEK(847)=1 THEN INPUT "REPEAT THIS SEARCH?"; YN$
4050 IF YN$="Y" THEN SYS 908: GOTO 4010
4060 GOTO 2000
```

A hashtotal routine (Chapter 10) and merge routine (Chapter 3) are also well adapted for use with these disks. Techniques involving indexed files, and other uses for merges, are not however for the faint-hearted.

There is no routine to turn off these disks once SYS 45056 has been issued. Any other routines using wedges, however, are unlikely to co-exist, unless the wedge is specially written to allow for the existence of other wedges. See Chapter 14 on this. However, a machine code routine to do this is easy to write. Because of the time spent processing the wedge, there may be noticeable time-saving, perhaps 20-30%. The same effect can be achieved by turning off the interrupt altogether, with

POKE 59411, PEEK(59411)-1 but the clock and keyboard are not updated/scanned if this is done, until POKE 59411, PEEK(59411)+1 restores the interrupt. This may be inconvenient, or it may not, depending on the type of program. A report program with no other function than to print out data probably doesn't need the keyboard on.

The BASIC>1 code which returns BASIC to normal (so $C for example causes ?SYNTAX ERROR) is this 14 byte relocatable routine:

```
.:  027A   78 A9 E6 85 70 A9 77 85
.:  0282   71 A9 D0 85 72 60
```

SYS 634 (with this location) replaces the first bytes of CHRGET with their normal values. This call can be made within a program; its converse, SYS 45056, can also be called in program mode, to reconnect the $ commands.

6.8 Problems, reliability, and maintenance

Problems and reliability Whilst the electronic circuitry of correctly-assembled computers is very reliable, it cannot be expected that disk and tape units, and other devices with moving parts, should be as error-free. This is not a problem exclusive to small computers. 'Head crashes' (where the recording head scratches the disk surface) are not unknown in large installations. For this reason, backup copies of data are almost invariably kept. There is also, of course, a possibility that human errors will occur. Disks may be mislaid, damaged, or demagnetised; or a software error may cause data to be deleted or overwritten; perhaps failure to follow some procedure will mean that a disk of useful data is lost. The purpose of this subsection is to give some perspective on these potential difficulties.

Minimising the chance of error The following general points apply to all microcomputer systems using diskettes. Hard disks ('Winchesters') are more reliable; nevertheless, in any applications where loss of data or programs would be inconvenient, similar precautions ought to be taken.

(i) Use some systematic copying method. A commonly-recommended technique is the 'grandfather-father-son' method, in which each disk has an earlier version preserved and a still earlier one. In this way, software errors or errors in the way data has been processed can be corrected by repeating a process on the original version. The point at which a copy is taken has to be decided on empirical and common-sense lines. Systematic labelling of diskettes helps. The 'header' feature with CBM disks, and the disk i.d., can be useful here. A log may be kept of dates on which copies were taken, and the processing which was performed on them. Important disks - i.e. any disk containing information difficult to replace and worth keeping - also need physical storage in a place where they will not be confused with other disks and are not likely to be damaged. They may, for example, be kept in lockable boxes, or in clearly-marked cases in a lockable drawer. Similarly it may be worth keeping copies in a different place - another room or another building - for security.

(ii) Ensure that all diskettes are storing data correctly. Since all the data is stored on diskettes, which are inherently a somewhat delicate medium, it makes good sense to have techniques to validate them. In the case of new diskettes, if they are destined to store valuable information, they should be tested thoroughly with a utility program which writes and reads back every location on the diskette's surface. This is worth doing even with diskettes which are warranted error-free. Most test programs aren't exhaustive, and confine themselves to opening a few files. Sectors or tracks of random data can be used. Alternatively, the bit patterns in RAM testing are suitable (i.e. 1010 1010 and 0101 0101. #$AA and #$55 and CHR$(170) and CHR$(85) are the hexadecimal and ASCII equivalents). Disks which hold data need a different approach. The point is that a user has a set of disks in envelopes which are unreadable in the

normal way, unlike card-indexes or ledgers, say. It is important that validation programs should exist which enable a disk to be checked for self-consistency of its data, providing some reassurance that its data does, in fact, correspond to its label. This precaution is quite often lacking in many systems, notably those from the cheap end of the market. But it is difficult to see how a user can have total confidence in a system lacking a verification facility. As an example of the type of thing I have in mind, a system's disks may be run through a program which calculates a hashtotal of a batch of data, and compares it with the same hashtotal previously stored on disk, reporting the results. Or a utility program may page rapidly through all the records, or a group of them, displaying the results. Or a routine may check the integrity of files by reading consecutive sectors through to the end. In this way, a diskette which perhaps was exposed to a magnetic field, or is otherwise suspect, can be checked without an actual program run. If this aspect of a system is thought out at the time of design, the subsequent effort is likely to be less than if it is introduced as an afterthought.

(iii) Don't overuse disk drives. Error rates are usually quoted as a proportion of disk accesses attempted. While this is a statistical artefact to some extent, it is true that drives which continually write data all over the disk's surface are more likely to put a sector in the wrong place than the same drives under conditions of less heavy loading. Other things being equal, it is likely to be good policy to cut down on disk use. For example, when relative files are opened, CBM's DOS allocates only enough disk space for the current records. If a relative file is intended to grow, it is best to generate the entire length of the file at the start, so that sectors will tend to be arranged in a tidy pattern, not interspersed with other data. A new diskette, free from the chaotic organisation of sectors resulting from many files being saved and later scratched, is a better vehicle for relative files for the same reason, that track-seeking movements are reduced. Data which is frequently reused may be better stored in RAM than repeatedly read from disk. This rule, however, is very dependent on other features of a system. If, for example, the chance of the computer being switched off or losing data in some other way is greater than the (small) chance of disk failure, then data should be stored immediately on disk.

(iv) If possible have a standby system. An advantage of microcomputers is that exact duplicate systems may be easily accessible, in the same organisation or user group. When they are, reciprocal agreements may be possible, so that even serious breakdown doesn't affect a system's work. Organisational quirks may make this more difficult to arrange than appears likely at first sight.

Summary of software bugs   These remarks apply to CBM disk units with DOS 1.x or DOS 2.x, including 2.5; new releases of DOS will make the comments obsolete, for those versions of DOS. At the time of writing, definite announcements on DOS bugs are infrequent from Commodore, and in the absence of fact, it it not surprising that rumour abounds.

(i) Write-protect tabs. CBM DOS detects the existence of a write-protect tab, which prevents an immediate write to the disk, as it is intended to. But a software bug in the internal processor leaves the head's write-gate enabled, so that as a diskette is searched by the head, a magnetic trace moves across the disk, erasing data, sync marks, and so on. The effect is as though a small magnet had run over the diskette; much of the data will be destroyed. Don't, therefore, use these tabs with CBM disks.

(ii) Duplicate disks. There is a potential problem with duplicate disks; since their i.d. is the same, they're treated as identical by the machine, so that the wrong disk of a pair, one of which has been updated, used by accident, while DOS stores the other's BAM, will cause a wrong BAM to be written, and cause sectors to be put in the wrong places. This is not likely to be a serious problem if all disks except backups have different i.d.s, or if disks are not taken out and replaced by others with identical i.d. It is also not a problem with the newer drives which automatically initialise all disks.

(iii) Unclosed files. COLLECT or PRINT#15,"V..." is the command which erases unclosed files. Again, this is not likely to be a problem, since it is rather easy to close files. Program development is a likely time for this bug to strike, since a syntax error aborts files, which may not be closed properly afterwards.

(iv) COPY. This is a useful command, which unfortunately has a bug when used to copy an entire disk, e.g. in COPY D0 to D1 or PRINT#15, "C1=0". If the disks have different i.d.s, only 8 files can be copied at one time. Also, relative files often copy

wrongly; a utility such as COPY/ALL is preferable, or BACKUP of course can simply create a duplicate disk without bothering with COPY. Note the syntax of COPY and of the equivalent PRINT#15 command. The reversal in order of the drive numbers can cause a file to be wrongly replaced by its earlier version. BACKUP and PRINT#15,"D." have the same reversal. This will not be a problem unless a user is careless; the best precaution is to use a program with explicit instructions to help perform these functions.

(v) Relative and sequential files. Several files can be opened to a single relative file. This is probably best avoided; there are several reports of buffers being wrongly written in these circumstances. Short relative files, with total record length less than 254, may have incorrect side sectors allocated. Make them several sectors long at least. DOS 2.5 can fill only about a third of the disk with a single relative file; several may have to be used where one would theoretically be best. However, it may be unsafe to write to relative files when several relative files are open; 'Before the write is complete a buffer may get overwritten', is one theory. The best way of using relative files seems to be not to have several parallel files, holding between them data on items 1 to N, but to have several files covering the range 1 to N/3, N/3+1 to 2N/3, and so on; in this way, the minimum number of files need be open. There are other rumours; to quote Jim Butterfield, 'There are a lot of rumors flying about ... you hear a lot of stories you can't believe'. One of these is that sequential and relative files ought not to be mixed. This may be a legacy from DOS 1, when relative files were a different species altogether, and seems to be without foundation.

(vi) A disk full error causes files to be left unclosed, since there is insufficient space on disk to store the final sector. Use COLLECT if this happens. (It should in any case never happen with a properly organised working system).

Software problems (i) Timing. Whenever large amounts of data are going to be stored on file, or when complex processing is to be carried out, the usual guesstimate style of inferring processing time from benchmarks based on small files should be replaced by an accurate trial. If this isn't done, it may be discovered late in the day that a sort takes 36 hours, or a sequential read takes 24 hours to get through a file. Test data can be generated by a program, and used to check the efficiency of a system in action. The data may be nonsensical, but its function is to be processed, not to simulate actual data. Sometimes program redesign can make a huge difference to overall processing time. I've seen a set of programs whose author didn't know about arrays; each of a hundred or so categories was extracted from a single file, in about six minutes per category. As a result, a report which could have been produced in about 10 minutes took about 10 hours.

(ii) Inaccurate storage. Despite the system of internal checks used by diskettes, there is a small chance that data may be stored wrongly; the most likely defect is a dropped bit, where a byte which should be #D0 loads as #90, for example, the correct bit pattern of 1101 0000 loading as 1001 0000. This is only likely to happen to old disks which have been untouched for some time, and is a very uncommon fault. But it is not impossible; I've found examples in old copies of DOS Support programs. The easiest way to check for this, if it is felt to be necessary, is to use a hashtotal program to conflate all the bytes of the program into a single-figure value, and check that the value agrees with the figure computed. This is a simple thing to do: see Chapter 14 for an example. Again, it is rarely done, in spite of the extra assurance it provides that a program has correctly loaded. This seems a pity, since long machine code programs are difficult to validate in any other way, and a few incorrect bytes can cause baffling failures and errors.

Diskette care Most diskette envelopes have a set of symbols printed on the back, providing pictorial warnings against maltreatment. Smoking and dust can damage the surface, and slowly degrade the performance. Magnetism is an obvious hazard: electric motors, TV sets, VDUs, transformers, telephones, demagnetizers can all help to erase data from disks. Diskette boxes can be lined with metal foil to provide a Faraday shield against magnetism; this looks efficient and may actually be useful. Some people believe that underground trains and X-ray scanners can erase disks unless they are metal-wrapped. Diskettes can be damaged ('glitched') by small pulses of magnetism if drives are turned on or off with disks in place. The probability of harm is reduced if the drive doors are open. 4040 and 8050 drives, unlike their predecessors, seem safe in this respect. Note that the earlier drives don't have a centering mechanism for their disks, so the recommended procedure is to keep the drive door open until the disk has begun to turn.

Disk drive care and maintenance   For obvious reasons, disk drives should ideally not be subjected to smoke or dust. It is usually good policy not to physically move them much. Individual manufacturer's units vary in their resilience; some drift out of adjustment quite easily, so that disks become more likely to be non-interchangeable between drives; others are more secure and robust. A sudden movement in one direction may be more harmful than in another direction. Facts on this topic are hard to come by, and tend towards the anecdotal ("X used his system for a year without any trouble"). Most drives sold in the U.K. to date have been American. The long trans-American and transAtlantic journeys aren't always good for these machines. Often they are not resold without being unpacked and checked, a fact which annoys some buyers. Routine maintenance is usually concerned with the heads: these can be cleaned with cotton wool and a solvent such as isopropyl alcohol, or with a head-cleaning diskette, which is a diskette case housing a diskette-shaped thin absorbent cloth.

    Disk drives are serviced by cleaning and lubricating the appropriate parts, perhaps replacing components if a design improvement has been announced, and realigning the heads. This is done by checking the track zero end stop, and using a specially recorded master diskette to check the output from the head. Disks of this sort are recorded eccentrically (in the technical sense!) so a suitable high-frequency oscilloscope records a pattern (called a 'Cat's eye') of two adjacent areas on the screen of the oscilloscope; when these areas are equal the head is in the middle of its track. The spindle motor is checked too. The price-range of disk units is such that servicing them is a tricky business, and there is considerable temptation for dealers not to get involved with this sort of work. For example, a sales director of a British chain said he was reluctant to get involved with hard disk units, because it would take another eight weeks to train the engineers. It is therefore worth making sure that you have access to a technically competent dealer, or to people whose business it is to design and use electronic hardware. The work is sometimes farmed out to other organisations; the results of this are unpredictable, and from a consumer viewpoint it's easier to deal with one company than with a maintenance organisation which is completely different from the hardware supplier.

    Sometimes because of a software quirk a unit may appear to be defective when in fact there is no serious fault. One example with Commodore's series of drives is connected with the use of two processors in those drives: the internal one can be lost, inaccessible to the IEEE. Initialization will bring it back to life.

    There are several points worth mentioning about the 8050 series CBM disk drives and its descendants. The double-sided version, called the 8250, runs a DOS version (2.7) apparently different from any 8050 DOS; Commodore may have problems numbering its subsequent ROM issues for the 8050 because of this. The 8061 and its double-sided equivalent the 8062 (8 inch IBM compatible diskettes), in spite of appearances in brochures, may be pre-empted by the 8250, and perhaps not appear. On shipping problems of these units: '... the 8050 drives are intended to float freely within their mounting case. However, in shipping, the outer case flexes against the too-tight cutouts, thus bending the drives. This in turn misaligns the heads, which are very critical on this octal density drive. Moral of story: dealers, learn how to realign Micropolis drives...'. This is Jim Strasma, quoting Bill Seiler of Commodore in Canada. New units use Tandon disk drives, replacing Micropolis, which themselves succeeded Shugart. These are presumed to be more reliable, but solid information about this is hard to get. It may be worthwhile specifically ordering the most up-to-date units, if you can find out what they are. This is a point on which user groups may be more helpful, or at least have alternative views, when compared with dealers who may have to shift relatively old stock. Finally, a smaller single disk unit is to be made available for the VIC; probably to be called the 4031 the idea is to provide cheap disk backup for home users, like the single Apple drives. At the time of writing the specification remains vague; these drives may be compatible with PET/CBM, or they may not.

CHAPTER 7: ALPHABETIC REFERENCE TO DISK BASIC COMMANDS

7.1 Notes on BASIC disk commands. BASIC 4 has fifteen keywords which earlier BASICs lack and which are all concerned with the disk operating system of Commodore's disk drive units. They are not intended for use with other manufacturer's equipment. The keywords are CONCAT, DOPEN, DCLOSE, RECORD, HEADER, COLLECT, BACKUP, COPY, APP-END, DSAVE, DLOAD, CATALOG, RENAME, SCRATCH, and DIRECTORY in ascending order of token. There is also a disk status indicator, resembling ST, which takes two forms, DS$ and DS. Earlier BASICs cannot list these tokens without a special program; in fact other keywords from FOR to REM and including ?SYNTAX ERROR will appear in their place. At first sight these commands look like radical additions to BASIC: they suggest that now we can read and write to disk in a way that was impossible before. In fact, this is not the case. The important thing to grasp about CBM disk units is that most of the processing is performed within the disk unit. All that BASIC does is send 'command strings' and data to the disk units, and receive data back again. The disk units are 'intelligent' and carry out their functions without the CBM's processor. Many other microcomputers store their DOS in RAM, or in ROM, like the cassette system of the CBM range, where it can be disassembled and examined. CBM's disk system resembles, and can be treated as, a 'black box'. What is all-important is the set of ROMs in the unit. The early disk operating systems, DOS 1 to DOS 1.2, have fewer features than DOS 2 to DOS 2.7, notably the absence of relative files. This difference is independent of the version of BASIC which uses the disks. So a DOS 2.1 disk is controllable even by the earliest PET but without the extra commands listed above. For this reason I have included equivalent IEEE commands using strings containing controlling characters along with the simpler BASIC 4 commands. Most DOS systems try to abbreviate as far as possible; often clashes between commands with the same initials have to be avoided, with strange circum-locutions like 'X' for 'Exit'. CBM disks have not been free of this difficulty. The table shows which BASIC 4 commands correspond to which command string characters and the names previously assigned to them. See also Chapter 15, BASIC 4 ROM from $D839.

7.2 Notes on BASIC 4 disk command syntax. BASIC 4 has an elaborate syntax checking technique (see Chapter 15, $DC68 in BASIC 4) allowing all the parameters including the file number to be arranged in any order. DOPEN "FILE",W,#3 and DOPEN#3, W,"FILE" are effectively identical. This has forced out some constructions which would now be am-biguous. R shows that a sequential file is to be read; it cannot be also used to indicate a relative file. So only the length of the record (e.g. L100) is a parameter when a new relative file is opened. On the other hand, some extra ambiguities have been introduced. Device number 9 may be specified by ON U9 or ,U9. Strings and numerals may be en-closed in brackets, but need not be if they begin with " or with 0-9 respectively. All string expressions, and numeric expressions not beginning with a numeral, must be in brackets, or ?SYNTAX ERROR appears. The following four examples of a DOPEN state-ment are equivalent, provided that drive 0 of device #8 holds the destination diskette:

```
DOPEN#1,"FILE OF NAMES5",U8,D0,W
DOPEN ("FILE OF NAMES" + N$),W,#1 ON U8  :REM ASSUMING N$="5"
DOPEN # (X),(FN$ + CHR$(N)),W           :REM IF X IS 1, FN$ IS "FILE
DOPEN#1,"FILE OF NAMES",W                OF NAMES", AND N=53
```

In this section I have assumed for consistency that the command/error channel with sec-ondary address 15 has been opened with OPEN 15,8,15. I have used upper-case to distinguish BASIC from ordinary text, and in the BASIC 4 examples used some lower-case commands (the 8032 - not the 4000 series though! - powers on into this mode).

BASIC<4	BASIC 4	BASIC<4	BASIC 4
---	APPEND	LOAD	DLOAD
D[UPLICATE]	BACKUP	OPEN	DOPEN
---	CATALOG	---	DS, DS$
V[ALIDATE]	COLLECT	SAVE	DSAVE
C[ONCATENATE]	CONCAT	N[EW]	HEADER
C[OPY]	COPY	I[NITIALISE]	---
CLOSE	DCLOSE	---	RECORD
---	DIRECTORY	R[ENAME]	RENAME
		S[CRATCH]	SCRATCH

APPEND

BASIC 4 disk file command

PURPOSE: APPEND reopens a closed sequential file, setting pointers to the end of the file and preparing to write to disk. In this way a sequential file can easily be extended to store more data.

NOTE: This BASIC 4 commond has no direct connection with the techniques discussed alsewhere to join BASIC progroms end-to-end.

Syntax: DOS 1+: APPEND is not directly available; concatenation of the old file to the new file must be used instead. See CONCAT.
DOS 2+: The DOS interface is 'Drive number: file name,A'.

APPEND is followed by these parameters in any order:
(i)   # then expression for the logical file number.
(ii)   String or string expression in brackets. This is the name of a sequential file.
(iii) Optional drive number. ,D followed by an expression for 0 or 1.
(iv) Optional device number. ON U or ,U with an expression for 4-31.
Typically this looks like:
APPEND# arith. expr. ,"name" [,D arith. exp.] [,U arith. exp.]

Spaces - except within APPEND itself or the string, are skipped by BASIC and have no substantial effect.*

Examples: BASIC 4. The first example creates a sequential file called "file of names" holding one hundred names, which are assumed to be present in the array N$(). Some time later - perhaps almost immediately, perhaps months after the file had been written - more names need to be added to the file. By definition, there is no alternative, with a sequential file, to writing these onto the end. The method is to open the file with APPEND, and write to the file, as program line 110 does here.When line 120 closes the file, the new data has been appended like this:

START OF FILE:	RECORD 1	RECORD 2	....	RECORD 100	NEW RECORD 1	...NEW RECORD N

```
10 dopen#1 "file of names"  ,w   :rem  open a sequential file for writing
20 for j=1 to 100: print#1, n$(j): next: rem write 100 strings
30 dclose #1                     :rem  close the file
100 append #2,"file of names"    :rem s,w, are both implicit in this
110 for j=1 to n:  print#2,n$(j): next : rem write n more strings
120 dclose #2                    :rem  close the file
```

BASIC<4. The example below is exactly equivalent, but omits BASIC 4's special commands. It may of course be run on a BASIC 4 machine, if it is required to ensure that a program is compatible with any BASIC.

```
10 OPEN 1,8,2,"0:FILE OF NAMES,S,W": REM SEC. ADDRESS IS UNIMPORTANT
20 FOR J=1 TO 100: PRINT#1,N$(J) CHR$(13);: NEXT
30 CLOSE 1
100 OPEN 2,8,2,"0:FILE OF NAMES,A"
110 FOR J=1 TO N: PRINT#2,N$(J) CHR$(13);: NEXT
120 CLOSE 2
```

Notes: [1] APPEND implies both a sequential file, and 'write' mode. The reason is that appending is not needed with relative files, since any record can be selected by its number. And since the pointers are set to the end of file, reading from this point would not achieve anything. Sequential files, because of their irregular record length, must usually be read from the beginning.

[2] Some CBM disk manuals omit this command.

Abbreviated entry: aP

Token: $D4 (212)

Operation: See Chapter 15 under $D977 in BASIC 4.

ROM entry: The kernel jump address is $FFAB; this jumps to $D977.

---

*This is true of all BASIC 4 disk commands, and I shall not explicitly state this fact for each of them

BACKUP

BASIC 4 disk system command

PURPOSE: Creates an exactly identical disk for security purposes and for the creation of multiple copies of programs.

Syntax: DOS 1+ and DOS 2+ have the same DOS interface, which is:
'D destination drive number: source drive number'. However, their processing is not identical - see note [1].

BACKUP is followed by one or two parameters:
(i)   D then expression for 0 or 1 TO D then expression for 1 or 0.
(ii)   Optional device number. ON U or ,U with an expression for 4-31.
Typically, this appears like: BACKUP D0 TO D1.

Examples: BASIC 4. The example duplicates a diskette in drive 0 (the right-hand drive) onto a disk in drive 1. The duplicate is formatted (and could be a new, unused disk) and copied block for block. For this reason a disk to be duplicated *must* be the same type as the drive which does the duplication, since the number of tracks and sectors must match. (Cp. COPY).

```
10 ? "Place ORIGINAL disk in Drive 0  (right-hand drive)"
20 ? "           COPY disk in Drive 1  (left-hand drive)"
30 ?:?"Press spacebar to duplicate"
40 get x$: if x$<>" " goto 40
50 backup d0 to d1
60 goto 10
```

BASIC<4. The following example performs the same function, without BASIC 4's special command. Note that the order of disks in duplication is apparently reversed! In fact the DOS interface interprets the number after D as the destination drive, and the number after the ':' as the source, so BASIC 4 takes the two disk drive parameters and sends them on the IEEE bus in this different order. If the drive numbers are entered wrongly and the command goes to completion, the data will be completely irretrievable; this is why it is desirable to use a program - like the above - with explicit instructions.
     50 PRINT#15,"D1=0":   REM READ THIS AS 'DRIVE 1 BECOMES DRIVE 0'*

D in this command string is the initial of 'Duplicate'. Perhaps this was felt to be too long a command for BASIC; hence 'Backup'. Note that only the initial is relevant; 50 PRINT#15,"DAISY1=0" would do as well.

Notes: [1] Older versions are slower (about 6 minutes compared to about 1-3 minutes depending on the type of unit). There is also a difference in the underlying philosophy: the earlier duplicate command took no account of errors, whereas the later BACKUP command aborts on finding errors. In fact some CBM disks are 'copy protected' by misrecording a few sectors. BACKUP starts at the outside of the disk and works inward.

[2] Remember that the disk formats must match. There are (at the time of writing!) three of these: 2040 and 3040, 4040, and 8050. Each type of disk must be treated separately when duplicating disks - see COPY.

Abbreviated entry: bA

Token: $D2 (210)

Operation: Chapter 15 explains BASIC 4's operation; the disk unit does most of the work

ROM entry: The kernel jump address is $FFA5; this jumps to $DA7E.

---

*Throughout this chapter I have assumed that the command/ error channel has been opened with logical file number 15, thus: OPEN 15,8,15.

CATALOG & DIRECTORY
BASIC 4 disk system command

PURPOSE: Displays the contents of a CBM disk on the screen. Data written directly to sectors will not show up, since it bypasses the file creation and directory entry routines which are otherwise used. Other types of disk, for example those formatted on other machines or not readable because of incompatibility (e.g. 8050 on 4040 drives), will, not surprisingly, not have their directories read in this way.

Syntax: DOS 1+ and DOS 2+ each store their directories in similar ways, as a BASIC program, complete with link addresses and line numbers, and zero termination bytes to signal that the end has been reached. The difference is that the 'line-numbers' represent sectors occupied by the program or file data, and need not be sequential. DIRECTORY and CATALOG (these have identical effects) are BASIC 4 commands which do not rely on a DOS interface, but instead display the directory by reading bytes, formatting them like BASIC with a number and text, and printing the result directly onto the screen. RAM is unaffected, except screen RAM, which becomes the storage device. Once DIRECTORY has scrolled off the screen it can be recovered only by another DIRECTORY. The syntax is

DIRECTORY [D arith. expr. for 0 or 1] [, or ON U arith.expn. for device no.]

Examples: BASIC 4. Unlike the DOS universal wedge program, this may be called in program mode; this is useful when producing hardcopy listings of the contents of a lot of disks. Both CATALOG and DIRECTORY or their short forms, e.g. cA and diR, are accepted. Note that this form lacks some of the features which are available by loading the catalog as a program, notably the production of subsets of the catalog. Since BASIC 4 is designed for use with larger amounts of storage than before, this seems a little strange.

```
open 4,4: cmd 4,;: cA dØ : rem print catalog of diskette in drive Ø
```

```
10 input "drive number";d: if d<>Ø and d<>1 goto 10
20 print "[clr] Press spacebar to pause"
30 directory d(d)
```

BASIC<4. The directory is loaded as a BASIC program; this has the drawback of overwriting any BASIC in memory, and the advantage of retaining it in memory. The disk operating system is able to identify several useful variations on the simple directory:

```
LOAD "$Ø",8          :REM LOADS DIRECTORY INTO MEMORY; NOW LIST IT.
LOAD "$1",8          :REM SAME, EXCEPT THAT DIRECTORY IS FOR DRIVE 1.
LOAD "$Ø:MY*",8      :REM LISTS ALL PROGRAMS, FILES, BEGINNING 'MY'.
LOAD "$Ø:??M/C*",8   :REM LISTS ALL PROGRAMS, FILES WITH 'M/C' IN
                           3RD, 4TH & 5TH POSITIONS (ON DRIVE #Ø).
LOAD "$1:FILE*=P",8  :REM PROGRAMS ONLY, STARTING 'FILE', ON DRIVE 1
LOAD "$Ø:*=S",8      :REM DIRECTORY OF SEQUENTIAL FILES ON DRIVE Ø.
```

DOS SUPPORT (UNIVERSAL WEDGE). This program is usable with both BASIC 4 and BASIC<4. Its method of displaying the directory is identical with that of BASIC 4, at least in outline: some differences may well show up, since several versions of DOS support exist. These differences largely affect the I/O processing but not much else, so that one version will print a directory to a printer after CMD, another won't. However, the commands sent to the disk are more flexible than BASIC 4 will permit, because any characters can be loaded into the buffer. The following constructions are therefore all acceptable:

```
@$Ø   or  >$Ø          :REM DIRECTORY OF DRIVE Ø TO SCREEN.
@$    or  >$            :REM DIRECTORIES OF BOTH DRIVES ARE DISPLAYED -
                             BE SURE EACH HAS A DISKETTE!
@$1:ASSEM*             :REM DIRECTORY OF DRIVE 1, BUT ONLY OF PROGRAMS/
       or  >$1:ASSEM*        FILES WHOSE NAMES BEGIN 'ASSEM'.

@$Ø:*=P or >$Ø:*=P     :REM DIRECTORY OF PROGRAMS ONLY FROM DRIVE Ø.
```

Notes: [1] <u>Contents of a directory</u>. The specimen
directory, from an 8050 drive unit, shows
the appearance of a typical diskette directory
on which programs and files have been stored.
The top line prints the drive number, and, in
reverse, the diskette's name and i.d. charac-
ters. The version of DOS is indicated, not as
2.1 or whatever, but 2A or 2C or some similar
code. 2C is DOS 2.5, the 8050 version of DOS
2, which allocates space for 2052 sectors (or
'blocks') on the diskette. Each filename, in
quotes, is displayed with the number of sectors
which it occupies; the first three programs,
for example, all occupy about 35 sectors, and
since CBM sectors are 256 bytes long, these
programs are about 35/4 or 9K of BASIC.

NOTE: files which were not CLOSEd, or which
were SCRATCHED when open, show various
warning signs, including file-type DEL or an
unexpected asterisk. At this point, if the data
is important, the disk should be overwritten
by a backup; if one has not been taken, in-
dividual files may be COPYed to another disk,
with luck.

0	▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮	
35	"BEEPROG432S"	PRG
35	"BEEPROG432P"	PRG
36	"BEEPROG832S"	PRG
4	"TELESOFTWARE"	PRG
19	"BEEPROGCASSETTE"	PRG
4	"INFO"	SEQ
4	"P1"	SEQ
4	"F1"	SEQ
4	"ALCC1"	SEQ
4	"CR1"	SEQ
4	"H1"	SEQ
4	"E1"	SEQ
4	"N1"	SEQ
4	"N2"	SEQ
4	"V1"	SEQ
4	"GBL"	SEQ
4	"NL1"	SEQ

1875 BLOCKS FREE.

[2] <u>Initialisation</u>. DOS 2.1 and 2.5 automatically 'initialise' (q.v.) their diskettes,
but earlier versions don't. This can be done by:

     OPEN 15,8,15: PRINT#15,"I∅" :REM "I1" FOR DRIVE 1

or, when DOS support is loaded, by:

     @I∅   or   >I1

Whenever a new disk is put into a drive it should be initialised, since otherwise
the operating system may write data according to a wrong block availability map,
overwriting data. If a program includes initialisation, or if it is automatic, then
it needn't be repeated. So directories always require initialisation, unless the
disk has been initialised or this is an automatic function. (I hope this is clear!).
It is possible to disable (i.e. switch off) the auto initialisation; this is a possible
though unlikely source of trouble.

[3] <u>Other versions</u>. The directory is an exceptionally accessible piece of code,
and it is instructive to disassemble DOS support or BASIC 4 if you wish to use
machine code with disks. An example: with DOS Support loaded, use the monitor
to examine the high end of RAM, e.g. 7F80-7FFF in a 32K machine. About 9 lines
up from the bottom is an ∅D, a carriage return character. Replace this by 92, the
hexadecimal equivalent of [RVSOFF]. Now the directory will print across the page
instead of in columnar form. This BASIC program mimics some of the DOS Support
and produces program and file names in columns:

```
10 OPEN 1,8,0,"$0"              :REM    DIRECTORY OF DRIVE 0
20 GET£1, X$: GET£1, X$         :REM    REJECT TRACK & SECTOR BYTES
30 IF X = 4 THEN X = 0: PRINT   :REM    PRINTS 4 COLUMNS, FOR 8032;
40 PRINT TAB(20*X);             :REM       (X=2 PRINTS 2 COLUMNS ONLY).
50 GET£1,X$: GET£1,X$: GET£1,X$: GET£1,X$: IF ST OR DS THEN CLOSE 1: END
51 :                            :REM    REJECT 4 BYTES INC. NO. OF SECTORS
60 GET£1,X$: IF X$ = "" THEN X = X + 1: GOTO 30
70 IF X$ = CHR$(34) THEN Q = NOT Q:      GOTO 60
80 IF Q THEN PRINT X$;          :REM       PRINT ONLY ...
90 GOTO 60                      :REM       ... STUFF WITHIN QUOTES
```

Abbreviated entry: cA and diR   Token: $D7 (215) and $DA (218)

Operation: See Chapter 15 on BASIC 4 and Chapter 14 on DOS Support.

ROM entry points: Both keywords have the same jump address, $FFB4 which jumps to
     $D873.

COLLECT

BASIC 4 disk system command

PURPOSE: COLLECT (or Validate) rewrites a diskette's 'Block Allocation Map' to exclude sectors in files which have not been closed correctly. The first byte of a file's directory entry indicates to the disk operating system that a file was open but was not closed. Files are checked, by reading consecutive sectors, for a correct termination byte. If this is not present, the file will be assumed to continue in another part of the disk, and sooner or later will become entangled with some other file. The object of COLLECT is to delete such files and ensure that a diskette contains only sound files.*

NOTE: This command is suitable for data which is held in the form of linked sectors. This includes BASIC programs, sequential files, and (DOS 2+ only) relative files. But data written by the user directly in sectors is not treated in the same way, so that the sectors are de-allocated from the block availability map. Subsequent write-to-disk operations by the file-handling system will erase these sectors as soon as the remaining disk-space before them is full. For this reason, disks with 'user files' are best kept distinct from disks with DOS files.

Syntax: DOS 1+ and DOS 2+ have the same DOS interface: 'V drive number', which in DOS 1 meant 'Validate', and was often confused with VERIFY, which is a program verification command, not a file verification function. There are differences in processing between DOS ROMs, notably when dealing with relative files, which DOS 1 doesn't recognise.

COLLECT is followed by D and an expression for 0 or 1.
(COLLECT alone defaults to drive 0).

Examples: BASIC 4. The example validates or collects - whichever you prefer - the files on drive 0:

```
collect d0
```

BASIC<4. This example does the same, without BASIC 4's keyword:

```
PRINT#15,"V0"
```

Notes: [1] COLLECT exists because of the possibility of corruption of data by files which are incorrectly stored. This type of difficulty is inevitable with any disk system which allows sectors to be written anywhere on a disk. The problem may be a long time in the making: an error may have been working on your disk for months, to quote Jim Butterfield. CBM machines are unusual in not having the validation as a normal part of the operating system. Similarly, they don't seem to have a method of indicating 'bad' sectors on disks. COLLECT therefore probably always ought to be used when programs and files are stored on disk; but data written to sectors by B-W and similar commands cannot be COLLECTed, since, instead, the block allocation map will reassign their sectors as unused, and subsequent file-writing will sooner or later occupy these sectors.

[2] See SCRATCH: this command too has special properties when corrupted data is involved. See also COPY with reference to relative files.

[3] If COLLECT signals an error - usually failure to read the disk - the block map on the disk isn't changed - yet. But the block map in RAM will have been modified to some extent, probably, so that it's risky to proceed without initializing the diskette again, reloading the old block map.

Abbreviated entry: coL

Token: $D1 (209)

Operation: See Chapter 15 for BASIC details.

ROM entry points: The kernel jump address is $FFA2; this jumps to $DA65.

---

*This command-like many in BASIC-operates by switching pointers and flags, leaving data largely intact. The data in dud files still exists after COLLECT, but the file name is removed from the directory, and its sectors are no longer officially in existence as recorded by the block availability map, and hence are liable to overwriting.

CONCAT

BASIC 4 disk file command

PURPOSE: CONCAT concatenates sequential files, so that the resulting single file contains all the data from the original files in sequence.

Syntax: DOS 1+ and DOS 2+ have the same DOS interface, which is:
'C Destination drive number:destination file name = drive:first file , drive:2nd file'. After this command, the destination drive holds a file of the destination name specified, which consists of 'first file' with '2nd file' appended to it. Note that CONCAT does not allow a new name to be specified; instead the name is taken by default from the second file name parameter. So a construction like: CONCAT old file TO newer file GIVING up-to-date file is *not* available with CONCAT, but *is* available when using the alternative form with 'C'.

The syntax is CONCAT [D with expr. for 0 or 1,] file name string or variable in brackets TO [D with expr. for 0 or 1,] file name [ON U device number expr.]

In practice, this looks like: CONCAT D0,"NEW STUFF" TO D1,"TOTAL FILE", which appends the data called "NEW STUFF", from drive 0, onto "TOTAL FILE" on drive 1. Files which are not sequential data files give an error in DS$.

Examples: BASIC 4. The first example concatenates two files from the same drive; the second concatenates files from different drives. Note that there is no provision for concatenation between drive units.

```
1000 concat d0,("new data" + str$(n)) to d0,"all data" on u8
1500 concat d0,"data1" to d1,"data"
```

The first example could be part of a loop which appends several files with names "new data 1", "new data 2", and so forth, onto the file "all data". Note that, because the drives are identical, the first file will disappear from the directory, The second example performs a copy before concatenating, so that the original files both still exist separately.

BASIC<4. To make the operation of this command clear, I've included a listing of a demonstration program which concatenates a file on drive 1 called "FIRST FILE", and a file on drive 0 called "SECOND FILE". The concatenated file is "RESULT", and is on drive 1. Lines 10-150 write the two sequential files which are to be concatenated. Lines 300-330 perform the concatenation, using syntax identical to that of the DOS interface, and which avoids the keyword CONCAT. Finally, lines 400-440 read the file called "RESULT" and print its contents, to show that the required concatenation has in fact taken place correctly. The syntax of the crucial command is

```
310 PRINT#15,"C1:RESULT=1:FIRST FILE,0:SECOND FILE" : REM FOLLOWING IS OK:
310 PRINT#15,"CONCAT1:RESULT=1:FIRST FILE,0:SECOND FILE"
```

Notes: [1] This command operates by switching the pointers at the end of the first file to point to the start of the second. Then the directory entry of the second is erased, so the file simply reads from one file to the next: the position of the sectors on disk will reflect this history. Relative files cannot be concatenated, with the present DOS, partly, presumably, because of the greater difficulty of programming this as compared with sequential files. CONCAT is closely related to COPY; when different drives are involved in CONCAT, the first stage is to execute COPY, so that the file to be appended is present on the same disk as the major file.

[2] BASIC<4 can execute what is effectively APPEND using concatenation; all that is needed is the data which would have been written to the file opened by APPEND on its own file, which after CLOSE can be concatenated onto the main file.

Abbreviated entry: conC

Token: $CC (204)

Operation: See Chapter 15 for BASIC details.

ROM entry point: The kernel jump address is $FF93; this jumps to DAC7.

DEMONSTRATION OF THE USE OF 'C' TO CONCATENATE

```
10 OPEN 5,8,5,"1:FIRST FILE,SEQ,W"
20 PRINT£5,"FIRST FILE.. 20 RECORDS"
30 FOR J= 1 TO 20
40 PRINT£5,"RECORD NO."J
50 NEXT
60 CLOSE 5
100 OPEN 5,8,5,"0:SECOND FILE,SEQ,W"
110 PRINT£5,"SECOND FILE.. 10 RECORDS"
120 FOR J= 1 TO 10
130 PRINT£5,"RECORD NO."J
140 NEXT
150 CLOSE 5
160 END
300 OPEN15,8,15
310 PRINT£15,"C1:RESULT=1:FIRST FILE,0:SECOND FILE"
320 CLOSE 15
330 END
400 OPEN 5,8,5,"1:RESULT,SEQ,R"
410 INPUT£5,X$:
420 PRINT X$;
426 IF ST<>0 THEN CLOSE 5: END
430 NEXTJ
440 GOTO 410
```

'RUN' SETS UP TWO SEQUENTIAL FILES, ONE ON EACH DISKETTE IN THIS CASE;
'RUN 300' CONCATENATES THE TWO FILES INTO A NEW FILE CALLED 'RESULTS';
AND 'RUN 400' DEMONSTRATES THE SUCCESSFUL CONCATENATION.

FIRST FILE.. 20 RECORDSRECORD NO. 1 RECORD NO. 2 RECORD NO. 3 RECORD NO. 4 RECOR
D NO. 5 RECORD NO. 6 RECORD NO. 7 RECORD NO. 8 RECORD NO. 9 RECORD NO. 10 RECORD
NO. 11 RECORD NO. 12 RECORD NO. 13 RECORD NO. 14 RECORD NO. 15 RECORD NO. 16 RE
CORD NO. 17 RECORD NO. 18 RECORD NO. 19 RECORD NO. 20 SECOND FILE.. 10 RECORDSRE
CORD NO. 1 RECORD NO. 2 RECORD NO. 3 RECORD NO. 4 RECORD NO. 5 RECORD NO. 6 RECO
RD NO. 7 RECORD NO. 8 RECORD NO. 9 RECORD NO. 10

COPY

BASIC 4 disk file command

PURPOSE: COPY permits the selective copying of any files from one diskette to
another, *except* relative files (at the time of writing). It also permits an entire
diskette to be copied onto a second diskette, without erasing the current con-
tents of either disk *except* with DOS 1+, in which case each file must be copied
individually. Note the distinction between COPY and BACKUP (or DUPLICATE);
COPY reads the file and writes it back as though it were being written from a
program; it is added to the current diskette contents. For this reason COPY can
convert the format of any readable disk into its write format; 3040 diskettes may
be copied by a 4040 drive. BACKUP is less 'intelligent' and produces an exact
replica, provided the format is consistent. This of course is essential if tracks
and sectors have been written by the user in ways which can't be read as ord-
inary files.

Syntax: DOS 1+ and DOS 2+ appear to have identical facilities as regards both COPY
and CONCAT, except that DOS 2+ can copy an entire disk when no file names
are specified in the COPY command. The DOS interface used by BASIC 4 is:
'C Destination drive:destination name = Source drive:source filename'. This is a
subset of the full interface, which causes concatenation rather than copying.

The syntax is: COPY   [D expr. for Ø or 1,][name] TO [D expr. for Ø or 1,]
[name], where both names may be omitted, or both names present. The destin-
ation file name is checked, and if found to exist, ?file exists error (63) is set
in DS$. So copies made to the same disk must use a different name.

Examples: BASIC 4. The first example copies an entire disk onto another. Since the
destination disk is not cleared in any way, the copy may abort with ?disk full.
Running HEADER first is therefore common - see note [1]. The second and third
examples copy a file to the same disk, and the other disk, respectively. Note
that the names cannot be the same in the second example.

```
100 copy dØ to d1
200 copy dØ,"text" to dØ,"text1"
300 copy dØ,"text" ti d1,"text"
```

BASIC 4. These are the equivalents without the keyword 'COPY'. Note that the
first example only works with DOS 2+, which was specially extended to include
it:
```
100 PRINT#15,"COPY1=0": REM NOTE THE REVERSAL OF SOURCE AND DESTINATION!!
200 PRINT#15,"C1:TEXT=0:TEXT"
300 PRINT#15,"COPY0:TEXT1=0:TEXT"
```

Notes: [1] When converting from DOS 1+ to DOS 2+, either through ROM upgrade or
change of hardware, COPY may be used to reformat the disks by reading the old
files and writing them to new disks. However, 8050 disks have more tracks, and
can't be read by smaller disk units, and vice versa. To copy this type of data
needs two disk drives connected to the same machine and a copy program: two
are available through user groups and clubs: COPY ALL by Jim Butterfield and
another version, COPY/ALL which copies relative files too. COPY with relative
files gets *most* of the copy correct, but not all. There is no syntax error.

[2] Watch for the reversal of order between COPY and PRINT#15,"C ...". This
occurs in BACKUP and PRINT#15,"D ..." too. It is less serious here; a mistake
simply won't find the source file, unless it erroneously exists on the destination
file, so ?FILE NOT FOUND ERROR is about the worst that can happen. Of course
the destination disk may be copied in its entirety onto the source disk too.

Abbreviated entry: coP   Token: $D3 (211)

Operation: See Chapter 15 for BASIC details.

ROM entry point: The kernel jump address is $FFA8; this jumps to $DAA7.

DCLOSE
BASIC 4 disk file command

PURPOSE: DCLOSE performs exactly the same function as CLOSE. It has an optional form, however, which closes all open files. The file closed need not be a disk file; any IEEE file, opened to device number 4 to 31, may be closed by DCLOSE.

Syntax: DOS 1+ and DOS 2+ can both handle this command, which is identical, except for syntax, to CLOSE.

The syntax is DCLOSE [#arith. exp.] [, or ON U arith.exp.]
where the first parameter is a logical file number (1-255) and the second the device number.
DCLOSE closes all files;
DCLOSE #1 closes logical file #1;
DCLOSE #1 ON U 9 closes logical file #1 on unit 9;
DCLOSE U8 closes all files on unit #8.

Examples: BASIC 4. The first example shows individual files being opened and closed; the second shows the use of DCLOSE to close all files. The BASIC<4 equivalent is in the REM statement. There is little difference between them.

```
10 OPEN 4,4      :REM OPEN FILE 4 TO A PRINTER
20 DOPEN #8,"TEST",W: REM BASIC 4 IS OPEN#8,8,2,"0:TEST,S,W" (ASSUMING
                          DEVICE 8 AND DRIVE 0).
30 --- PROCESSING ---
1000 DCLOSE #4 ON U4: REM OR USE BASIC 4'S CLOSE 4
1010 DCLOSE #8      : REM SAME AS CLOSE 8

or 1000 DCLOSE : REM CLOSES ALL FILES ON DEVICE 8 ONLY. UNLESS OTHER FILES
                    HAVE BEEN OPENED, THIS IS EQUIVALENT TO CLOSE 8 HERE.
```

Notes: [1] DCLOSE, like CLOSE, has the effect of completing file processing. More details are given in CLOSE, but basically there are two things which may need to be done: one is to remove the file details from the file table, so that further attempts to read or write will need the file to be reopened. This happens to all closed files. However, files for writing (as opposed to files for reading) must also be finalised by writing the last buffer of data onto disk; otherwise there will be no record of the last items which were to have been written; and, what is worse, the chaining of blocks and sectors is left incomplete. This is the reason that correct file closure is stressed. This is not important with tape, as a rule but should be avoided on disks used for serious data storage. Very often, of course, this is not a problem: the programmer simply CLOSEs the files! If a syntax error of some kind occurs, however, it may be important to close write files from the keyboard; DCLOSE is useful for this purpose. But note that files are closed if a program is edited; in this case use the method in CLOSE, of poking the number-of-files-open location. A CBM manual says that (using our file numbering convention for the error channel) OPEN 15,8,15: CLOSE 15 will close all the files currently open. Both the directory entry, showing the length of the file, and the Block Allocation Map, as well as the data, are written on CLOSE or DCLOSE of a write file.

Abbreviated entry: dC

Token: $CE (206)

Operation: After checking the syntax, this routine calls CLOSE after the point at which parameters have been input. A single file number is checked to ensure it isn't zero; DCLOSE alone simply searches the table of device numbers for any file of the correct device - usually the default U8 - and closes each of these. (The routine is from DA1B - DA30).

ROM entry point: The kernel jump address is $FF99; this jumps to $DA07.

DLOAD
BASIC 4 disk file command

PURPOSE: Loads a BASIC program (or other contiguous RAM, e.g. machine-code) into RAM at the same locations from which it was saved. DLOAD unless otherwise stated assumes CBM disk unit #8. DLOAD is in fact virtually identical to LOAD; the only differences are the syntax and the fact that DLOAD validates the device number to ensure it is an IEEE device, so tape files for example won't DLOAD. See LOAD, therefore, for more about this subject.

Syntax: DOS 1+ and DOS 2+ are similar: in each case individual bytes are returned by the unit when it is made a talker, and these bytes are processed by BASIC which pokes them into memory from the start address which it also receives. DLOAD's syntax permits the following parameters, separated by commas, to be used in any order:
(i) String or string expression in brackets. This is the program name.
(ii) Optional D followed by expression for 0 or 1. This is the drive number; it defaults to drive 0, so drive 0 will be searched for the program if this parameter is omitted. It will go on to search drive 1 in this case if the file doesn't exist on drive 0.
(iii) Optional unit number, U followed by expression for device number 4-31.

Modes: The action of this command when called from within a program differs from that in direct mode; the difference is identical to that for LOAD, q.v. See note [2] on this page.

Examples: BASIC 4. All the following examples assume that the diskette is correctly initialised; this may or may not be an automatic function.

```
DLOAD "MY*"              :REM LOAD 1ST PROG. FROM DØ WHOSE NAME STARTS 'MY'
100 DLOAD (X$ + .COPY),U8:REM LOADS THE COMPUTED FILENAME
DLOAD "PROGRAM",D1       :REM LOADS 'PROGRAM' FROM DRIVE 1
```

BASIC<4. The following are exact equivalents. I've assumed device 8 throughout:

```
LOAD ":MY*",8
100 LOAD ":X$"+".COPY",8
LOAD "1:PROGRAM",8
```

In each case, if the file isn't found, the disk error channel will return error 62, ?FILE NOT FOUND ERROR.

DOS SUPPORT (UNIVERSAL WEDGE). The slash symbol (/) is equivalent to DLOAD; the up arrow (↑) to DLOAD *and* RUN. So, for example,

```
↑*              :REM LOADS & RUNS 1ST FILE ON DRIVE Ø (ERROR IF IT'S DATA!)
/MY*            :REM LOADS FIRST PROGRAM STARTING 'MY' FROM DRIVE 1.
↑1:PROGRAM,:REM LOADS 'PROGRAM' FROM DRIVE 1, THEN RUNS IT
```

Notes: [1] BASIC 4 forces dl"*[RETURN]run[RETURN] into the keyboard queue if the Shift-stop key is pressed. This is quite easy to do when editing the screen, and acts like dload "*", then run. This will erase your current program if you are using BASIC. See Chapter 17 for remedies.

[2] On DLOAD, a program is re-run from the beginning, retaining its variables. How can machine-code be loaded? Suppose we have saved 'OLD' on disk and wish to load it from a program: 0 DLOAD "OLD" loads the routine, but then starts the program over again; so the program keeps loading OLD until Stop puts it out of its misery. However, since the variables are retained, this construction may be used, provided the machine-code doesn't change variables' values:

```
0 X = X + 1
1 IF X=1 THEN DLOAD "SCREEN" :REM LOADS INTO $8000 ff
2 IF X=2 THEN DLOAD "OLD"    :REM LOADS ON SECOND RUN
3 CONTINUE FROM HERE!
```

[3] The DOS interface is Drive number:command string with secondary address zero. The program in note [3] to CATALOG & DIRECTORY illustrates this.

Abbreviated entry: dL   Token: $D6 (214)

Operation: Apart from some syntax checking, DLOAD is identical to LOAD: see Ch.5.

ROM entry point: The kernel jump address is $FFB1; this jumps to $DB3A.

DOPEN

BASIC 4 disk file command

PURPOSE: Opens a file, entering its parameters in tables, and sends a message to an IEEE device on the bus to set up its buffer. Only sequential or relative files may be opened for write with DOPEN; but any file (SEQ, REL, PRG, USR) may be opened for read. Unless otherwise stated, device #8 is assumed. This command is very similar to OPEN, except for syntactical differences, the restriction of the device number to 4 or more, and limitations on file types. See OPEN.

Syntax: DOS 1+ and DOS 2+ differ considerably: the first will not, and the second will, accept commands to set up relative files. Apart from this difference, the disks are not very different. However, DOPEN only sends a subset of the commands available from OPEN: in particular, USR and PRG files cannot be mentioned-not that they are used much anyway. For this reason OPEN can still be useful even when BASIC 4 is fitted. Note that DOS Support has no short form of OPEN. The syntax is fairly complex: the following parameters appear in any order:-

(i) #, then expression, which is the logical file number. This must be 1-255.

(ii) String or string expression in brackets, which is the name of the file to be OPENed. This has a maximum length of 16 – unless the open-with-replace option is used, when it begins with '@' and may be 17 characters long. In this case a file of the same name is overwritten without causing error 63, ?FILE EXISTS. See e.g. SAVE for warnings about this function.

(iii) Optional D followed by expression for 0 or 1. This is the drive number.

(iv) Optional unit number, denoted by ON U or ,U with expression for 4-31.

(v) Optional file type parameter. This may be *one of:*L then expression for 1-254; this is the relative file record length parameter, and write is assumed. *Or:* W alone, which indicates write, but to a sequential file. Parameters like S,R,P and so on are not accepted. The table should make this clear:

'L' parameter	'W' parameter	Signification
Yes	No	Open relative file for write to diskette
No	Yes	Open sequential file for write to diskette
No	No	Open relative, sequential, program or user file for read only

Finally, when writing to disk is involved, remember that a logical file number of 1-127 sends carriage return after PRINT# ..: while higher file numbers send carriage return plus line feed.

Examples: BASIC 4. The DOS interface has three forms, exemplified by: '1:FILE', '1:FILE,W' and '1:FILE,L,100'. The third will not work with 3040 or 2040 drives, and usually causes ?file not found error. These three types are exemplified, as they appear in BASIC, by these commands:-

```
DOPEN#5,"OLD DATA"    :REM OPEN FILE - COULD BE REL,SEQ,PRG,USR - FOR READ
DOPEN#6,"NEW DATA",W  :REM OPEN 'NEW DATA' AS A SEQUENTIAL FILE FOR WRITE
DOPEN#7,"REL DATA",L87:REM OPEN NEW RELATIVE FILE FOR WRITE. REC.LEN.=87
```

All these assume drive zero, unit eight. Most practical examples will look like them, but for completeness we may add the following examples:

```
DOPEN#8,"@NEW REL FILE",L55,D(X) ON U(Y): REM WRITE A NEW RELATIVE FILE,
WITH REC.LEN. 55, ON DRIVE X OF UNIT Y
DOPEN#9,"#"            REM OPEN CHANNEL FOR DIRECT ACCESS TO DISK
```

The first of these final two examples opens logical file number 8 to a new relative file, which, when it writes, will replace the previous file of the same name, if there is one. The record length is 55; this includes the carriage return character at the end of each record. The drive and unit numbers are expressed as variables, and are therefore controlled by the rest of the program; ?SYNTAX ERROR will appear, of course, if they are not within their allotted ranges.

The last example opens a 'direct access' channel to a disk, so that tracks and sectors may be written and read without the intervention of DOS. These can be very useful, although they are not well documented and not particularly easy to use.

BASIC<4. The following OPEN statements are exactly equivalent to the DOPEN statements which we've just examined. Note especially the format to be used to open a relative file without DOPEN; the parameter must be sent as a single byte to simulate DOPEN.*

```
       OPEN² 5,8,5,"0:OLD DATA,SEQ,READ" :REM OR OPEN 5,8,5,"0:OLD DATA,S,R" &C
  or   OPEN 5,8,5,"0:OLD DATA,PRG,READ"
  or   OPEN 5,8,5,"0:OLD DATA,USER,READ"
```

Note that OPEN 5,8,5,"0:OLD DATA,REL,READ" is accepted only by DOS 2+. The secondary address 5 has no particular significance. BASIC 4 generates its own from a table. Secondary addresses of 0,1, and 15 are, of course, already reserved for other purposes.

```
       OPEN² 6,8,6,"0:NEW DATA,SEQ,WRITE":REM OR OPEN 6,8,6,"0:NEW DATA,S,W" &C
       OPEN 7,8,7,"0:REL DATA,L" + CHR$(87)*
```

Again, this latter form can only work if the disk is fitted with DOS 2, DOS 2.1, or DOS 2.5; what I've loosely referred to as DOS 2+. The parameter for the record's length must be within the range 1-254. In BASIC 4 this is checked by DOPEN's syntax, but here it is the programmer's job to keep the parameter in acceptable limits.

```
       OPEN 8,Y,8,X$ + "@NEW REL FILE,L" + CHR$(55)
```

Where Y is the device number, and X$ is assumed to be either "0" or "1".

```
       OPEN 9,8,9,"#"
```

This is an example of a file opened for direct access to the disk unit; now, PRINT#9 is followed by commands of the B-W and M-E type (q.v.) This is a user file and it might be expected that a name could be assigned to it when it is opened, but this seems never to be done in practice

Notes: [1] There is not space here for full demonstration programs to open files/ write to them/ close/ read back. These in any case involve many of the functions of DOS, such as RECORD, DCLOSE, SCRATCH and so on. Chapter 6 has a set of disk file demonstration programs which illustrate the possible permutations and combinations of the CBM disk drives.

[2] Relative files, opened for write, are treated differently from sequential files. The latter, roughly speaking, are allocated a buffer and identification on the catalog, and are written as the need arises, so that if several are written at one time their sectors will interweave in a leapfrog-like manner. Relative files need an indexing system. On DOPEN or OPEN a buffer has to be allocated for the side sectors as well as the main file. Now, if RECORD # file-number, 200 is executed by BASIC, the entire file must be generated for 200 records. In this way the file may be created in a more orderly fashion than is possible with sequential files. This may reduce disk read errors, since the read head has far fewer track and sector skips (on average).

[3] A complete list of interface commands available through OPEN can only be made by disassembling each DOS. There are certainly more than appear in CBM documentation. Mike Todd (in IPUG, July '81) says that opening a sequential file for read, while it is being written, using something like OPEN 8,8,8,"FILE,N" enables the file to be read back, or at least its buffers. Harry Broomhall found a command using '&' which may be used as a diagnostic routine - if you know how.

Abbreviated entry: dO   Token: $CD (205)

Operation: Chapter 15 and Chapter 5 outline the workings of DOPEN and OPEN so far as BASIC goes. The other work is performed by the disk units themselves.

ROM entry point: The kernel jump address is $FF96; this jumps to $D942

---

*This is not the only example of undocumented CBM functions which use a parameter of this type. Some CBM printers can only be made to perform certain functions when a byte parameter is sent in this way.

²There's a serious bug in OPEN which shows itself if the drive number is omitted (this is sloppy programming, of course). The data default is different from the directory default, so data is written to the wrong disk unit! Hence all the drive numbers.

DS$ & DS
BASIC 4 reserved variables

PURPOSE: provides a record of the status of the disk system after any operation. In this way an error condition can be noted without stopping BASIC. The variables DS and DS$ are reset on being read. These extra variables are part of the price paid for having an external DOS. If DOS were accessible to BASIC,ST could record all status conditions, although not in its present form with only 8 values at most. However, in fact it must be read from the disk drive like other data. ST is still needed to convey information like ?device not present, while DS$ stores information about the results of disk handling. Errors can range from the gross, when for example no diskette is found, or a blank disk can't be read, to the subtle, such as a failure to exactly match a file name. These are not serious errors; in fact, they show that the system is working correctly. Hard errors, on the other hand, such as mistakes in the internal checking system of a sector, may be serious.

Syntax: DOS 1+ and DOS 2+ both treat DS and DS$ in the same way; in each case the variable is read from disk through the command/ error channel with secondary address 15. However, the variable is only recognized as DS or DS$ by BASIC 4 or some toolkit-enhanced BASIC<4 variations (notably Disk-0-Pro). Otherwise it is input, and any convenient name may be given to the variable(s). Like TI and ST, DS and DS$ are not variables in the normal sense; they are not stored in RAM with other variables, but instead are computed when they are asked for. Statements like DS=5 or DS$="HI!" are specifically filtered out by LET, but the either variable may be printed or compared:

      ?DS$
      IF DS>19 THEN PRINT DS$: STOP

DS at present may take values below 75; not all are used, some perhaps have been reconsidered, others (e.g. 2-19) never used.
DS$ has the format 'Error number,message,track,sector' where the latter two variables may be irrelevant and set to zero. '0,ok,0,0' , '21,read error,18,01' '50,record not present,24,7' are typical disk status messages.

Examples: BASIC 4. This, like the other examples, assumes the error channel is open. If it isn't, OPEN 15,8,15 will open it - the file number may be different, the device *may* not be 8, but the secondary address must be 15.

      PRINT DS$: REM PRINT FULL MESSAGE WITH FOUR PARAMETERS
      IF DS>19 THEN GOSUB 10000: REM GO TO AN ERROR-HANDLING ROUTINE.

BASIC<4. If DOS Support isn't loaded, the following can be used:
      1000 INPUT#15,EN,EM$,ET,ES
      1010 IF EN>19 THEN PRINT EN "," EM$ "," ET "," ES: CLOSE1: CLOSE 15
      1020 RETURN

If a program crashes with a disk error, the error channel may need to be read in direct mode. The easiest entry is
      oP15,8,15: iN15,e,e$: ?e,e$: rem only bother with major variables

DOS SUPPORT (UNIVERSAL WEDGE). All that is needed, provided the error channel is open, is
      @ or >.

Notes: [1] The problem with this variable is when to use it, like ST. DS could be checked after every disk operation, but the drives won't usually indicate errors if a good program is running. A program which crashes, and leaves the red LED in the centre of the drive on, is an obvious candidate for directly reading DS to find out what went wrong (and turn off the LED). Programs which INPUT# data can afford to test DS after each field is read; but can programs which us GET#? There is no general answer to this question: it depends on the degree of security which the system requires. It may also depend on such hardware factors as the age of the system, its state of maintenance, and the quality of the disks.

[2] All BASIC 4's keywords for disk handling clear DS$, DS and ST. The routine to do this, at $DBE1, is called whenever the command string buffer is filled prior to transmission on the IEEE bus. To read DS$, a routine at $D991 or $D995 is called, depending on whether the string is being fetched or re-fetched. It's read one byte at a time until a carriage return character is encountered, after making the disk a Talker on secondary address 15. The string is poked into RAM at the low end of the strings, and pointed to by ($0E), with length parameter in $0D. From here it can be printed. This process avoids any problems associated with inputting strings containing commas. DS is evaluated by converting the first characters up to the first comma into a numeral in a floating point accumulator. (In fact Disk-0-Pro evaluates this string and stores it after all disk operations, so it more closely resembles a normal variable). Some pre-BASIC 4 programs use DS and/or DS$, unaware of the promotion in store for these variables, and these programs will crash when they meet statements like 40 INPUT "SIZE OF DIRECTORY";DS$: DS=VAL(DS$) if they are run on a BASIC 4 machine. DS and DS$ can be set up, like TI and ST, as ordinary variables by poking, but this is a trick of little practical value.

[3] The table summarises the important features of disk status messages.

Message type:	Information (not an error)	Programming mistake or simple mechanical error	Hard error
Input/ Output Errors at Disk Level	0 Everything OK 1 Files scratched (gives number) 2-19 Undocumented. Not important.		20 Sector header not found 21 Sync mark not found 22 Sector not found 23 Checksum error in byte 24 Byte read error 25 Readback compare error 26 *Write protect tab on* * 27 Checksum error in header 28 Next sync mark not found
Init'ion Syntax Errors		29 Disk i.d./BAM mismatch[2] 30 Syntax error 31 Unrecognised command 32 Overlength command 33 Wrongly used ? or * in name 34 File name omitted 39 Unrec. command to channel 15	
Relative and Seq. Files	50 Expand rel. file size	50 Reading past end of file 51 Relative record too long 52 Relative file too big for disk	
File Errors		60 Attempt to read a write file 61 File not open 62 File doesn't exist 63 File does exist 64 File type mismatch	
Track & Sector Errors DOS Errors		65 Block-Allocate error: gives next available track & sector 66 Track or sector out of range. 67 System track or sector error 70 Channel to disk unavailable 71 Error in BAM[2] 72 Disk (or directory) full 74 8050 drive hasn't disks	73 *DOS mismatch*[3]

*[3] These may lead to problems later. See Chapter 6 on write-protect tabs on CBM disks and on DOS incompatibilities. [2] Generally re-initialisation is required.

DSAVE
BASIC 4 disk file command

PURPOSE: Writes a consecutive block of RAM bytes to CBM disk, usually a BASIC program, and updates the directory so the RAM dump is retrievable. DSAVE is similar to SAVE in most respects.

Syntax: DOS 1+ and DOS 2+ use the same DOS interface for SAVE. This is simply the drive number followed by the file name. An additional feature is the optional leading @ before the file name, producing the much-discussed 'save-with-replace'. DSAVE uses the following parameters, in any order, separated by commas:
(i) String or string expression in brackets. This is the file name.
(ii) Optional D followed by expression for 0 or 1. This is the drive number. The default value is drive 0.
(iii) Optional U followed by expression for 4-31. This is the IEEE device number. Its default is 8.
A null name ("") is not accepted; the name string must have length>0, even if it is only CHR$(0)!

Examples: BASIC 4. The examples are typical DSAVEs. Usually this command is used in direct mode to save a new or rewritten BASIC program, but it may be used within BASIC sometimes: see note [2].

          DSAVE "BASIC PROGRAM"   :REM SAVES ONTO RIGHT-HAND DRIVE, DRIVE 0.
          DSAVE "@BASIC PROGRAM",D1
          DSAVE D1,(X$) ON U8

If the program exists, DS$ signals error 63, unless save-with-replace (the second example) bypasses this test. The usual maximum length restriction holds: BASIC checks the name, ensuring 16 characters maximum. (More than 16 characters can be sent to the device by avoiding this test).

BASIC<4. The commands corresponding to the previous three examples are:

          SAVE "0:BASIC PROGRAM",8
          SAVE "1:@BASIC PROGRAM",8
          SAVE "1:" + X$,8

DOS Support provides no short form of SAVE.

Notes: [1] Save-with-replace. The remarks made in Chapter 5 on SAVE apply also to DSAVE. A fairly early manual mentions error 67 in DS (this is 'system track or sector error') as a possible result of DSAVE "@...". It seems best to avoid the construction with

          SCRATCH "BASIC PROGRAM",D1: DSAVE "BASIC PROGRAM",D1

[2] SAVE and DSAVE dump RAM from pointers ($28) to ($2A) or, with the old BASIC 1, from ($7A) to ($7C). The very last byte is not saved. The machine code monitor relies on this to save different areas of RAM. In fact it is easy to save from BASIC, using only a few pokes to reset the 'start of BASIC' and 'end of BASIC' to new values, and repoke them to the true values after saving your data. The following example shows how a screen can be saved; in future, when it is loaded, its contents will replace whatever was present on the screen. This can be useful in demonstrations, graphics, and so on.

          57000 PL=PEEK(42): PH=PEEK(43): REM STORE END-OF-BASIC POINTERS FOR BASIC>1
          57010 POKE 40,0: POKE 41,128: POKE 42,0: POKE 43,136: REM OR 132 IF 40 COLS
          57020 DSAVE   "SCREEN PIC"   :REM OR SAVE "0:SCREEN PIC",8
          57030 POKE 40,1: POKE 41,4: POKE 42,PL: POKE 43,PH : REM RESET POINTERS
          57040 RETURN

Now, drive zero has 'SCREEN PIC' stored on its disk. In direct mode, DLOAD or LOAD will put it straight into the screen. In program mode, you'll need something like this (see DLOAD and LOAD for the reason):

          0 ON X GOTO 400,600,1000,..
          990 X=3: DLOAD "SCREEN PIC",DO
          1000 REM CONTINUE HERE WHEN 'SCREEN PIC' IS LOADED AND PROGRAM RESTARTS

Abbreviated entry: dS   Token: $D5 (213)

Operation: Chapter 15 and Chapter 5 under SAVE discuss this command.

ROM entry point: The kernel jump address is $FFAE; this jumps to $DB0D.

HEADER

BASIC 4 disk system command

PURPOSE: HEADER (or NEW) formats a blank disk which then becomes usable for any CBM disk operation. It allows a name to be given to the disk, and a two-character identifier. There is an alternative short form which erases a disk's directory and enables the disk to be renamed, but does not change the identifier. NEW as a disk command - *not* the same as BASIC NEW - is the BASIC<4 version of HEADER.

Syntax: DOS 1+ and DOS 2+ have identical DOS interfaces for this command: 'N D1:F1 [,D0]' where N or NEW signals the command, D1 and F1 are drive number and header name respectively, and D0 is the optional identifier. Although the interfaces are identical, they are processed differently, or so the document-ation says. BASIC 4 has several differences compared with earlier BASICs; the syntax is different, the cautionary ARE YOU SURE? message prevents accidental disk erasure, and ?BAD DISK ERROR appears if, after the disk unit is finished, DS exceeds 1. This is checked within the BASIC 4 ROM.

HEADER uses the following parameters in any order, separated by commas:
(i) String or string expression in brackets; this is the header name, and it is checked to ensure that it doesn't exceed 16 characters.
(ii) D with expression for 0 or 1. This drive parameter is compulsory.
(iii) Optional 2-character identifier. This is entered as I followed by the two characters (e.g. A1 or 00 or $a or zz etc.). Note that these two characters are stored in $033F and $0340 and could be poked in if a modified HEADER command were used. The characters are simply read directly by BASIC. So I (X$) or I4, or I , where the syntax is variously wrong, yield i.d.s of (X and 4, and , in order.
(iv) Optional unit number. U followed by an expression which evaluates to 4-31.

Examples: BASIC 4. The first is a 'long', the second a 'short', header or new.
```
HEADER D 1, IRW, "PRICE LIST PROGS"
HEADER D0,"EXPERIMENTS"
HEADER (X$), D1 ,U9            :REM LEN(X$) MUST BE >0 AND <17
```
The final example shows how a string expression is used.

BASIC<4. Exact equivalents are given, omitting the BASIC 4 keyword 'HEADER':
```
PRINT#15,"NEW1:PRICE LIST PROGS,RW",8: REM ANY ALPHABETICS OK, EG NOUGAT
PRINT#15,"N0:EXPERIMENTS",8
PRINT#15,"N1:" + X$,9
```
In this case, ARE YOU SURE? won't appear and DS or DS$ must be read by the programmer if something seems amiss. The system won't do it for you.

Notes: [1] If HEADER or NEW shows an error, read DS to establish the cause. It may be something as trivial as a missing disk. Or it may be that the disk is of poor quality, or a write-protect tab may be in place. There is a potential risk in this situation that other disks will be partly erased: see Chapter 6 on write-protect tabs for example. If the disk won't be formatted switch off before using other diskettes if you wish to be certain that they won't be corrupted.

[2] The shorter version is faster: say 45 seconds against several minutes, depending on the DOS. Note that part of the directory is cleared, but the files all remain on disk without their pointers, a situation rather like BASIC NEW when a BASIC program is 'erased'. Issue 10 of Compute! ('Disk File Recovery Program', by David L. Cone) has a program to make data recovery possible.

[3] After HEADER put DOS Support (or any other program which you would like to be the first to load) on the disk *before* other programs.

[4] $DB9E (56222) prints ARE YOU SURE? and expects either Y or YES (unshift-ed, with no spaces) and clears carry if it finds it. Not very usable from BASIC.

Abbreviated entry: hE   **Token:** $D0 (208)

Operation: See Chapter 15 for the BASIC ROM processing.

ROM entry point: The kernel jump address is $FF9F; this jumps to $D9D2.

INITIALISE
Disk system command unavailable directly in BASIC 4

PURPOSE: Causes the disk unit to read the directory and the block allocation map (BAM) from a diskette in drive 0 or 1 or both. This operation may be performed in direct mode - when starting up, for example - or in program mode. It is not needed usually in drives fitted with DOS 2+.

> *NOTE: Some disk systems (e.g. Apple/ITT 2020) use the same command INITIAL-ISE to format a diskette, an operation called HEADER or disk NEW in the CBM system. INITIALISE will not damage the data on a CBM disk; it will erase it from an Apple/ITT disk.*

Syntax: DOS 1+ and DOS 2+ all recognize this command and employ the same DOS interface 'I [,D0]'. 8050 units, sensibly, perform this as a hardware feature, so disks cannot be sneakily changed without initialisation. 4040 units don't have this feature; instead DOS checks, before some operations, that the disk i.d. matches the disk in the drive. See note [1].

The syntax is PRINT#15,"I [,0 or 1]" or PRINT#15,string expression which evaluates to "I [,0 or 1]".

Modes: Direct and program modes are both accepted.

Examples: oP 15,8,15: pR15,"i": 10 "*",8 :rem do this on switching on.

This is the short form of OPEN 15,8,15: PRINT#15,"I": LOAD "*",8 and may be used with any BASIC/DOS combination to initialise two disks, then load the first program from drive 0 into RAM. I've assumed device #8. Shift-Stop with BASIC 4 will load and run the same first program. It uses DLOAD "*".

```
PRINT#15,"I0"  and
PRINT#15,"I1"
```
Initialise disks in drive 0 and drive 1 respectively (provided channel 15 is open).

DOS SUPPORT (UNIVERSAL WEDGE). PRINT#15 is performed by this wedge, so
```
@I   or   >I
@I0  or   >I0
@I1  or   >I1
```
may be used in direct mode to initialise both disks, drive 0, or drive 1.
Programs running disks equipped with DOS 1+ need to initialise new disks put in the drives; and 4040 disk units will need this too for disk operations which do not read the directory (see note [1]). Either type of routine is satisfactory:-
```
100 INPUT "DRIVE";D$: D$="I"+D$: PRINT#15,D$ :REM EXCLUDES VALIDATION
100 PRINT#15,"I0"
```

Notes: [1] After initialisation, the disk read/write head is left positioned over the directory track (18 or 39 in the 8050), ready to read/write to sectors of the disk; this central position is used to cut down head seeking time. If a disk hasn't been initialised the DS$ error message will read DISK ID MISMATCH (error 29). This is true unless the disk has the same i.d. as the BAM, but nevertheless was not the disk initialised, in which case data will simply overwrite the disk now in place, because DOS cannot tell that it has changed. For this reason strong warnings are often made of the disasters which may happen if disk i.d.s aren't carefully selected to be different. Similarly, backup disks should be treated with care, stored, perhaps, in another place. Automatic initialisation of 4040 disk units is *not* invoked by BACKUP, COLLECT, HEADER, or RECORD, or any direct access command. To be on the safe side, therefore, initialisation, using the format above, may well be carried out before these commands are used; for example, COLLECT on a wrong disk may scramble up data.

[2] A CBM manual gives this method for turning off auto-initialisation, apparently for 4040 disks: PRINT#15,"M-W" CHR$(243) CHR$(16) CHR$(1) CHR$(1)

RECORD

BASIC 4 relative file command

PURPOSE: Positions the relative file pointer to the start of any record within the relative file, or to any byte position within a record. This command is unusable with sequential files, and works with DOS 2+ only, not DOS 1+. It may be simulated by BASIC<2, provided this BASIC runs DOS 2+ disks.

Syntax: DOS 1+ has no relative file facility, unless specially written direct access commands are used. DOS 2+ accepts RECORD; the DOS interface is
P sec. address byte record no. low then high byte position within record.
As an example, the string "P"+CHR$(5)+CHR$(9)+CHR$(0)+CHR$(1) sent with PRINT#15 to the disk positions the pointer to the relative file opened with secondary address 5 at the first byte of record 9. The syntax of RECORD is less baffling. Unlike most other BASIC DOS commands the order of its parameters is invariable. The syntax is:
RECORD # expression for logical file (1-255), expression for record number (0-65535) [, optional expression for byte within the record (1-254)].

If the byte parameter is omitted, the value 1 is assumed by default.

Examples: BASIC 4. The short illustrative program opens a relative file and writes ten records into it. See Chapter 6 for a longer example. Note that the records are shorter than their permissible length. This is possible because the starting point of every record is computed by the system, and because a record, as it is written to disk, is terminated by Return. On reading back, by INPUT# for example, the record is read until the next Return is encountered, so the lack of data at the end of the record has no effect. There is one proviso, however – RECORD must be issued before each PRINT# statement, to move the pointer to the required record. If this isn't done – try renumbering line 30 as line 15 – the records may be written consecutively. This could be a useful feature, but appears to be unreliable. It's safer to use RECORD with the byte parameter.

```
10 dopen #1,"random file",140  :rem length parameter implies open for write
20 for j = 1 to 10
30 record #1,(j)
40 print #1,"abcdef" + str$(j)
50 next: dclose #1
```
I have omitted references to DS$ from this program. If it is checked after line 30, it will return status message 50, 'record not present', showing that the file is being expanded. As an example of the byte parameter, run this program too:
```
10 dopen #1,"random file"      :rem open for read, by default
20 for j = 1 to 10
30 record #1,(11-j),4          :rem read records in order 10-1 starting at 4
40 input#1,x$: print x$: next: dclose#1
```
This prints def 10 then def 9 then def 8 ...

BASIC<4. The equivalent programs are as follows, avoiding BASIC 4 keywords:
```
 0 OPEN 1,8,5,"0:RANDOM FILE,L," +CHR$(40)
_0 FOR J = 1 TO 10
30 PRINT#15,"P" + CHR$(5) + CHR$(J) + CHR$(0) + CHR$(1):REMEMBER CHANNEL 15!
40 PRINT#1,"ABCDEF" + STR$(J)
50 NEXT: CLOSE 1
```
and
```
10 OPEN 1,8,5,"0:RANDOM FILE"  :REM READ ASSUMED
20 FOR J = 1 TO 10
30 PRINT#15,"P" + CHR$(5) + CHR$(11-J) + CHR$(0) + CHR$(4)
40 INPUT#1,X$: PRINT X$: NEXT: CLOSE 1
```

Notes: [1] 'Error' 50 is generated whenever a relative file is expanded beyond its currently allocated limits. If INPUT# attempts to read beyond these limits, the same message appears, and ST is set to 64. BASIC returns carriage return.

Abbreviated entry: reC   Token: $CF (207)

Operation: See Chapter 15 (BASIC 4 references from $D7AF).

ROM entry point: The kernel jump address is $FF9C; this jumps to $D7AF.

RENAME

BASIC disk file command

PURPOSE: Changes the name of a disk file. Any type of file may be renamed. If the new name selected is already present on the diskette, DOS generates error 63, file exists.

Syntax: DOS 1+ and DOS 2+ each have this command. In each case the DOS interface is identical: 'R D1:F2 = D1:F1' where F1 and F2 are the old and new names, and D1 is the code for whichever drive the file is on. The internal operation of each DOS may be different. Certainly Harry Broomhall*detected errors in DOS 1 in which scratched entries in the directory made RENAME (or to be precise PRINT#15,"R ...") fail to work. Like COPY, this operation has the odd feature of having its parameters reversed in its two versions.

RENAME has the following parameters in any order separated by colons:
(i) Two strings, or string expressions in brackets, separated by TO. These of course are the before-and-after file names.
(ii) Optional D with an expression for 0 or 1. This is the drive number; the default is drive 0.
(iii) Optional U with expression for 4-31. This is the device number; its default value is 8.

Examples: BASIC 4. A couple of typical RENAMES follow. Note that COPYing a file to the same drive - if there's space for it - or to another drive, then scratching the original and recopying the file back if it's now on another disk, has the same effect as RENAME.

```
RENAME D1,"OLD NAME" TO "NEW NAME"
RENAME (X$) TO (Y$)
1000 rename ("file" + str$(x)) to ("file" + str$(x+1))
```

The final example shows how a file's name can reflect a version number after being updated, for example, so that its current standing or historical status is easy to see. A date or some other meaningful numbers or letters can be used in the same general way.

BASIC<4. The examples which follow are identical in effect to the three just printed, but don't use the BASIC 4 keyword. They will therefore work with any DOS and any disk (subject to possible bugs ... see note above, and note [1]).

```
PRINT#15,"RENAME1:NEW NAME=OLD NAME" :REM NOTE THE ORDER!
PRINT#15,"R0:" + Y$ + "=" + X$
1000 PRINT#15,"REN0:" + "FILE" + STR$(X+1) + "=" + "FILE" + STR$(X)
```

Notes: [1] This doesn't work with unclosed files. They should be closed in any case.

Abbreviated entry: reN

Token: $D8 (216)

Operation: Apart from the syntax check this is carried out entirely by DOS.

ROM entry point: The kernel jump table address is $FFB7; from here the routine jumps to $DB55.

---

*Harry Broomhall (of Heronview Ltd) is a British authority on the CBM and its disks

SCRATCH

BASIC 4 disk file command

PURPOSE: Deletes one or more files from disk. CBM's pattern matching sequence may be used - with caution - to scratch several files. DS$ returns DS=1 followed by 'files scratched' and a parameter showing the number of files which have been scratched. 'Scratch' is a word peculiar to Commodore; most systems 'erase' or 'delete' files.

Syntax: DOS 1+ and DOS 2+ use the same DOS interface. BASIC 4 however only sends a subset of the possible command string, namely 'S D1:F1' where S signifies SCRATCH, D1 is a drive number and F1 a file name. As the examples in BASIC <4 demonstrate, more elaborate constructions may be used. The penalty for making a mistake is high - a scratched file is not easy to recover.
SCRATCH uses the following parameters in any order separated by commas:
- (i) String, or string expression in brackets. This is the 'name string'; often simply a name, it may include * and/or ? symbols, and thus be treated as a string holding pattern-matching symbols.
- (ii) Optional drive number, D followed by an expression for 0 or 1. This defaults to drive 0.
- (iii) Optional unit number, U followed by an expression for 4-31. This defaults to unit number 8.

Note that scratch prints ARE YOU SURE? and expects Y or YES, in direct mode. Like HEADER, when called from a program, this question-and-answer precaution is omitted. After SCRATCH, and again only in direct mode, DS$ is read, and if it is not null (i.e. if some files actually were scratched) printed out, as for example 01,FILES SCRATCHED,03,00 when 3 files were scratched. This same sequence of messages is obtainable from BASIC<4, but must be input from the error channel manually.

Finally, note that there is no warning if a file to be scratched doesn't exist. If its name was misspelt it'll still be there.

Examples: BASIC 4. These examples are, I hope, self-explanatory. Note the program example; this is the way to avoid dopen "@test",125.

```
SCRATCH "FIND CHECKLETTER",D1:REM SCRATCHES A PROGRAM (OR COULD BE DATA)
SCRATCH D0,"DIS*"            :REM SCRATCH ALL STARTING 'DIS'
scratch d0,"*",u9            :rem scratch all on drive 0 of unit 9

100 scratch "test file",d1
110 dopen "test file",d1,125
```

BASIC<4. The equivalents to BASIC 4 follow. There's an extra example to show the extended syntax possible with this version of the command.

```
PRINT#15,"SCRATCH1:FIND CHECKLETTER"
PRINT#15,"S0:DIS*"
PRINT#16,"S0:*"              REM ASSUMES OPEN 16,9,15 TO UNIT 9
100 PRINT#15,"SCR1:TEST FILE"
110 OPEN  1,8,8,"1:TEST FILE,L," + CHR$(25)

PRINT#15,"S1:TEST,1:MC.OLD,0:X-FILE4,0:TESTER"   :REM MULTIPLE DELETES
```

Notes: [1] Files currently open, or thought to be open, aren't scratched. COPY is said to have the effect of using internal channels so that SCRATCH believes the file to be open, and leaves it. It can be scratched later.

[2] Don't scratch unclosed files; COLLECT or VALIDATE the disk. Otherwise the last existing sector will not point to a sector with a zero termination byte, and this sector may apparently connect with another file, so DOS will scratch parts of that too. Suppose a sequential file, writing to disk, signals 'disk full'; its directory is marked '*' as unclosed. SCRATCH now replaces this with file type DEL, and will probably produce strange effects. But COLLECT is fine, or read and write back with CLOSE if the data is valuable. Or ?syntax error may abort a write file; re-open channel 15, then close all files. Make COLLECT/VALIDATE a rule.

Abbreviated entry: sC   Token: $D9 (217)
ROM entry point: The kernel jump address is $FFBA; this jumps to $DB66.

CHAPTER 8: OTHER PERIPHERALS AND HARDWARE

8.1 Tape cassettes.

Cassettes and tape recorders Commodore's tape recorders exist in three forms: built into the machine, in the earliest models; the C2N external recorder; and VIC's tape recorder. There have been internal changes in printed circuit board construction, so that special load/ save routines may not operate with every model. The C2N has a short cable and edge-connector; power is supplied from the PET/CBM, so the device is not easily usable away from the computer. VIC's recorder is in white plastic in place of the earlier black. It is compatible with the PET/CBM and cheaper. Each PET/CBM has ports for two cassettes, and a small amount of software exists which keeps files on one tape, reading and updating them onto the second. The ports are arranged differently in each of the main designs; see the diagram in Chapter 1. This is usually no problem, but from time to time, when switching machines, a user may be surprised to find no response from the machine, because he is addressing the wrong port. The recorders are assigned device numbers 1 and 2, and the whole of their operating system is in ROM. There are improvements in BASIC>1 over the original, but the main features are identical in all ROMs. Consequently external tape recorders, which are portable and robust, are often useful in transferring programs between machines even when they are equipped with disk drives, because there are few compatibility problems. The top of the edge-connector should be labelled; it's often possible to connect it upside-down, when recording won't take place.

All tape recorders use similar principles: there is an *erase head* and a *recording head,* arranged so that during recording the tape is first demagnetised, then recorded. Physically, the recoding takes the form of vertically magnetised fields on the tape, their form depending on the amount and frequency of the magnetic flux generated in the recording head. On reading back, the erase head is off and the recording head acts in reverse as a read head, the tape, as it passes, inducing current in it which is amplified. (Some machines have separate heads for recording and playback). The *capstan* and *pinchwheel* drive the tape at a constant speed; the capstan rotates, driving both the wheel and the tape, whenever they are brought in contact by pressing 'Play'. The leading (right-hand) spool is maintained under tension, so the tape is tightly wound, and variations in the effective diameter of the take-up spool have no effect. Leaving 'Play' pressed when the recorder is off may cause the (now static) capstan to dent the pinchwheel, and cause irregular playback. In fast forward or fast backward mode, the capstan is disengaged and drive applied directly to one or other spool. Routine recommended maintenance involves cleaning and demagnetizing; again, all recorders are much the same, and cleaning kits for non-computer cassettes are fine, consisting of cotton-wool swabs and solvent (e.g. isopropyl alcohol) to remove tape debris. Demagnetizing is always recommended, and sometimes carried out. The movable type of demagnetizer, relying on the inverse-square law to magnetize the head in alternate directions with ever-decreasing flux, seem to be best. Head alignment problems may arise when using tapes recorded on equipment different from that used in playback. A recorder with its recording head not vertical will usually read back without difficulty, because reading exactly matches the magnetic pattern deposited on writing, a recording made with the same machine. If a recorder has persistent difficulty in loading tapes, this may be the reason. Adjusting the head is fairly easy.

Cassette tape is cheap, portable, and easy to send through the post. (Some tape 'magazines', e.g. 'Cursor', and Petsoft and Commodore cassette programs bear commercial witness to this). The best type is ferric oxide (not chromium) of reasonable quality. A screw-type casing (which can be taken apart if the tape is tangled) may be better than the glued type. We shall see how to estimate storage on tape; the best length of tape depends on the user's purpose, some preferring C-10 or C-12, others C-45 or C-60. Avoid thin tape. In principle it's a good idea to test tape, and a number of test programs exist.* Unfortunately this is a time-consuming process, much more so than with disks. The best compromise is probably to test tapes to be used for 'master' storage. Three tips: (i) don't save useful stuff directly onto a brand-new tape; test it or unwind and rewind it first to unstretch it; (ii) it may help to demagnetize tape, since PET/CBM uses high recording levels, to erase old programs or data; (iii) record the first program/data with a few seconds' extra leader.

*J Butterfield (e.g. CCN Sept.'81) and Kilobaud-Microcomputing (March '80) for example.

Commodore's newest cassettes are equipped with a tape counter; the older models are not. Section 8.4 has information on ways around the restriction. It has additional material on fast forward winding and the possibilities of constructing tape directories.

8.2 Data Storage on CBM Tape.

Introduction Commodore's tape system stores data coded as square waves. The diagram shows, as an example, how a byte is stored; it has a marker followed by nine bits, the last being the parity bit. (Odd parity is used).

←—Byte Marker—→←——Low Bit→ ←—High Bit→.... Parity Bit

←——— 684μs→ ←—524μs—→ 364μs 524μs—→ ← 524μs 364μs
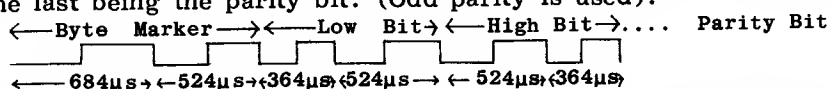
There are three frequencies, which we can call short, medium and long. A byte marker is one long wave followed by a medium wave. A bit is either a single short wave followed by a single long wave (bit 0), or vice versa (bit 1). The periods in microseconds in the diagram were quoted in an article by M Maynard of Audiogenic Ltd. The actual process of record and readback is complex. So far as I know, source code for the operating system has not been released.*

Data storage with BASIC There are two types of file available, which are exactly analogous to disk files' PRG and SEQ file types. BASIC programs are held, like machine code routines, as a continuous dump from memory. The header holds the load address, so the program may be reloaded in the correct (i.e. original) place in RAM. Data is stored sequentially in ASCII form, i.e. as if PRINTed to the tape, and, like disk sequential files, the result can be read back, but not updated directly. This type of storage needs buffers in RAM, since there is no equivalent to the entire program in memory; instead, data is generated and stored in a buffer. When the buffer fills, it is written to tape and emptied. Conversely, when a file of data is read from tape, the cassette motor automatically runs from time to time, loading the next batch of data. At this point, without looking at the separate items stored on tape, lets consider tape timing, and the estimation of the storage capacity of tape in the CBM system.

PROGRAM FILE: { -- LEADER --	HEADER	TONE	--PROGRAM (1)--	--PROGRAM (2)--		}		

DATA FILE: { -- LEADER --	HEADER (1) (2)	TONE	1ST BUFFER (1) (2)	TONE	2ND BUFFER (1) (2)	TONE	... }	

TIME (SECS):   10   4   2 About .009 seconds per byte /2+ sec. gap

We can see from the diagram that 10000 program bytes are written or read in about 196 seconds. 10000 bytes of data take more time, because of the 'wastage' caused by the inter-block gaps. The increase is something like 60%. (It may appear to be more, because the buffers need to be filled too). The amount of tape used is increased in proportion. Using approximate figures, 10000 program bytes require 3 1/3 minutes to store or load; 10000 data bytes require about 5 minutes.

CASSETTE TYPE:			C10	C20	C30	C60
Minutes per side:			5	10	15	30
PROGRAM STORAGE SPACE	1K average		8	16	25	50
PER SIDE:	5K average		2	5	8	16
	10K average		1	3	4	9
DATA STORAGE SPACE	1 file		9K	20K	30K	60K
PER SIDE:	5 files		1½K	3K	5K	11K

---

*There is little published material on tape storage. Those wishing to know more might disassemble their ROMs and examine the result in association with the notes in Chapter 15. A pair of articles in Compute! (G Campbell, Sep./Oct. '80 and K Falkner, Jan. '81) has, respectively, a method to load Applesoft programs (Apple floating-point BASIC) into the CBM, and an Apple program to load CBM tapes. The latter is a well-documented source listing, with error-recovery 'better than the PET'. Much of it converts tokens into the Apple equivalents, and performs other functions not very relevant to the CBM. 'Rabbit' is a 2-K package (available on tape or as a 2-K ROM (A000-A7FF) for fast load/ save of programs only. Its rewrite of the operating system doesn't work with every recorder. PCW (M Shelley, Jan. '81) has a BASIC 2 routine, with explanation, which is designed to help recover partly overwritten tapes. (The explanation, however, says nothing of its actual procedure). "Arrow" is another fast tape system in EPROM.

The table gives a guide to the storage capacity to be expected from typical cassette types.* Thus, a C20 cassette will hold about 5 5-K programs, if they are recorded consecutively. Each will take about 2 minutes to record or read, because 10 minutes of playing-time is shared between 5 programs. 10K of data takes a little more than five minutes to record, and so on. Tapes can be played on ordinary recorders[2] and the phenomena illustrated in the pair of diagrams on the previous page verified. There is a short tone separating the two halves of the program and the block recordings, each of which is recorded twice for security.

LOADing and SAVEing BASIC programs For detail on the workings of these commands, see Chapter 5. Briefly, the syntax appears like this:

```
LOAD "NAME",1 or LOAD "NAME",2   :REM SEARCH FOR PROGRAM CALLED 'NAME' ON
                                  CASSETTE#1 OR CASSETTE #2
LOAD                             :REM LOADS FIRST PROGRAM ON CASSETTE #1
LOAD "",2                        :REM LOADS FIRST PROGRAM ON CASSETTE #2
LOAD "NAME"                      :REM LOADS 'NAME' FROM CASSETTE #1
```

These examples show that the default is device 1. The instruction 'PRESS PLAY ON TAPE #1' or '#2' always appears on the screen; further display includes 'OK' when the key is sensed, and 'READY.' or '?FILE NOT FOUND ERROR' or '?LOAD ERROR', depending on the success of the search and load. When the program's header block has been found, the message 'LOADING' or 'LOADING NAME' also appears on the screen. If the LOAD is carried out *in program mode,* most of these messages (i.e. unless there is an actual error) are suppressed, to keep the screen relatively tidy. And if 'Play' on the cassette is down *before* LOAD, no messages at all appear, unless there is some error. Loading from within a program causes the new program to begin execution once it has finished loading; the object is to enable programs to chain, keeping their variable values. With small (say 8K) machines, this can be useful, both in extending the processing capacity, and spreading the tape-reading time out. This may require care with the relative lengths of programs, and with strings (which must be stored in high RAM if they are to transfer properly). Function definitions also aren't generally carried over between programs. (Chapters 2 and 5 explain in detail).

```
SAVE is equally simple:
SAVE "PROG NAME",2              :REM SAVES BASIC IN MEMORY ON TAPE #2 AS
                                'PROG NAME'
SAVE                           :REM SAVES BASIC IN MEMORY ON #1 WITH NO NAME
```

Security. To be on the safe side, SAVE important programs twice, either on the same tape or on a master tape. Note that SAVE always begins the recording process at the point at which the tape is positioned; there is no searching process as there is with LOAD. As for LOAD, ?LOAD ERROR occurs if bit 4 of ST is set. Sometimes errors can occur in the absence of this message, and recommended practice is to print ST to check that it is zero as it should be. A further refinement is to look at the results of the error-correcting processes of the operating system: PEEK(192) for tape #1, or PEEK (193) for tape #2, should also be 0, although values up to and including 4 are also acceptable.[3] (For BASIC 1, the peek locations are 630 and 631 respectively).

LOADing in program mode does not reset ST, so these tests can be incorporated in the start of a new program, if a wrong load would cause trouble. Or a machine-code routine to calculate a hashtotal of the program bytes may be used, similar to that suggested for disk-based programs in Chapter 6. However, these methods are normally used only when tapes are being tested or head alignment is being checked.

---

*The figures are calculated on the hasis that each file has 16 seconds of leader; then program storage (BASIC or machine-code) takes a further .009 x 2 seconds per hyte, since each byte is recorded twice. Data storage takes place in 191 hyte chunks, each heing written twice. The numher of hlocks required must he rounded up; each takes ahout 2 seconds for the inter-hlock gap and 2 x 192 x .009 seconds to record or read. The total is about 5½ seconds per 191 hytes (which is rather slow). For example, a 5000 hyte program file records in 16 + 90 = 106 seconds. 5000 data hytes need 27 blocks, taking 16 + 27 x 5½ = 165 seconds.

[2] If a speaker is attached to the machine, as explained in Chapter 9, a pin on the user port can be attached, like CB2, and the program listened to while loading. Pins 6 and 7, for tape #1 read and tape #2 read (see manual) are the ones.

[3] See e.g. D Isaacson, 'Detecting Loading Prohlems...', in Compute! Jan.'81

End-of-tape Marker When a tape is being written (either by SAVE or by OPEN/
PRINT#) a secondary address of 2 – strictly, of a non-zero even number – causes an
'End-of-tape' marker to be written when the SAVE is finished, or the file CLOSEd.
This is quite a simple idea, but can cause some puzzlement. An extra buffer is written
onto tape, exactly like any other buffer except for its very first value. When this tape
is read back, a buffer of this sort is interpreted as the end of the tape, and the
reading process will go no further. The marker need not actually coincide with the
physical end of the tape; a 200-byte program can be saved with an end-of-tape mark,
so if the tape is searched for some other program, reading will go no further than
the marker, and the user will be spared the tedious process of waiting for the rest of
the cassette to read. Thus:

```
SAVE "IMPORTANT PROGRAM",1,2
OPEN 7,1,2,"MAJOR FILE"
```

respectively save BASIC to tape 1, with end-of-tape marker, as 'IMPORTANT PRO-
GRAM', or rather as the first 16 characters of this name; and open a file, for writing,
to tape 1, called 'MAJOR FILE', again with end-of-tape.

Writing and reading files: OPEN, PRINT#, INPUT#, GET#, and CLOSE These BASIC
commands (with CMD) are all that is required for tape files. Although these files are
really quite straightforward, they aren't all that easy to get used to; the next page
has a simple demonstration program, which writes data and reads it back, printing
the results at each stage on the screen. Lines 10-60 open a file for writing, without
assigning it a name, and print 256 values to tape. Lines 100-160 read them back, with
GET#, so that each byte is separately read from tape; you will see that carriage ret-
urn characters act as record separators. Lines 200-260 read the same file with INPUT#,
which accepts only a range of ASCII values, and also implicitly regards carriage ret-
urns as separators.
    OPEN and CLOSE typically look like this:

```
OPEN1           :REM OPEN LOGICAL FILE 1 TO DEVICE 1 FOR READ
OPEN 3,2,1,"DATA" :REM OPEN FILE 3 TO TAPE 2 FOR WRITING; NAME IT 'DATA'
OPEN 1,1        :REM SAME AS 'OPEN 1'
```

As these examples suggest, the four parameters (logical file, device, secondary add-
ress, and name) have defaults, apart from the compulsory file number of 1-255. The
device defaults to cassette #1, the mode to 0, or read, and the name to spaces. De-
faulting to read is of course intended to avoid accidental overwriting of data.
    When a file is closed, its last buffer is written, with an end-of-file zero byte.
This is picked up on readback, setting ST to its end-of-file value. It is probably
better to write some end-of-tape marker, or to include a count of the records being
written, than to rely on ST, the treatment of which varies between ROMs.
    Tape files are sequential files, like disk files of type SEQ, and they are subject
to the same rather painful restrictions. These are made worse with tape by the fact
that only one file may be open at once, unless two cassette units are available. The
attempt to OPEN one file for read, and another for write, although legal, leaves the
tape positioned as though only the second command had been issued. Tape data files
are therefore used only to hold data which was input, or which is for printing, or
which can be updated entirely in RAM before rewriting to tape, unless there is a sec-
ond cassette or disk unit.
    BASIC 1 (the oldest BASIC version) has two bugs in its data file operating
system: see section 8.4 for corrections.

Storage of machine-code programs and data via the monitor The syntax for save or
load from the monitor is illustrated by these examples:

```
.S "MACHINE CODE",01,0400,0615
.L "MACHINE CODE",01
```

which save the contents of memory from 0400-0614 on tape #1 as 'MACHINE CODE', and
load it again, respectively. Note that .S operates from the start address until its point-
er equals the end address; the last byte does not get written to tape. (An end-of-tape
marker can be forced by poking $D3 = 211 with 2. (BASIC 1: $F0 = 240). As we shall
see (Section 8.4) this process can be carried out from BASIC, without entering the
monitor, when the appropriate locations have been found. Before this we'll examine the
way data is held on tape in rather greater detail, which includes the structure of the
header block and the four types of buffer which the PET/CBM system has.

TAPE DEMONSTRATION PROGRAM: Showing how to write to tape, and the differences
between the two methods ('GET and 'INPUT') of reading it back.

```
0 REM        ################################################
1 REM        # SHORT BASIC TAPE DEMONSTRATION PROGRAM #
2 REM        ################################################
3 REM
4 REM
5 REM        #############################
6 REM        ######   WRITE TO TAPE  #####
7 REM        #############################
8 REM
9 REM
10 OPEN 1,1,1 :REM OPEN THE FIRST TAPE RECORDER FOR WRITING; FILE NUMBER = 1
20 FOR J = 0 TO 255
25 X$ = CHR$(J) :REM THE LOOP GENERATES ALL THE POSSIBLE SINGLE CHARACTERS
26 PRINT J;:REM SHOW ON SCREEN THE ASCII VALUE OF THE CHARACTER BEING WRITTEN.
30 PRINT#1, X$ :REM PRINT A SINGLE CHARACTER - AND ALSO A CARRIAGE RETURN
40 NEXT
60 CLOSE 1,1,1
70 PRINT: PRINT "WRITING TO TAPE IS COMPLETE.
71 PRINT: PRINT "PLEASE STOP THE TAPE RECORDER AND REWIND THE TAPE;
72 PRINT: PRINT"THEN PRESS ANY KEY TO CONTINUE.
73 GET X$: IF X$ = "" THEN 73 :REM WAIT FOR ANY KEY TO BE PRESSED
94 REM
95 REM        ##################################################
96 REM        ##### GET SINGLE CHARACTERS FROM TAPE #####
97 REM        ##################################################
98 REM
99 REM
100 OPEN 1,1,0 : REM OPEN THE FIRST TAPE RECORDER FOR READING; FILE NUMBER = 1
120 FOR I = 0 TO 520 :REM THE LARGER NUMBER OF 'GETS' HAS TO ALLOW FOR RETURNS
130 GET#1, X$ : REM GET A SINGLE CHARACTER FROM TAPE
135 IF X$ <> "" THEN PRINT ASC(X$); :REM KLUDGE, BECAUSE ASC("") IS DISALLOWED
136 IF X$= "" THEN PRINT "NULL"; : REM FOR REASON GIVEN BEFORE
140 NEXT
160 CLOSE 1
170 PRINT: PRINT "GETTING SINGLE CHARACTERS FROM TAPE IS COMPLETE.
171 PRINT: PRINT "PLEASE STOP THE TAPE RECORDER AND REWIND THE TAPE;
172 PRINT: PRINT"THEN PRESS ANY KEY TO CONTINUE.
173 GET X$: IF X$ = "" THEN 173 :REM WAIT FOR ANY KEY TO BE PRESSED
194 REM
195 REM        ################################
196 REM        ######  INPUT FROM TAPE  #####
197 REM        ################################
198 REM
200 OPEN 2,1,0 : REM OPEN THE FIRST TAPE RECORDER FOR READING; FILE NUMBER.= 2
210 FOR I = 0 TO 260 :REM NOTE THAT THE SMALLER NUMBER APPLIES NOW
220 INPUT#2, X$ : REM THIS TIME, INPUT A SINGLE CHARACTER FROM TAPE
230 IF X$ <> "" THEN PRINT ASC(X$); :REM KLUDGE, BECAUSE ASC("") IS DISALLOWED
240 IF X$= "" THEN PRINT "NULL"; : REM FOR REASON GIVEN BEFORE
250 NEXT
260 CLOSE 2
270 REM        ###########
271 REM        ##  END  ##
272 REM        ###########
```

Headers, cassette buffers, and blocks When a program is saved to tape, or a file is
opened to write to tape, the cassette operating system writes a 'header' to the tape.
This is a single buffer of data, containing the program or file name, two addresses,
and a single byte at the start, which the system identifies as the marker for a header.
Conversely, when a program is loaded, or a file opened for read, the operating system
searches the tape for blocks of the form which declare themselves as headers. The
name is checked, and, if it matches the required name, loaded into RAM (program) or
stopped until GET# or INPUT# asks for data from subsequent buffers. OPEN 1 loads
the first header on tape into the cassette buffer for the (default) device, tape #1;
OPEN 1,2,0,"HELLO" loads HELLO's header, from cassette #2, into cassette buffer #2.
The RAM buffers are 192 bytes long, but only 191 bytes store data; the first is the
marker. Buffer #1 is $027A - $0339 (634-825); buffer #2 is $033A - $03F9 (826 - 1017).

These buffers are used only by tape, *except* in BASIC 4 disk handling with CBM disks. Consequently, if disks and tape are both in use together, cassette #1 only should be used; two cassettes and BASIC 4 disks can be used provided the disks are not used while a file is open to cassette #2. Alternatively, the disks can be controlled by commands which avoid BASIC 4's special disk commands, concat, dopen, dclose,... Note also that both buffers are usable to store machine-code, provided that no tape activity overwrites them. For example, a BASIC routine which pokes machine-code into either buffer is fine, and can be loaded from tape. But machine-code in buffer #1 can't be saved to tape: the first thing that happens is that the buffer is replaced by the header details of addresses and name, which will delete any code in the buffer. Buffer 2, when BASIC 4 disk commands are used, is safe from $0381-$03E8 (897-1000). The limits of the buffers are automatically set by ROM routines depending on the device number. See F667/F656/F695 in BASIC 1/2/4.

Program headers have this structure

1	1	4	17	4	72	69	76	76	79	32	32	32	32		
i.d.	start		end		program name						spaces				
	$0401		$0411		H	E	L	L	O						

This header is for the program   10 PRINT"HELLO" , which was loaded from a tape in cassette #1. So PEEKing 634 -650 or so gives the decimal information listed. (Note that BASIC 1 starts at $0400). The first byte is $01 = 1.

Data headers   have a marker byte of 4:

4	122	2	58	3	68	65	84	65	32	32	32	32		
id.	start		end		file name					spaces				
	$027A		$033A		D	A	T	A						

This is the header for a file called 'data', which will load into cassette buffer #1. Note that the end address is recorded as $033A; in fact, addresses $027A - $0339 only are used by the file data as it is read from tape.

Data   is stored in buffers of this form:

2	72	69	76	76	79	13	0		
id.	H	E	L	L	O	CR.	Zero		

The marker byte is 2. Data is followed by carriage return. (It may also be separated by commas and colons; see INPUT in Chapter 5 for details). The zero-byte is the end-of-file marker, written when the file was CLOSEd. On detecting this, ST is set to 64. However, if ST is ignored, the zero byte is simply read past, and previous data ('garbage') will be read.

The End-of-tape header is a duplicate of the file's   header, but with id=5:

5	58	3	250	3	32	32	32	32	32		
	start		end		spaces						

This example follows an unnamed data file stored in cassette buffer #2.

Programs are stored in a single block of data; ST=4 or ST=8, 'short block' and 'long block' errors, happen if a program is read as data. They do not have a marker value; all the information needed to load them is held in the header.

In addition to CHR$(0), which causes ST=64 to be set, CHR$(10), linefeed, is treated as a special character: in fact it isn't written (as data) to a tape. There is a special routine to remove it. The absence of such a routine led to problems with the disk unit's files. CHR$(29), cursor right, also doesn't get through. The demonstration program shows these features. As far as INPUT# is concerned, though not GET#, many characters are anomalous: CHR$(13), CHR$(32), CHR$(34), CHR$(44), and CHR$(58), which are Return, space, quote, comma, and colon, for example, give strange effects. Leading spaces are deleted, for instance.

CMD enables programs to be stored as data; a data file is opened, CMD directs output to the file, then LIST saves the program as an ASCII file, so that PRINT is stored as 5 bytes. See 'MERGE' (Chapter 5) for details and examples.

Since OPEN 1 loads a program's header, its load address and save address can be found with PEEK(635) + 256*PEEK(636) and PEEK(637) + 256*PEEK(638). The name can also be peeked out; so can any machine-code which may have been written into

the buffer as a security device.

ROM routines and machine-code programming

OPEN, CLOSE, LOAD, and SAVE can be investigated by disassembling the kernel (the jump table almost at the end of ROM). Each of these commands has a jump address here. Then, at some later point, after taking in the command's parameters, a set of branches occurs:

```
LDA $D4 ;DEVICE NUMBER
BEQ xxxx;BRANCH IF KEYBOARD
CMP #03
BCC xxxx; TAPE ROUTINES
```

This, or something similar, is the routine at which IEEE devices are separated from cassettes. The address after BCC is the start of the tape processing. After this, the two possible tape device numbers are distinguished in various ad hoc ways, for example by decrementing and branching if equal to zero, which finds device #1. The tape timing mechanism is complicated; it involves both VIA timers, and also resets the interrupt vector from a table; there are three interrupts (apart from the usual keyboard servicing routine), which deal with writing the header, writing data, and reading tape respectively. The keyboard processing is cut off, so no keypress gets through to the keyboard buffer. But the stop key is tested by its own subroutine, so there is some control over the tape. A method like this is necessary to maintain accurate timing, since the keyboard processing routine doesn't take a constant time. The instruction DEC $813 is used to disable the normal interrupt, prior to resetting the interrupt vector; it has the effect of setting bit 0 to 0. $E813 = 59411, so the same trick can be performed from BASIC: POKE 59411,PEEK(59411)-1 turns off the interrupt, and with it the keyboard and stop key; it also speeds processing slightly. The interrupt must be turned on again if the keyboard is to be reactivated.

The table below lists some RAM locations, in the interface chips, which are relevant to tape. A few other locations (ROM and RAM) are included, where they are closely connected with cassette operating:

CASSETTE:	CASSETTE #1			CASSETTE #2		
ROM:	BASIC 1	BASIC 2	BASIC 4	BASIC 1	BASIC 2	BASIC 4
Motor: On	Bit 3 of $E813 (59411) off [Usual value $35 (53)]			Bit 3 of $E840 (59456) off [Usual value $CF (207)]		
Off	Bit 3 of $E813 (59411) on [Usual value $3D (61)]			Bit 3 of $E840 (59456) on [Usual value $DF (223)]		
Cassette status flag*	$207 (519)	$F9 (249)		$208 (520)	$FA (250)	
Both motors off	JSR FFED SYS 65517	JSR FCA6 SYS 64678	JSR FCEB SYS 64747		same	
Restore normal IRQ	JSR FCFB SYS 64763	JSR FC7B SYS 64635	JSR FCC0 SYS 64704		same	
Key Sense pressed[2]	Bit 4 of $E810 (59408) off			Bit 5 of $E810 (59408) off		
not pressed	Bit 4 of $E810 (59408) on			Bit 5 of $E810 (59408) on		
# Chrs. in tape buffer	$271 (625)	$BB (187)		$272 (626)	$BC (188)	
Buffer addresses	$027A – $0339 (634 – 825)			$033A – $03F9 (826 – 1017)		
Tape read interface	Bit 0 of $E811 (59409)			Bit 3 of $E84D (59469)		
Tape write interface	Bit 3 of $E840 (59456)			same		

*If non-zero, the contents of this location signal that a cassette key is pressed

[2]Reverse, Fast Forward, or Play

The tape motor(s) can be turned on and off from within BASIC with the help of data from this chart. A complicating factor is the IRQ service routine (E685/E62E/E455), which under normal circumstances turns off either or both motors if it finds them on. (The sequence LDA E810/ ASL/ ASL/ ASL tests the cassette sense line for both cassettes: if the carry flag is set, a button is not pressed on cassette #2, and if the minus flag is set, a button is not pressed on cassette #1. In either case the motor is turned off, provided the cassette status flag (asterisked in chart) is zero). To avoid the motor being turned off by the usual interrupt processing, poke a non-zero value into the status flag location for the cassette.

OPEN, CLOSE, LOAD, and SAVE can of course all be performed from machine-code. To see how the inbuilt routines might be used, let's consider some published examples of unorthodox use of the cassettes:

(i) A program can be loaded into addresses different from those on the header, by first loading the header alone, then changing its pointers and loading the remainder of the program. It could, for example, be watched loading into the screen; or a machine-code routine, saved on a machine with a lot of memory, might be loaded into a smaller machine in this way.

(ii) 'Append' of tape programs or subroutines can be achieved by loading two programs so that the second starts where the first ends.

(iii) A program can be saved along with data in its header. Normally, this can't be done, since SAVE clears the buffer, filling it with spaces, before putting the start and end addresses and program name in. If a program is saved with machine-code in its header, this acts as an anti-copying device up to a point, since SAVE in the usual way will erase this buffer, so the copied program won't run if the buffer's routine is essential to it.

(iv) Another anti-copying device is to change the header pointers so a program loads into the keyboard buffer. It can thus be made to RUN immediately on LOADing. Provided the stop key is disabled, the program is made relatively difficult to enter.

(v) Some programs for BASIC 1 were made 'uncopyable' by setting the header pointers to start at an address below the start of BASIC, causing SAVE not to work. 'Microchess' used such a principle; it also included its own save routine, so that an appropriate SYS call copied the program.

The table which follows should provide a useful map for those readers who wish to explore tape ROM. Some significant ROM routines and RAM locations concerned with loading and saving programs from/ to tape are listed. Roughly speaking, the lower-level subroutines are further toward the end of ROM, so the trickiest programs must use routines with higher ROM addresses.

FUNCTION/ LOCATION	BASIC 1	BASIC 2	BASIC 4
Tape LOAD (assumes parameters are set)	F3A5	F395	F3D4
Tape SAVE (assumes parameters are set)	F6F6	F703	F742
Save (i.e. write) header (LDA #1) or e-o-t (LDA #5)	F5AE	F5DA	F619
Save program or own header etc.*	F8C1	F88E	F8D3
Load (i.e. read) next header	F5AE	F5A6	F5E5
Load named header	F495	F494	F4D3
Load rest of program	F3C3	F3B9	F3F8
Load any data*	F88A	F85E	F8A3
Device number (1 or 2, stored as $01 or $02)	F1	D4	
Length of name (0 means no name assigned)	EE	D1	
Start address of name, if there is one	(F9)	(DA)	
Start address for load/ save	(F7)	(FB)	
End address -1 for load/save	(E5)	(C9)	
Load/ verify flag (0 = load, 1 = verify)	020B	9D	
Secondary address (0,1, or 2)	F0	D3	
Delay before writing	0279	C3	

*Start and end addresses, and other parameters, need to be set.

Examples. (i) Loading machine-code or BASIC into screen RAM: The easiest demonstration is to load the header, change its pointers, and load 1 or 2 K of the program. OPEN 1 reads the first header on tape #1. Locations $027A onwards (i.e. 634 onwards - try FOR J = 634 TO 654: PRINT PEEK(J);: NEXT for i.d. and addresses and ASCII values of the name) hold the header, so we poke the start address with $8000 and the end address with $8400 or $8800. POKE 635,0: POKE 636,128: POKE 637,0: POKE 638,132 or 136 works for any ROM. Then calling F3C3/F3B9/F3F8 completes the load, after taking these addresses form the buffer. So (depending on the ROM) SYS 62403/ 62393/ 62456 completes the load into screen RAM.

(ii) Saving machine-code from BASIC. RAM can be saved to tape as a named program file without entering the monitor. All that's needed is to poke all the relevant parameters into place, and call a routine which writes to tape. This is not entirely straightforward, because the X-register needs to be loaded with zero before SAVE is

called, and this can only be done in machine-code. If we save this machine code at the start of BASIC, we have a routine like this:

```
0 REM ""j 1v7"  SYS 1031 CALLS:  LDX #1  DEX  JMP $F6F6 or $F703 or $F742
1 REM BYTES IN 1031-1036 ARE 162,1,202,76, AND THE 2 JUMP BYTES (LOW-HIGH)
10000 REM * SUBROUTINE TO SAVE MEMORY IMAGE TO TAPE OR DISK FROM BASIC *
10010 REM * LIMITS ARE $0400-$8000, AS CASSETTE BUFFERS ARE USED, AND  *
10020 REM * MEMORY ABOVE $8000 ISN'T SAVED ON TAPE.                    *
10030 REM * INPUTS. B=BOTTOM OF MEMORY TO BE SAVED, T=TOP+1; N$=NAME;  *
10040 REM *
10050 POKE 251,B AND 255: POKE 252,B/256: REM BOTTOM ADDRESS INTO ($FB)
10060 POKE 201,T AND 255: POKE 202,T/256: REM TOP ADDRESS INTO (C9)
10070 FOR J = 0 TO LEN(N$)-1: POKE 826+J, ASC(MID$(N$,J+1)): NEXT: REM POKE NAME
                                                    INTO CASSETTE BUFFER #2
10080 POKE 212,1: POKE 209,LEN(N$):     REM DEVICE IN D4; NAME'S LENGTH IN D1
10090 POKE 218,58: POKE 219,3:          REM 033A = (DA) = START OF NAME
10100 SYS 1031: RETURN                : REM CALL SAVE ROUTINE
```

Note than line 0 has been used to store machine-code. Everything after REM is ignored so this is perfectly acceptable. However, there must be no zero in the line, or it will be treated as an end-of-line marker and generate a spurious line if the program is edited; hence the use of LDX #1/ DEX in place of LDX #0. The bytes remain intact unless line 0 is edited, in which case some may be changed – peek them to check. Quite long machine-code routines can be stored like this. Note that BASIC 1 has a different set of zero-page addresses, which can be found in the table on the previous page.

8.3 Miscellaneous: fast forward winding, directories of tapes, BASIC 1 bugs, security

Timing fast-forward tape movement CBM cassettes have no fast-seek facilities of the sort which are sometimes met with, for example in the 'stringy floppy' with loops of tape, or certain of Sharp's machines, which store a marker on track 2 of the tape, to give advance warning of the presence of a header. This perhaps makes no great difference; tape is inevitably clumsy compared with disk. Nevertheless it is possible to say a few useful things about fast-forwarding tape.*

In fast-forward mode, we can assume that the drive motor is rotating at a fixed speed, so – with apologies to those who don't know calculus – after time t seconds, the distance (ds) the tape moves in interval dt is proportional to the circumference of tape on the take-up spool x dt. This circumference = $2*pi*$(radius of spool + k*t), where k is a constant related to the speed of the motor and the tape thickness. So $s = k_1 \int k_2 + k_3*t \, dt = k_1 t + k_2 t^2$. In other words, we can expect, or at least hope, that a simple quadratic expression relates fast-forward time to distance along the tape.[2] The diagram illustrates the situation. In fact, this model does approach reality with sufficient precision to be useful. Note that the tape is predicted to advance faster towards the end, which of course it does.
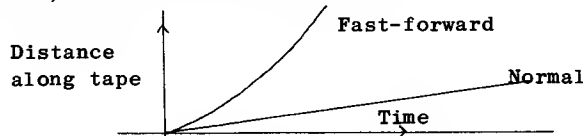
The point of this type of relationship is this: suppose we have a program on tape which is (say) 5 minutes' playing-time from the start. Can we estimate the fast-forward equivalent time? If we have a graph like the one sketched above, we can simply read off an estimated time; in view of the latitude allowed by leaders, this is usually good enough to find the program in a reasonably cost-effective way.

*Articles (empirical rather than theoretical) include N Thomas, IPUG Jan.'80 and Sept. '80 and W McCracken, CPUCN #7. 'Micro' has published articles on this topic.

[2] Readers who have followed me so far will be able to check that:

ds = circumference * revs.per sec. * dt, so that

ds=2*pi*(radius of spool + revs.per sec.*thickness*t) * revs.per sec. * dt

So Fast-forward distance in t_f secs. = $2*pi*rps*$(radius*t + rps*th*t^2/2).

Distance in normal operation = 1 7/8 * t_n inches. So the ratio of fast-forward to normal time required to cover some fixed distance of tape is implicit in

1 7/8 * t_n = $2*pi*rps*r*t_f + 2*pi*rps^2*th*t^2$/2. The constants can be guessed or measured to provide a quadratic equation; typically, to the first term, $t_f \doteq .008 * t_n$.

There are two ways in which this information can be applied. We can design a tape system so that a number of equal spaces are allocated on tape; in this way, we can fast-forward to any program, either to write or read it, fairly easily. Alternatively, we can write a directory program which reads any tape, printing the times taken to find each program, and perhaps predicting the corresponding fast-forward time. (The programs of McCracken and Thomas respectively illustrate these approaches). For a precise job, it's necessary to write test data to tape, then read it back after a timed fast-forward. A program to do this is probably of too limited interest to be included here. How can fast-forward timing be measured? We can turn off the motor after a predetermined time like this:

```
10 INPUT "NUMBER OF SECONDS FAST FORWARD";S
20 PRINT "PRESS FAST FORWARD KEY"
30 IF PEEK(59411) <> 53 GOTO 30   :REM AWAIT KEYPRESS
40 T = 60*S + TI
50 IF TI<T GOTO 50                :REM TIME LOOP FOR 60*S JIFFIES
60 POKE 249,1: POKE 59411,61      :REM STOP MOTOR (USES STATUS FLAG)
```

and in this way we can fast-forward to any point on tape.

Tape directories It is easy, and useful, to write a program to list the contents of a tape. It is true, however, that such a program will be slow, and isn't really a substitute for notetaking on the contents of tapes. The program below repeatedly loads headers, reporting the start/end addresses found, printing the name of the file, and reporting the time taken to read the tape at normal speed.

```
10 TØ=TI: OPEN 1: T1=TI: CLOSE 1 :REM READS HEADER; T1-TØ IS TIME TAKEN
20 PRINT "NAME:";:
30 FOR J = 639 TO 654: PRINT CHR$(PEEK(J));: NEXT : PRINT: REM 16 CHARACTERS
40 PRINT "START ADDRESS"; PEEK(635) + 256*PEEK(636)
50 PRINT "  END ADDRESS"; PEEK(637) + 256*PEEK(638)
60 T = T + (T1-TØ)/6Ø
70 PRINT "NORMAL SEARCH TIME"; T; "SECONDS"
80 GOTO 10                        :REM CAN INCLUDE END-OF-TAPE TEST, PEEK(634)=5
```

This simple program can be enlarged to report whether a file holds a program or data, for which PEEK(634) is 1 and 2 respectively. Hex addresses, likely BASIC programs, estimated fast-forward times, and the contents of headers which have other than spaces after the program name, are examples of the sort of thing which may be of use.

Bugs in BASIC 1's tape handling There are two serious bugs in the data file operation of BASIC 1, not the program loading and saving, which can be corrected by software kludges. (i) A write file opened from cold doesn't set the pointers to cassette buffer #1 or #2 as it should; poke them into (F3) with POKE 243,122: POKE 244,2 ($027A for cassette #1) or POKE 243,58: POKE 244,3   ($033A for cassette #2).

(ii) The interblock gap doesn't allow for motor start-up: the kludge for this is to start the motor before the buffer is full, e.g.

```
40 PRINT#1,Z$:REM WRITES DATA TO TAPE FILE #1 ON CASSETTE 1
50 IF PEEK(625)>160 THEN POKE 59411,61                :REM MOTOR ON
```

The value of the parameter in line 50 depends on the length of the strings being written to tape, and the frequency with which they are written. The maximum value in 625 is 192; allow enough time for the motor to run about 1/3rd second.

Miscellaneous (i) The '<' key. There is a programming error in BASICs 1 and 2 which makes this key appear to relate to cassette #1. (See locations $E6AB ff. in BASIC 2, for example). Because of this, if a key is sensed from cassette #1, '<' repeats if it is held down if for example the tape holds a recording. Sometimes '<' appears to become inactive; poking E811 (59409) so bit 7 is low may help, e.g. POKE 59409, PEEK (59409) AND 127.

(ii) SYS 62485/ 62493/ 62556 print the last-loaded or last read name to the screen, both for cassette #1 and cassette #2.   •

(iii) Security. Tapes are less copyproofable than disks, because ordinary audio copying can be used (and is, for commercial duplication of programs on tape). An interesting routine ('Auto-Run-Save') for BASIC 1 only by W Kolbe

in 'Micro' (Sept. '80) enables a BASIC program to be saved with a modified header so that it always runs immediately on being loaded. This, combined with disabling of the Stop key, provides considerable security against LISTing and amending of BASIC. The program ('Auto Run Saver') offers a further possibility of modifying the interrupt to test for direct mode at every interrupt, and calling the reset routine if direct mode is detected. This makes automatic running very secure against LISTing. (Something like: **LDA $78/ CMP #02/ BEQ RESET/ JMP IRQ** inserted before the normal interrupt detects immediate mode; in BASIC 1 the equivalent is **LDA $CA/BEQ RESET/JMP IRQ**). Unfortunately, while this process is relatively easy for BASIC 1, BASICs 2 and 4 have been reorganised in a way which makes auto running difficult to achieve. (It can be done with disks - see Chapter 6). This diagram shows why:

```
          -----HEADER (192 BYTES)-----        KEYBD. BUFFER PROG.  ----MAIN PROGRAM----
     ┌─┬──────────────┬──────┬─────────────┐  ┌─┬──────────────┐  ┌────────────────────────┐
     │ │Prog.  loads  │      │             │  │ │              │  │ BASIC or MACHINE-CODE  │
     │1│──────┬───────│ NAME │ MACHINE-CODE│  │7│ SYS655[RETN] │  │   with e.g. SYS 1037   │
     │ │from  │  to   │      │             │  │ │              │  │                        │
     └─┴──────┴───────┴──────┴─────────────┘  └─┴──────────────┘  └────────────────────────┘
     634            639    655
```
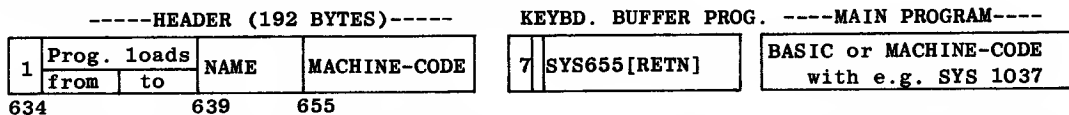
The program is saved in three parts, not the usual two with a header and program. The diagram is intended to be read from left to right; this is the order in which the three components are saved to tape and read back. The header has a start and end address pointing to $020D to about $0218; these 12 bytes include the keyboard buffer ($020F - $0218) and the location holding the number of characters in the buffer. When the header loads (into cassette #1 only), the short following program is loaded and direct mode is entered; but since the program deliberately fills the keyboard buffer, the operating system inputs the buffers contents, which are SYS 656 [Return]. The header has machine-code saved with it; this is called by the SYS command, and has two functions: (i) to put Run [Return] in the keyboard buffer, (ii) to load the next (i.e. ordinary BASIC) program using the header already in cassette buffer #1. So the program is loaded and immediately RUN.

BASICs 2 and 4 have the keyboard buffer in the same place as BASIC 1, but the location containing its contents has been moved to the zero-page. This means that the intermediate pseudo-program must load into a region which crosses the stack. Since the tape loading routine uses the stack, this is difficult or perhaps impossible to arrange. The only alternative seems to be to load a one-byte 'program' into the IRQ vector, ($90), in the zero-page, to temporarily deflect the interrupt into machine-code in cassette-buffer #1. This seems impossible with BASIC 2, because the interrupt points to the wrong part of memory, but it should be possible with BASIC 4.

To save a header and program with machine-code included in buffer #1 and with any load address and end address, use this routine, which can be called from BASIC or machine-code:

```
START LDA #01
      STA $D4    ;DEVICE #1 = TAPE #1
      LDA #7F
      STA $DA    ;POINTER TO START OF
      LDA #02    ;NAME, ASSUMED PRESENT
      STA $DB    ;IN BUFFER #1
      LDA #AA    ;LENGTH OF 'NAME', I.E.
      STA $D1    ;INCLUDING M/CODE
      LDA STLO
      STA $FB    ;LOAD ADDRESS TO BE
      LDA STHI   ;STORED IN HEADER
      STA $FC
      LDA ENDLO
      STA $C9    ;END ADDRESS TO BE
      LDA ENDHI  ;STORED IN HEADER
      STA $CA
      JSR $F656  ;SET TAPE #1 BUFFER [BASIC 4:$F695]
      JSR $F847  ;AWAIT PLAY & RECORD [BASIC 4:$F88C]
      LDA #01    ;HEADER TYPE = 1
      JSR $F5DA  ;WRITE HEADER         [BASIC 4:$F619]
LDPRG STORE POINTER TO START OF PROGRAM
      OR PSEUDO-PROGRAM IN ($FB)
      STORE POINTER TO END OF PROGRAM
      OR PSEUDO-PROGRAM IN ($C9)
      JMP $F71B; WRITE WITHOUT HEADER [BASIC 4:$F75A]
```

This routine writes a header, including program name and machine-code, using details assumed to have been assembled directly in cassette buffer #1; other locations of course are possible. It also writes the program which the header will load; a second program, to be loaded by the header's machine code, can be written by entering a second set of start/end addresses and writing the bytes without a header.

8.4 Printers

Printers in general   Printers serve several purposes: they enable permanent records to be kept on paper, for example of program 'listings', as they are invariably called. They enable data to be output in a more-or-less readable form, as 'printout'. This may include both finished output and audit trails. Finally, they can produce documents with features which mimic typed or printed output, for use in word processing, letter-writing, and so on. In principle, they are simple: often they are receive-only devices, which convert a limited range of bytes into characters. In practice there are several complications making this aim difficult to achieve. Some printers, usually the more expensive daisy-wheel type, are available with 'KSR' (keyboard send and receive) features, enabling messages typed at the printer to be received by the computer, but we need not consider this aspect in detail, since it is unlikely to be useful except for configurations of several remote computers. Let's first look at the current methods by which the actual impression is made on paper, before considering the questions of interfacing and of firmware.

In approximate order of expense, these are the printer types now available:

(i) <u>Teletypes</u>. These provide a paper terminal which can both send and receive data to a computer. They were widely used in computer installations, but have been largely superseded by VDUs. They are rather large, heavy, and noisy, and have upper-case text only; however, second-hand models can be got very cheaply. The interface is RS232.

(ii) <u>Thermal and spark printers</u>. Printers of these types require specially prepared paper, sensitive to heat in one case, and conductive in the other. Characters are made up of dots on the dot matrix principle. Thermal printers have a head containing elements which rapidly vary in temperature, causing dots to be plotted as the head moves across its paper. Spark printers use aluminised paper; a series of small high-voltage bursts burns dark marks on the paper. Printers like this are silent, but the paper is expensive, and usually available in narrow rolls only.

(iii) <u>Modified electric typewriters</u>. Reconditioned golfball typewriters with an interface to accept computer data have had some popularity before prices of dot-matrix printers dropped to competitive levels. They produce good-quality text, but the speed is limited by mechanical components driving the golfball.

(iv) <u>Dot-matrix printers</u>. These are by far the most widely-used printers with microcomputers. The print-head, made by a specialist manufacturer, has typically 7-9 wires arranged vertically, which are driven into contact with the ribbon and paper by solenoids, each wire having its own solenoid. The 4022 for example has 8 wires; as the head scans the paper, any of the 256 combinations of wires printing or not printing can be triggered, and characters are built from these fundamental dot patterns. Each 4022 character is 6 dots wide, so six separate sets of impacts make a character, unless some of the 'impacts' are of a blank column of dots. So, for example, a printer working at the rate of 100 characters per second, with an 8 by 6 character structure, makes 600 sets of impacts per second maximum. Printing at this maximum rate may cause problems of overheating; 4022 users are warned in the manual not to print much text in reverse characters.

(v) <u>Daisywheel printers</u>. A 'daisywheel' has 100 or so spokes (or 'petals'!) arranged radially around a thicker hub, each spoke terminating in a raised, reversed character. The wheels are made of light metal or plastic, designed with low rotational inertia so that they can be spun fast. Fortunately they are largely standardised, so that (for example) Qume and Diablo wheels run on each others' machines. Some wheels (e.g. Ricoh) have upper and lower case on the same spoke. Speeds of 50 or 60 ch.p.s. are quite common. The print quality is good. As is the case with golfballs, the common letters (e,t,a,i,o,n,s) are clustered near each other, to cut down on time spent moving the correct letter into place. 'Spinwriter' is similar, but uses a 'thimble'.

Features of printers   A few words on stationery, ribbons, switch-selectable printer features, and maintenance are necessary here.

(i) <u>Paper drive mechanisms and stationery</u>. Computer printers normally use continuous fan-fold stationery driven by sprocketed rollers. 'Pinfeed' or 'sprocket feed' usually implies that the roller has sprockets of a fixed separation; 'tractor feed' implies variable width, the 'tractors' being able to slide along the roller, and, in some designs, spreading the load over several perforations with a caterpillar-track arrangement. 'Pinchfeed' permits some printers to use unperforated stationery, e.g telex paper. Single sheets can be fed, one at a time, using 'cut sheet feeders'; these are

optional extras (usually for daisywheel printers) and are expensive. Computer station-ery can of course be obtained in multi-thickness form; printers vary in their capacity to handle copies.

(ii) Ribbons. Many printers have cartridge ribbons; the cloth type is arranged as an endless loop, held within the cartridge either loose, packed in a random pattern so the direction is constant. Also the typewriter-style spools which reverse direction are sometimes met with; these are cheaper than cartridges, but there is a risk of dam-age with some types of ribbon, for example through clogging the print head (with the wrong type of ink) or bending the wires (with eyelets in the ribbon). Carbon or film ribbons are used for high quality impressions with daisywheel and golfball printers. Multi-strike ribbons are more economical and give a slightly inferior appearance. Such ribbons offer once-only use, and it may be important to ascertain how much work a single ribbon can produce, since otherwise a program run may require more attention than ought to be necessary, at a greater cost.

(iii) Features. External switches on printers - apart from on/off! - may include paper control (paper feed, set top-of-form, move to top-of-form) and/or automatic linefeed on/off, at the simpler levels, up to a full range of facilities, controlling baud rate, parity, horizontal and vertical spacing, margins, tabs, etc. Internal switches, accessible only by removing the lid, may be used to set characteristics like the baud rate, the type of interface, and (e.g. with Centronics printers) the type font suited to national needs, permitting currency symbols, diacritical marks, and special charac-ters (Dutch ÿ, German ß) to be printed. Many printers have some form of self-check or 'internal diagnostic' routine; Commodore's smaller printers for example have two channels available on reset, so switching on with the paper feed button pressed causes a jump to be made to a subroutine which repeatedly prints out the character-set.

The speed of a printer is usually quoted in characters per second or lines per minute. Neither measure is completely satisfactory. A 50 ch.p.s. daisywheel printer may produce a fairly sparsely-filled document more rapidly than a matrix printer rated at twice the speed, by skipping blank spaces instead of covering them at the same speed as the text. A bidirectional matrix printer is likely to be faster than a similarly rated unidirectional printer, because it need not waste time returning to the leftmost margin after every line. A printer with a large buffer may take less computer time to print, since the buffer (RAM held within the printer) may be able to accept larger batches of data for printing before spending time handshaking, waiting to take in the next batch. Some 'intelligent' printers move to the next line when they detect that the rest of a line is blank; 'Lines per minute' is a useful measure only when this doesn't happen, and even then there may be variability if the lines' total length can be con-trolled.

Maintenance is generally a dealer function; some machines may have to be deliv-ered by the user, particularly if they are cheap, even if there is a maintenance contract. It is worthwhile estimating the probable amount of use of a machine; if (say) an average page has 60 lines of 50 characters, a box of 1000 fanfolded sheets takes 3 million characters.

Printer features which are relevant to operating convenience include noise, port-ability, ease of paper loading, and, with some models, the choice between a free-standing machine or the desk-top equivalent.

8.5   Commodore and CBM-plug-compatible printers

2000, 3000 and 4000 series   These printers are Commodore's standard low-cost range. The 2022 and 2023 are, or were, 80 column printers, the first having 'tractor feed', i.e. sprocket drives of adjustable separation and the second, cheaper, model pinch feed. These were renamed the 3022 and 3023 to coincide with the 3000 series CBM computers. The 4022, with tractor feed, superseded these at about the start of 1981. There is no 4023. It differs from its predecessors, and in fact is closely related to the Epson MX-70. The firmware in these printers has not been completely successful. Two sets of ROMs (set 3 and set 4) have been issued, and others have been tested but not issued. Apart from minor bugs, the principal error is in the handling of lower-case lettering, which has to be done in a way not compatible with output to the screen or to other printers. Future ROM issues will have to be designed on the basis of the difficult decision of making the printers easy to use, but incompatible with existing CBM printer software, or to retain the previous weaknesses.

The 4022 dot-matrix printer   This printer is wired as IEEE device #4. It is controlled by a 6502. A red LED gives evidence that a channel is open to receive data. The dot matrix is 8 by 6; most characters are printed within a 7 by 6 rectangle, with descenders and the lowest line of reversed characters occupying the bottom row. The printer is controlled in two ways: firstly, a range of secondary addresses enables semi-permanent aspects of a printout (lines per page, spacing between lines etc.) to be set; and characters similar to the screen-editing VDU characters enable the second set of more temporary features to be controlled. These include reversed characters and multiple-length, 'enhanced', characters. The 4022 has 11 secondary addresses; earlier models had 7. We shall first look at these secondary addresses. I have assumed that a channel to the printer has been opened with OPEN 4,4. Other channels, which assign a file number to a secondary address, also need to be opened, and usually it is easiest to number the file equal to the secondary address, for example OPEN 6,4,6. A few points are worth noting: PRINT# has to be used as a rule, because CMD followed by PRINT sometimes fails to work (e.g. after GOSUB). Some parameters have to be entered as CHR$(), although this is not mentioned in the manual. If a command seems not to be working, try the combinations of (say) "60", CHR$(60), and 60 until you find the correct formulation. And the special features of output formatting and of user-definition of a character only apply to a single format string and a single character at one time. So a table, in which each line resembles the previous line's layout, is straightforward to print; whereas interleaved lines of different format require that the format string be redefined within the printer. Finally, beware of plugging the printer into the PET/CBM with its plug upside-down, which may be possible if the polarising pins in the plug work loose; this will damage the computer.

4022 Secondary address	Function	Notes
0	Print 'as received'	This is the default option (no secondary address) Tab, Clear, etc. don't work; hangs on back-space, CHR$(20).
1	Print in format	Prints according to the format last printed to secondary address #2. Overflow (or other error) resets secondary address to 0, fills the field with warning asterisks, and prints out the type of error if this is enabled by sec. address #4.
2	Define format	A single format can be defined at one time in COBOL-like form, e.g. S$$$$9.99 causes 12.345 printed to secondary address 1 to appear as:- + $12.34
3	Set lines per page	This sets the number of lines which are printed before six blank lines automatically print (to move past the perforations). CHR$(147) turns on paging; CHR$(19) turns it off.
4	Enable diagnostics	OPEN 40,4,4:PRINT#40:CLOSE40 causes diagnostic messages to appear on errors. There are six messages, consisting of a single letter prefixed by '*PE:'. ('Printer Error' presumably).
5	Define character	Define own CHR$(254). One only at a time. Can't be changed during a line; this gives a 'Terminator error'. *Can* print several on one line using CHR$(141) as Return without linefeed.
6	Set vertical spacing	OPEN 6,4,6: PRINT#6,CHR$(N): CLOSE 6 sets the line separation to 144/N inches. So CHR$(18) prints 8 lines/inch, keeping the characters their usual height, so there is no separation.
7	Upper case	OPEN 7,4,7: PRINT#7: CLOSE 7
8	Lower case (in part)	OPEN 8,4,8: PRINT#8: CLOSE 8
9	Disable diagnostics	OPEN 9,4,9: PRINT#9: CLOSE 9
10	Reset	Reset all the semi-permanent features set via the secondary address to the values obtaining on switch-on, with OPEN 10,4,10:PRINT#10:CLOSE 10.

The most significant aspect of printing in formats is that numbers are easier to deal with than would otherwise be the case. 23 and 1234.567 can be converted instantly to 23.00 and 1234.56 with these printers. Unfortunately the formatting process deals only in whole lines, so it often happens that text is mixed with numerals, and text is often easy to align without the need for a format definition. In other words, when printing a mixed output of text and numerals, the textual part may well be more tiresome to arrange in formatted form than it would have been to print out directly. The full details of formatting, with examples, would take too much space here, but the short example which follows illustrates how a literal, a string, and a numeral can all be simultaneously formatted.

The object is to print a single line, given a name N\$ and a sum of money D (in dollars - the only currency symbol available when formatting). When a set of lines are printed, they are to appear like this:

```
PAY ERROL T. ZINZINHEIMER       $45.67
PAY J. DIBBINS                  $7.50
```

that is, the word 'PAY' followed by the left-justified name N\$ and finally D, formatted to 2 decimal places and preceded by '\$'. The following program rounds D to the nearest half-cent as well:

```
100 OPEN 1,4,1: OPEN 2,4,2
110 PRINT#2,"[RVS]P[RVS]A[RVS]Y AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA $$$$$.99"
120 INPUT N$,D: REM NAME AND AMOUNT
130 PRINT#1,N$ CHR$(29) D+.0045
140 GOTO 120
```

Line 120 causes a format to be stored in the printer's RAM. Note that a reverse character signals that the next character (only) is to be treated as a literal: hence line 130 prints 'PAY' immediately, followed by N\$ left-justified into the alphabetic field. The cursor-right or 'skip', CHR\$(29), forces an end to the field, so the numeral with the leading '\$' and numerals after the decimal point prints next. The maximum value is 9999.99 in the example; if secondary address #4 is active, D larger than this will generate a brief and modest error-message, which the programmer can pretend has some esoteric function. Even if D is zero, the decimal point appears: \$.00 and this keeps the appearance tidy. There is no easy way to retain a leading zero for values less than 1; .55 is easier to print than 0.55, although many people prefer this latter form.
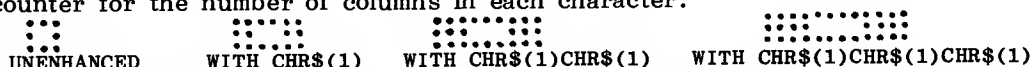
Error messages. The 4022 has 6 diagnostic error messages, which are printed if secondary address #4 has been enabled. These are:

    PE:C ... Secondary address exceeds 10.
    PE:E ... Exponent error; number in scientific format requires E±xx.
    PE:F ... Format sent to secondary address #2 was invalid.
    PE:L ... Lines per page, sent to secondary address #3, were out of range.
    PE:M ... Mismatch - alphabetic data sent to secondary address #1 numeric field.
    PE:T ... Secondary address changed before Return or Linefeed send.

4022 printer control characters. These characters are printed to the channel with secondary address zero, either in quotes or in the form CHR\$(x). They are exactly analogous to screen editing characters, such as cursor right or Clear, which may be 'printed' to the screen. In fact some screen editing characters are also printer control characters, affecting the printer differently from the screen. Because of this inconsistency, it it usually not possible to print the same output either to the screen or the printer, unless all characters are upper-case.

n	chr$(n)		chr$(n + 128)
1	chr$(1)	Enhanced printing	chr$(129) [or Return] Unenhanced printing
10	chr$(10)	Linefeed	
13	chr$(13)	Carriage return	chr$(141) Carriage return without linefeed
17	chr$(17)	Lower case	chr$(145) Upper case
18	chr$(18)	Reverse printing	chr$(146) Reverse off
19	chr$(19)	Top of form	chr$(147) Set top of form
29	chr$(29)	Skip space	
32	chr$(32)	Space	chr$(160) Shift-space; not a leading space
34	chr$(34)	Quote	
			chr$(254) User-defined special character

Enhanced Printing Most dot-matrix printers have this feature, which is quite easy to implement (in contrast to daisywheel printers, for example, where the best approximation to this facility is to overprint **like this**). In Commodore's version, CHR$(1) causes subsequent characters to be printed with an additional column of dots; carriage return or CHR$(129) turns off this feature. This is *not* a doubling of character width for each CHR$(1), as it is often described. The diagram schematically shows what happens. Presumably, the printer's RAM is loaded with a constant, which is decremented, acting as a counter for the number of columns in each character.

UNENHANCED   WITH CHR$(1)   WITH CHR$(1)CHR$(1)   WITH CHR$(1)CHR$(1)CHR$(1)

Remember that enhanced characters will fill a line with fewer characters than are required with normal-width characters.

User-defined character, chr$(254) The single programmable character available to users of this printer is defined by printing a six-character string into a file opened to secondary address #5. Subsequently, PRINT#4,CHR$(254) prints out the character, and PRINT#4,CHR$(1)CHR$(254) prints it double width, and so on. The character apparently is confined to 7 by 6 dots, so that descenders are impossible to get. The columns of dots correspond straightforwardly to the bit patterns:

	64	32	16	8	4	2	1

So the pattern of dots illustrated is printed by the string CHR$(2)+CHR$(65)+CHR$(73)+CHR$(90)+ CHR$(9)+CHR$(0). The example program below defines a character (made up of CHR$(65)s), puts this in secondary address #5, and prints the result as CHR$(254). This gives a rather unexciting pair of vertical lines, one row at the 64 (top) level, the other representing 1.

TOTAL: 2   65   73   90   9   0

SPECIAL PRINTER CHARACTER-

```
10 OPEN 4,4          : REM PRINTER CHANNEL
20 OPEN 5,4,5        : REM SPECIAL CHANNEL TO PRINTER, INTO WHICH ..
30 PRINT#5, "AAAAAA" : REM .. WE PRINT A 6-CHARACTER STRING
40 PRINT#4, CHR$(254): REM PRINT THE SPECIAL CHARACTER
50 CLOSE 4: CLOSE 5
```

In the U.K., the '£' sign is useful. One example is CHR$(1)+CHR$(13)+"?MM!" or you may prefer CHR$(9)+"?IIA!". Try these strings in line 30 of the demonstration program, replacing "AAAAAA", to get the feel of the procedure.

Lower-case printing This is a problem. The designers of the system don't seem to have realised the considerable similarities between 'lower-case' and 'graphics' modes on the screen (most characters, except alphabetics, remain the same in each mode). In place of a simple switch, similar to that caused by POKE 59468 with 12 or 14, and perhaps an optional set of routines to take account of BASIC 1's oddities, each lower-case line has to be prefaced by a cursor down character. Without the cursor-down, even in lower-case mode the printer produces this sort of thing:

1. ◆ET UP STOCK FILE - ADD NEW RECORDS   ONTO END OF FILE

So that in order to produce a printout like this extract:

```
Sales: Ø          Tax: 4          Foreign: 0
Hazard:           Origin: 8       Distribution: M
Status: #         Spare1:         Spare2:
```

The following print statements are necessary:

```
1080 PRINT"                                              "
1085 PRINT"    ◆ALES:    "J$(12)"      AX: "J$(13)"     _OREIGN: "J$(14)"
1090 PRINT"    IAZARD: "J$(15)"  RIGIN: "J$(16)" ISTRIBUTION: "J$(17)"
1095 PRINT"    ◆TATUS: "J$(18)"  ◆PARE1: "J$(19)"     ◆PARE2: "J$(20)"
1100 PRINT"                                              "
1105 PRINT"  ◆UPPLIER CODE: "J$(21)"                    "
```

The machine-code routine which follows is a lower-case LISTer. It is written for BAS-IC 2 ROMs. The alternatives are BASIC 4 equivalents. The routine is entirely relocatable, *except* for the underlined 2-byte address, which must be reset if a relocated version is to work.*

```
.: 033A  A9 00 85 11 85 12 20 (2C)   A3
.: 0342  (C5) 68 68 A0 01 84 09 B1    B5
.: 034A  5C F0 46 20 E1 FF 20 (E2)   DF   Note: may need POKE 16,123 for line-feed.
.: 0352  (C9) C8 B1 5C AA C8 B1 5C    BA
.: 035A  C9 FF D0 04 E0 FF F0 31
.: 0362  84 46 20 (D9 DC) A9 11 20    83 CF
.: 036A  (45 CA) A9 20 A4 46 29 FF    46 BB
.: 0372  20 C2 03 C9 22 D0 06 A5      Address (03C2) to be changed on relocation.
.: 037A  09 49 FF 85 09 C8 F0 11
.: 0382  B1 5C D0 10 A8 B1 5C AA
.: 038A  C8 B1 5C 86 5C 85 5D D0
.: 0392  B2 4C (89 C3) 10 DA C9 FF    B3 FF
.: 039A  F0 D6 24 09 30 D2 38 E9
.: 03A2  7F AA 84 46 A0 FF CA F0
.: 03AA  08 C8 B9 (92 C0) 10 FA 30    B2 B0
.: 03B2  F5 C8 B9 92 C0 30 05 20
.: 03BA  (45 CA) D0 F5 49 80 D0 AC    46 BB
.: 03C2  48 09 C0 C9 DB 90 08 C9
.: 03CA  E0 10 04 68 49 80 48 68
.: 03D2  4C (45 CA)                    46 BB
```

Other Commodore printers   Commodore offer two other printers, at the time of writing; a heavy-duty dot-matrix machine, and a daisywheel printer. Both seem to have been produced in conjunction with other printer manufacturers, and are less distinctively CBM than might appear at first sight. The 8024 is a 9 by 7 dot-matrix printer with a head speed of 160 ch.p.s and capable of producing multiple copies. It prints standard ASCII, i.e. no PET/CBM graphics. Or so at least the (rather scant) documentation says; presumably, since true ASCII has upper and lower case different with respect to CBM, there is some facility for mixed lower- and upper- case printing. Its ST differs from that of other CBM printers; it can accept 3 secondary addresses.

The daisywheel printer is a modified Olympia electronic daisywheel typewriter. The 8026 is the keyboardless receive-only version; the 8027 has a keyboard. The cost is similar to the 8024. The maximum speed of the unit (16 ch.p.s) is, by printer standards, very slow; a page like this one (without the type-face changes!) might take five minutes or more.[2]

The 8026 and 8027 have many features found on more expensive machines, but as might be reasonably expected, these are typically not so easy to use. Line feed, form feed and tabbing, for example, are all relatively awkward. Moreover, several sets of ROMs have already appeared, which are (in small ways) incompatible with each other. Some ROMs lack variable line separation. The control commands are similar to those of many printers, using ASCII control characters of CHR$(10) and CHR$(13) for linefeed and return, CHR$(7) for BEL, and in particular the escape character, CHR$(27), followed by a whole set of possible parameters. These include horizontal spacing (10,12, or 15 ch.p.inch), tabs, direction of printing, and allowance for the type of printwheel.

CBM 'plug-compatible' printers   Because of Commodore's major market position in the U.K. it is not surprising that manufacturers have produced printers which plug into the computer without an external interface box. For example, Anadex have done this, and there are a few very cheap Japanese printers. The Epson MX-80 (though not in

---

*This routine, published in 'Micro', is the work of Jim Strasma and is printed here with his permission. (The BASIC 4 amendments are straightforward ROM address changes; I have not tested the result). Jim Strasma is active in the Central Illinois PET Users' Group, which publishes the 'Midnite Software Gazette', a quarterly review. The third edition of Osborne/McGraw-Hill's 'PET Guide' is edited by Jim and his wife.

[2] CCN, Sept.'81, gives some information on Commodore's daisywheel printers.

this 'very cheap' category) is widely preferred to CBM's printers. These machines, many of which were sold when CBM printers were subject to delivery delays, are often non-Commodore in fairly subtle ways; they may be advertised as having graphics, but the character-set may be non-CBM; the handling of print statements and formatting is very likely to differ from Commodore's, since mimicry of the entire range of secondary addressing and other features would involve a lot of work. Often this makes no difference; only when a program is run on a new machine (i.e. one which uses different conventions) will any problems arise.

Non-CBM and non-plug-compatible printers   Many of the major names in printers make machines which cannot be directly connected to PETs; Qume, Centronics, Diablo, Spinwriter are examples. Centronics printers have their own standard; other standards include the 'current loop' and the 'RS232', a serial interface which accordingly needs few wires. These need 'interface boxes', devices which plug into the computer and also into the printer. An interface box may need its own power supply. There are a few hazards to watch for. The most important, yet again, is the treatment of upper and lower case text. The switch between modes within the computer cannot generally be detected by an interface box, so that, unless this is fitted with a switch, you may find that only upper-case text, say, can be easily printed. Probably some sort of conversion routine will be provided, perhaps a painfully slow one in BASIC. This is probably something most users could do without. It is occasionally true that an interface, because of a design oversight, won't transmit certain characters, such as 'Escape'.

As an example of the methods of programming such printers, which rely on ASCII control characters rather than secondary addressing, look at the following short BASIC program extracts. They apply respectively to a Qume and a Centronics printer. There is no standardisation in the characters following 'Escape'; the specimen printout shows a Qume printing data sent from a program designed for a Centronics machine. Note that an appendix has a complete list of ASCII characters, mnemonics, and their meanings.

```
1000 OPEN 4,4: PRINT#4,CHR$(12);:      REM QUME.  FORM-FEED...
1010 PRINT#4,CHR$(27)"E10"CHR$(27)"L04";: REM HORIZONTAL SPACING 10/120 = 1/12
                                          INCH; VERT.SEPN.=4/48 = 1/12 INCH


4100 OPEN 4,4: CMD 4              : REM CENTRONICS
4110 PRINT CHR$(27)CHR$(20)CHR$(27)CHR$(15): REM COMPRESSED AND ELONGATED CHRS!
```

```
BX 6 J ⊞ f r                    E      4.7U    0          , 3   04

BE 0 N ⊞ Cro d A u    2 2      E      14U    0          1     0

BE 0 04 ⑤ e e t l w    0 2      E      2 43 U    0          7     0
```

Printers: a Summary   Users looking for good-quality printing, without the obviously computer-produced appearance of dot-matrix printers, have little other choice than a daisywheel or modified golfball printer. If there is a definite need for speed, and many copies of an original, or 132 columns per page, then a heavy-duty printer will be necessary in the first cases, and a wide platen or programmable narrow characters (e.g. 16.7 ch.p.inch) in the second. Paper width(s) and type may be restricting factors, since many machines aren't versatile in their paper-handling. If an interface box is necessary, be sure a good one is available. It may be necessary to ensure that a printer can operate with several different types of computer. Because of the chance of unexpected programming difficulties, it is advisable to test any combination of hardware which is new to you; mixed upper- and lower-case text, and graphics if these are important, are likely to be problem areas. In this way, with luck, a good, fast system can be put together, in which fundamental aims of the system have not been overlooked. Many users, of course, simply buy CBM equipment. Current market surveys may (or may not) suggest better buys. Many such surveys though are not very thorough, and are little more than assemblages of information provided by advertisers. In practice, I suppose dealer advice and exchange of recommendations between friends and colleagues are about equally important forces determining final choice of hardware. There is one other, perhaps obvious, point: be sure that paper, ribbons, spare fuses, extra daisywheels and so on aren't too inaccessible, and that maintenance is available if it's needed.

8.6   The Modem.

Commodore's Modem (the '8010') is wired as IEEE device #5, and can be read/ written to with the normal INPUT#/ PRINT# commands to logical files with secondary address 5. The function of a modem ('modulator-demodulator') is to process ordinary data as it is output from a computer into a form suitable for transmission to another modem, which reconverts it into data to be input by a second computer. In this way data or programs can be transmitted across country or between continents. Typically, ASCII data is formatted with start bits , a parity bit, and framing by a UART or USRT type chip, sent out, and decoded by a similar chip at the other end. High speeds of data transfer are *possible*. The 8010 is an 'acoustically coupled' modem, using modulated sound sent by normal telephone lines. (It has Post Office approval in the U.K.). A telephone handset is used in conjunction with the 8010.

Software - programs and development aids - can be received in this way; since thay can also be sent through the post, this may not be very advantageous. However data can also be received from and sent to other computers; if the CBM is to be used to process a subset of a company's data, this may be very useful, since data which might otherwise need to be keyed in can be processed by program automatically.

Several points related to the actual operation of the modem should be borne in mind. Its rated speed is 300 baud, presumably meaning an upper limit of about 30 bytes per second. At this rate, a 40-column screen will take about 30 seconds to fill, and an 80-column screen 1 minute. This may be too slow for some purposes. There are likely to be difficulties over the transmission of mixed upper- and lower-case CBM alphabetic characters, since these are not consistent with the ASCII set. Finally, some software is needed to process the transmitted data, and moreover hardware interfacing may be needed to correct disparities of convention between the computers. Some commercial products are available incorporating one or both of these facilities, for example to read and store Prestel pages.*

8.7   The keyboard.

Introduction and physical description   The keyboard is not a peripheral in quite the same sense as the other items of hardware discussed in this chapter, because of its intimate connection with the computer. It resembles the screen in this respect. The next chapter is concerned largely with programming the CBM's screen, but there is an overlap because of the echo of keyboard to screen which usually happens except with GET commands.

All Commodore keyboards are equipped to handle most ASCII characters, with the exception of the control characters (not needed) and the more obscure punctuation symbols (curly brackets, underlining, single quotes sloping backwards). Many have graphics characters, but in the 'business machines' the decoding is different, and these are excluded to some extent. The keyboards have evolved in several stages, from the early tiny rectangular array of keys to the typewriter-style keyboard, with a numeric keypad, and with punctuation characters obtained by shifting (e.g. ! is shift-1 and so on). All include screen editing keys and the stop/run key, which are specifically CBM functions. TAB, ESC, and repeat occur on the 12 inch models only. There are some minor differences; some 3000 series machines' keys are marked with very wide characters for instance.

The keyboards are very reliable; sometimes an old one may benefit from being taken out and cleaned. They are not quite free from software bugs: the '<' key has been mentioned already; in BASIC 4, the 8032's right shift with reverse and C,B,>, and 3 repeats the wrong character.

In 8 inch models, RVS slows the screen scroll (½ second delay before scrolling).

In 12 inch models, the left arrow slows scrolling, and * or : cause scrolling to pause indefinitely, until one of a range of other keys is pressed.

There are a few incompatibilities that may be a source of difficulty; for example, the Run key on the 8032 is in the same position (top right) as Clear on earlier keyboards, so it is rather easy to load and run a disk program instead of clearing the screen if a user is accustomed to the early layout. This of course will erase any BASIC program in memory.

Commodore keyboards have no inbuilt reset key; this has drawbacks, notably when developing machine-code. If a crash results, the machine has to be switched off

---

*CCN Aug. '81 has an article by P Barker on microcomputers as terminals which includes some explanatory detail and programs relevant to the 8010. CCN of Oct. '81 has a 16 page centre section and some useful programs.

and reloaded - if the programs were saved, that is. On the other hand it prevents a program being wrecked in mid-run, something which is possible on (say) the Apple II. (VIC has a reset, using the NMI line). See the next section (8.9) on methods of constructing reset keys for PET/CBMs.

Physically, the keys are made of light plastic, surmounted by a grey or black key bearing the legend. The tops of the keys can be levered off and replaced; there is a feeble-seeming spring inside each one which prevents the contact at the base being made until it is pushed. As we shall see, we can redefine the keyboard decoding function, so that non-standard keyboards, such as the 'Dvorak' non-QWERTY layout can be tried out.

A reliable way to disable the Stop key, but still keep it usable for emergencies, is to use a guard over the key. The diagram (plotted by a PET!) shows the general app-earance and dimensions of such a guard. (The figures may not apply to all keyboards). A rectangle of stainless steel, and access to metalworking facilities, are necessary prelim-inaries. The hole in the top allows the key to be pressed with a pencil or biro. I have found this system completely reliable.
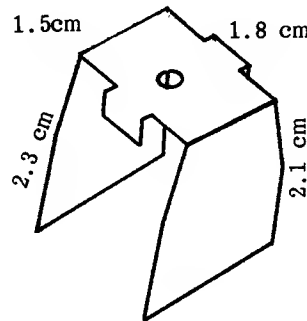
BASIC and the keyboard On the subject of disabling the Stop key, the most well-known method is to use a simple poke:

```
POKE 537,139  [BASIC 1]
POKE 144,49   [BASIC 2]
POKE 144,88   [BASIC 4]
```

BASICs 2 and 4 can be simultaneously covered with POKE 144,PEEK(144)+3. This popular method has the drawback of turning off the clock, for reasons we shall see. Also, any operation changing the interrupt vector (tape operations for example) is likely to re-store the normal interrupt with its test for Stop. Poke the identical value minus 3 in each case to return to normal operation.

The keyboard is treated as device #0 by the operating system; then come the cassettes and the screen (#3), before proceeding to the IEEE bus. We can open a file to the keyboard:

```
OPEN 1,0    :REM OPEN 1,0,55,"XYZ" IS SYNTACTICALLY OK, BUT HAS NO EXTRA EFFECT
```

and now, in program mode, we can INPUT#1 and GET#1 from this file. (PRINT#1 gives an error message). This can be used to give rudimentary input protection; if Return is pressed with no other entry, INPUT#1 returns CHR$(0). Commas and other punctu-ation separators won't now print ?EXTRA IGNORED, because a file is considered to be open.

This same (input protection) effect can be more easily got by a simple poke,

```
POKE 3,X [BASIC 1]    POKE 14,X [BASIC 2]    POKE 16,X  [BASIC 4]
```

where X is any non-zero value. Surprisingly, this method has never yet been docu-mented, so far as I know.

GET is the BASIC function which takes characters directly from the keyboard. As we have seen in chapter 4, this is a relatively easy way to modify keyed input, to avoid the limitations of INPUT and to code keys for some purpose, without echoing the character to the screen, as in this simple example:

```
10 GET X$: IF X$="" GOTO 10   :REM GET KEYBOARD CHARACTER (WAIT FOR KEYPRESS)
20 IF X$>="A" AND X$<="Z" THEN PRINT "ALPHABETIC!"
30 IF X$>="0" AND X$<="9" THEN PRINT "NUMERIC!"
40 IF ASC(X$)=13 THEN PRINT "[CLEAR]" :REM DEFINE RETURN KEY TO CLEAR SCREEN
50 PRINT "OTHER": GOTO 10:REM IGNORING SHIFT CHARACTERS, TO SIMPLIFY PROGRAM!
```

BASIC 1 and 2 have a feature (seemingly carried over from non-PET BASIC by Micro-soft) in which input of CHR$(15), 'Shift In' in ASCII, causes no output to appear on the screen. In fact, this character doesn't exist on CBM keyboards, but the suppress-ion feature remains: POKE 100,X [BASIC 1] and POKE 13,X [BASIC 2] where X exceeds 127 prevents PRINT from operating.

There are differences between (true) ASCII and PET/CBM's somewhat modified version, but these are only important when communicating between machines. A table

of true ASCII characters is printed in the appendices.

ROM: how the keyboard is scanned When a PET/CBM is operating normally, its program is interrupted at regular intervals and a subsidiary routine performed; this processes the cursor, turns off one or both cassette motors if they are on, and scans the keyboard for a keypress. If a new key is pressed, it is added to the keyboard buffer, unless this is full. If it is, BASIC 4 ignores the key, BASICs 1 & 2 cancel the buffer and start over. Let's examine this process in greater depth.

Firstly, how many interrupts occur per second? We can time them approximately with BASIC and a 6502 subroutine: Type SYS 4 to enter monitor; now type .M 027A 027A .: 027A E6 00 4C xx xx where the unknowns are the IRQ low and high bytes respectively as they appeared on entry to the monitor. Overwrite IRQ with 027A and enter .G 0004, which causes the interrupt to execute the short routine; it increments location $00 with each interrupt. Type .X and enter the BASIC program 0 IF TI-T<60GOTO and 1PRINTPEEK(0):T=TI:POKE0,0:GOTO which is written to run as fast as possible. The contents of $00 after 1 second are printed out. The value is about 60 for 8-inch screen machines, 50 for 12-inch screen machines.

When an interrupt is generated, the 6502 finishes its current instruction, saves values on the stack, and jumps to the address in ($FFFE), if the interrupt is not masked. This address is E66B, E61B, and E442 in BASICs 1,2, & 4. If these addresses are disassembled from, it is clear that A,X, and Y are all saved, for later recovery; this means that the interrupt program is entirely separate from the normal program. Moreover, the status register is examined to test for a BRK, signalled by the break flag, or a standard interrupt. In the former case, the monitor is entered (except in BASIC 1), which is why SYS 4 or SYS 1024 or SYS of any location containing zero causes a break entry (signalled by *B) to monitor. The interrupt jumps to an address held in RAM as two bytes; this address can be changed, so the programmer can write new interrupt servicing routines (like the small-scale example above which increments $00 at each interrupt).

($90) holds the address of the interrupt servicing routine; this is the address printed under IRQ when .R is entered in the monitor. This is ($0219) in BASIC 1. Thus fifty or sixty times per second the interrupt takes place and the code pointed to by this address is executed. This code - at E685,E62E, and E455 in BASICs 1,2, & 4 - is of some length, and takes up a measurable time, perhaps 7% of the total, to perform. This time can be saved by temporarily turning off the interrupt.

There are four parts to the interrupt servicing, which those interested can see by disassembling the appropriate code and examining it. The parts are:

(i) Update the clock (TI and TI$ locations) and check the Stop key. FFEA is a kernel jump command which carries this out. BASIC 4 (12-inch screen) includes a patch which increments the clock an extra 'jiffy' every 5 interrupts, so the timekeeping is irregular. See TI in Chapter 5 on the 'correction clock'.

(ii) Service the cursor by reversing the character under the cursor whenever the countdown becomes zero.

(iii) Switch off the cassette motor(s), unless a flag is set.

(iv) Scan the keyboard and perhaps update the keyboard buffer.

Of these, (iv) is the most intricate and takes the longest time. A table on the next page lists all the relevant RAM locations of these operations in an easily-referenced form. Some - the screen processing and the clock - are not directly relevant to the keyboard, but a single table is more convenient than several smaller tables.

To understand the keyboard scanning process, we must briefly consider the 6520 chip (PIA, or 'Peripheral Interface Adapter') which handles the hardware side of it. Chapter 14 maps out and explains this chip in greater detail. The PET/CBM has two of these chips; the first, 'PIA1', appears at E810 - E813. The locations which control the keyboard are labelled:

E810 (59408)	[| | | | Row select]	PORT A
E811 (59409)	[| | | | 0/1* |]	PORT A'S CONTROL REGISTER
E812 (59410)	[Row input (all 8 bits)]	PORT B
E813 (59411)	[| | | | 0/1* | 0/1²]	PORT B'S CONTROL REGISTER

*When 0, Port A's or B's contents configure the port for input or output. On switchon, bits 0-3 of Port A are configured as output, and bits 4-7 as input, by storing #0F in A. Port B is configured for input only.

²When 0, the interrupt is disabled; this turns off the keyboard scanning process.

E812 is wired so that all 8 bits are normally 1. The pattern of bits 0-3 of E810 determines which (of 9) rows is examined. To scan the entire keyboard therefore requires a loop to change E810's rightmost bits from 1 through 9, testing E812 each time for a zero bit, which indicates a keypress. There is an elaborate loop in the interrupt servicing routine which shifts the contents of E812 rightwards, looking for a zero, and continues this process while changing E810. When a zero is found, the corresponding key is found from the keyboard decode table. (See Chapter 15 for a diagram). It is quite easy to mimic this process in BASIC (there is a routine in 'PET Revealed'). The short machine-code routine shows which keys belong to which 'rows':

```
START LDA #00                    .: 027A A9 00 8D 10 E8 AD 12 E8
      STA E810                   .: 0282 AA A9 00 4C xx xx
      LDA E812
      TAX                        0 INPUT "ROW (0-9)";R
      LDA #00                    1 POKE 635,R: SYS 634: GOTO 1
      JMP DC9F/DCD9/CF83 [BASIC 1/2/4]
```

This code and its BASIC driver prints out the contents of E812 in decimal; if a single key is pressed in the row being scanned, a value 127,191,223,239,247,251,253, or 254 will be printed. No key is shown by 255. Each of these figures has a bit pattern of a single 0 within seven 1s (0111 1111, 1011 1111, etc.). The figures correspond with those shown in the keyboard decode tables; for example, 40-column BASICSs include =,·,Stop,<,Space,[, and Reverse in row 9. This row represents the normal state of E812 (on switchon and after the keyboard scan) which is the reason that LDA E812 is used as a test for the Stop key, and can also be used to test for the other keys in the row. Thus 'Wordcraft' for example uses the Reverse key and the Stop key to control its modes. Note that the rows are not arranged in straight lines on the keyboard, but are wired to select alternate keys (roughly).

The keyboard decoding matrix has 80 entries, 10 rows of 8 ASCII values, some disused. The table is scanned from the end to the beginning. Note that the shift key is signalled by 0.

IRQ Servicing - Keyboard, Screen, and Clock Locations.			
Description	**BASIC 1**	**Locations:** **BASIC 2**	**BASIC 4**
3-byte clock	0200-0202	8D-8F	8D-8F
Correction (2 bytes)	0205-0206	99-9A	99-9A
Stop key test (copy of E812)	0209	9B	9B
Cursor - 0=off, <>0 means on	0224	A7	A7
Cursor countdown (from #$14)	0225	A8	A8
Blink flag - 0=unreversed, 1=reversed	0227	AA	AA
True character under cursor (peek)	0226	A9	A9
Displacement 1-80 from start of table (or 255 if no key pressed)	0223	A6	A6
Shift key - 0=off, 1=on	0204	98	98
Displacement 1-80 of previous key, for comparison with 0223/A6/A6	0203	97	97
Number of characters in keybd. buffer	020D	9E	9E **12" 40 col:**
Length of keyboard buffer			E3 03EB
Keyboard buffer	020F-0218	026F-0278	026F-
Repeat - <128 off, 128+ on			E4 03EE
Repeat countdown			E5 03EA
Change of key indicator			E6 03E9
IRQ SERVICING ROUTINE	E685	E62E	E455
KEYBOARD DECODING TABLE	E75C-E7AB	E6F8-E747	E609-E658 (40-col) E6D1-E720 (80-col)

Bit 2 of the control registers of the PIA controls the data direction of the ports; by setting the bit low and entering new values in port A or port B, anomalous results can be obtained. POKE 59409,0 is a simple BASIC example; the full range of rows is no longer obtainable. Similarly, if the cassette interrupts are in use, poking E810 (in machine-code) with #XA in place of #X9 disables the Stop key. At this point, we can pause to examine Stop-key disabling: the usual POKE mentioned previously works by

redirecting the interrupt vector to point 3 bytes beyond its normal point; in this way, the first instruction of the IRQ servicing routine (e.g. JSR FFEA) is skipped; thus the Stop key is disabled and the clock turned off. Another method is to store #FF in the location which copies E812, and which is used to test for Stop:

```
JSR FFEA; UPDATE CLOCK/ COPY E812 FOR STOP TEST
LDA #FF
STA 9B  ; COPY OF E812 SET TO 'NO KEY'
JMP E458; CONTINUE NORMAL INTERRUPT
```

This routine, for 40-column BASIC 4, when inserted into the IRQ, behaves exactly like the usual interrupt, except that the Stop key, when pressed, is overwritten. Thus the internal clock runs normally, while Stop is disabled.

The Keyboard Buffer   It is well-known in CBM circles that the keyboard buffer can be programmed independently of the keyboard. (The process is sometimes described as 'a program writing itself' with certain applications).For a description see section 4.1.9 in Chapter 4. The keyboard buffer can be watched as it queues characters. This is worth doing to understand the process. Enter one of these routines:

```
.:027A A5 9E 09 30 8D 00 80 A0   :A5 9E 09 30 8D 00 80 A0   :A5 9E 09 30 8D 00 80 A4
.:0282 0A B9 6E 02 99 02 80 88   :0A B9 6E 02 99 02 80 88   :E3 B9 6E 02 99 02 80 88
.:028A D0 F7 4C 2E E6            :D0 F7 4C 55 E4            :D0 F7 4C 55 E4
      BASIC 2                      BASIC 4, 40-COL.          BASIC 4, 80-COL.
```

Then change the IRQ vector to 027A. The keyboard buffer, and the number of bytes in it, are displayed in the top-left of the screen. (Lower-case mode will make them more readable). A short routine of this sort: 1 GET X$: FOR J=0 TO 1000: NEXT:GOTO 1 will enable characters to be queued up from the keyboard; they are then removed at about 1 per second. The separate routine for 12" screen BASIC 4 allows for the fact that the buffer is not fixed at the normal 10 characters, but can be varied by poking $E3 (=227).   Characters remain in the buffer until they are fetched by GET or INPUT, or until the buffer is deleted (in BASIC <4) when more than 10 characters are entered. Poking the number of characters to zero in effect clears the buffer, giving the same effect as 100 GET X$: IF X$<>"" GOTO 100. Conversely, poking the number of characters to some non-zero value makes them available to the system. Try:

```
10 FOR J=623 TO 627 : READ X: POKE J,X: NEXT : REM 5 BYTES IN KEYBOARD BUFFER
20 POKE 158,5 : END                         : REM SET # BYTES = 5
30 DATA 72,69,76,76,79
```

The five bytes are printed out by the system when the program ends, exactly as if they had been keyed in. Replace the data statement with 30 DATA 76,73,83,84,13. Now the word 'LIST' followed by a carriage return is entered in the buffer, and the command is carried out. There are several examples of this type of routine in this book; see for example DELETE in Chapter 5. Note that BASIC 1 has different locations (525= #characters, 527-536 = buffer).

Examples of programming using the Interrupt   We shall see now how to carry out some ambitious routines involving the interrupt. First, we shall consider software repeat keys. These are not needed in 12" screen CBMs, which include repeating keys as a standard feature. But in the other models they are useful. The principle is not very difficult; the keyboard normally prevents automatic repeating by comparing the key pressed during a scan with that pressed before. If they are the same, nothing is done. Therefore, if our program intercepts the interrupt and sets the previously recorded key value to 255, the key is reinput to the buffer. See the next page for examples. The second example moves the entire interrupt routine into RAM, where it can be modified freely. For example, the rate of cursor flash can be changed, and this can be a useful reminder that a non-standard keyboard is in use. In this way the keyboard can be modified, as mentioned before, perhaps to provide a hex keypad, or a Dvorak typewriter, or to use one or both shift keys as special function keys. The third example is a single-key BASIC entry routine; one shift key, plus a key, enters an entire keyword. This principle can be generalised, so that for example a key may print any predefined string to the screen, and some commercial software and toolkits can do this.   The processing is of course virtually instantaneous; similar effects can be got in BASIC, with GET X$: IF X$="X" THEN X$="SOMETHING ELSE" , but this is slower, and takes up program space. Moreover, individual keys (e.g. both Shift keys) cannot be

distinguished from BASIC.

Software repeat keys (for 8" PET/CBMs, BASICs 1,2, and 4)

REPEAT KEY FOR BASIC 2.

```
027A LDA #84 ; INITIALISE          .: 027A A9 84 85 90 A9 02 85 91
027C STA 90                        .: 0282 60 EA A9 FF C5 97 D0 05
027E LDA #02                       .: 028A A9 (14) 8D 83 02 CE 83 02; Delay.
0280 STA 91                        .: 0292 D0 0B 85 97 A9 (05) 8D 83; Repeat.
0282 RTS                           .: 029A 02 A9 (03) 85 A8 4C 2E E6; Cursor.

0283 NOP      ; REPEAT           SYS 634 INITIALISES THIS REPEAT KEY.
0284 LDA #FF
0286 CMP 97                      Notes:
0288 BNE 028F                    [1] Varying the three bytes controls the
028A LDA #14 ; First delay 1/3 sec.   time taken before repeating starts, the
028C STA 0283                    rate of repetition, and the rate of flash
028F DEC 0283                    of the cursor.  For example, POKE 1 into
0292 BNE 029F                    each for the maximum repeat rate of 60
0294 STA 97                      characters per second.
0296 LDA #05 ; 1/12 sec. between [2] .M 0090 0090 from the monitor can be
0298 STA 0283;  repeats          used as a method of turning 'repeat' on
029B LDA #03 ; 1/20 sec. between and off, and redirecting IRQ generally.
029D STA A8  ;  cursor flashes   [3] If the cassette buffer is corrupted
029F JMP E62E                    while the repeat is operational, the
                                 interrupt will probably crash.  The
                                 exception is for tape activity itself,
                                 which resets the IRQ and so disables
                                 the repeat.
```

BASIC 1: Old ROMs differ in (i) IRQ vector, (ii) IRQ servicing location,
         (iii) keypress indicator location, and (iv) cursor flash countdown.
         These are: (i) ($021A), (ii) $E685, (iii) $0223, and (iv) $0225.
         The same logic may be used, but the resulting code inevitably occupies
         more space. The cassette buffer #2 version is this:

```
0 DATA 169,70,141,26,2,169,3,141,27,2,96: REM INITIALISES
1 DATA 0,169,255,205,35,2,208,5,169,20,141,69,3,206,69,3,208,13,141,35,2
2 DATA 169,5,141,69,3,169,3,141,37,2,76,133,230: REM REPEAT KEY DATA
3 FOR J=826 TO 870: READ K: POKE J,K: NEXT: SYS 826
```

BASIC 4: 80-column machines are equipped already with repeat. (Incidentally, this
         can be turned off by poking $E4 (228) with a value<128 and vice versa.
         And the repeat cursor flash rate is controlled by the contents of $E5).
         40-column machines are similar to BASIC 2, except that the IRQ destination
         is different: so substitute JMP $E455 and ... 4C 55 E4 in each BASIC 2
         program on this page. 12" screen 40-column models have some differences.

TINY REPEAT KEY.   (BASIC 2)

By using a zero page store, and having only one delay constant, 'repeat' can be
reduced to 19 bytes only. This is the shortest routine I've been able to write,
and I include it for those who like little routines:

```
    LDA #FF                     A9 FF C5 97 D0 06 85 97
    CMP 97                      A9 (10) 85 00 C6 00 F0 F6; Delay Constant.
    BNE +6 ┐                    4C 2E E6
  ┌→STA 97 │
  │ LDA #10 │ ; delay constant  Use .M 0090 0090 to point ($90) to the
  │ STA 00  │ ; zero page store start of this routine.
  │ DEC 00↵ ; (can use others)
  └─BEQ -10 ; decimal branch    At fast rates of repeat the cursor will
    JMP E62E                    flash too slowly to be always visible.
```

BASIC RELOCATING LOADER TO ENABLE USER TO DEFINE HIS OWN KEYBOARDS (E.G. HEX PAD).

BASIC 2

```
0 PRINT"[CLEAR][REVS]LOADS USER-DEFINED SPECIAL KEYBOARD AND PROTECTS IT IN TOP OF M
     EMORY
10 L = PEEK(52)+256*PEEK(53): T1=L: REM L IS CURRENT TOP OF MEMORY; T1 STORES IT
15 T = L - 302: REM WE RESET TOP OF STRINGS TO ENABLE SAFE INPUT OF CHARACTERS
16 LH%= T/256: LL% = T - LH%*256
17 POKE 52,LL%: POKE 50,LL%: POKE 48,LL%: POKE 53,LH%: POKE 51,LH%: POKE49,LH%
20 FOR J = 59207 TO 58926 STEP -1: POKE L-1, PEEK(J): L = L-1: NEXT
30 REM MOVE INTERRUPT ROUTINE $E62E-$E747 TO TOP OF MEMORY - 1; L = LOWER LIMIT
40 KT = L+201: REM THIS IS THE KEYBOARD-TO-ASCII-TABLE POINTER IN RAM (RTS)
50 KH% = KT/256: KL% = KT - 256*KH%: REM LOW AND HIGH BYTES OF NEW TABLE
60 POKE L+111,KL%: POKEL+112,KH%
70 POKE L+160,KL%: POKEL+161,KH%
80 REM BOTH REFERENCES TO THE KEYBOARD TABLE CHANGED TO POINT TO RAM, NOT ROM
90 PRINT"[CLEAR][REVS]ENTER *** WHEN YOUR KEYBOARD IS COMPLETE": REM COULD BE OTHER E
     XIT STRING
100 PRINT"[DOWN]PRESS KEY TO BE CHANGED [REVS]OR[RVSO] ENTER
101 INPUT"ITS ASCII VALUE, LIKE THIS: V14";K$ :REM TO DISTINGUISH 3 FROM CHR$(3)
102 IFK$="***"THEN500
105 K=ASC(K$) : REM IF A VALUE HAS BEEN ENTERED, NEXT LINE FINDS CORRECT K
110 IF LEFT$(K$,1)="V" AND LEN(K$)>1 THEN K=VAL(MID$(K$,2))
120 FOR J = 1 TO 80:IF PEEK (59127+J)<>K THEN NEXT: PRINT "[REVS]NOT FOUND": GOTO100
130 J = J + KT : REM J IS THE POSITION IN THE ROM TABLE AS RAM TABLE MAY VARY
140 REM WE NOW HAVE POSITION (=J) OF SOUGHT KEY IN RELOCATED TABLE IN RAM
200 PRINT"[DOWN]PRESS KEY TO REPLACE [REVS]OR[RVSO] ENTER
201 INPUT"ITS ASCII VALUE, LIKE THIS: V14";K$
202 IFK$="***"THEN500
205 K=ASC(K$)
210 IFLEFT$(K$,1)="V" AND LEN(K$)>1 THEN K=VAL(MID$(K$,2))
220 POKE J,K :REM REPLACE THE PREVIOUS KEY IN THE RAM TABLE WITH THE NEW ONE
230 GOTO 90 : REM  NOW DO THE NEXT CHARACTER
500 REM FINAL ROUTINE; POKE 2 ROUTINES TO CHANGE INTERRUPT ADDRESS
510 DATA 120,169,46,133,144,169,230,133,145,96
520 DATA 120,169,46,133,144,169,230,133,145,96
530 FOR J = L-20 TO L-1: READ X: POKE J,X: NEXT
540 L = L-20: REM NEW LOW MEMORY LIMIT = START OF ENTIRE ROUTINE
550 PRINT"[CLEAR][DOWN]SYS";L;"TURNS ON THE NEW KEYBOARD,AND
560 PRINT"SYS";L+10;"RETURNS TO NORMAL KEYBOARD
570 POKE L+2, (L+20) - INT((L+20)/256)*256: POKE L+6,(L+20)/256
580 REM TURN-ON ROUTINE WILL NOW LOAD THE CORRECT INTERRUPT ADDRESS.
590 PRINT"[DOWN]SAVE FROM"L"TO"T1
595 PRINT "            ($";: GOSUB600: PRINT " TO $";: L = T1  : GOSUB600: PRINT ")"
596 END
599 REM  ### ONE LINE DECIMAL TO HEX CONVERTER ###
600 L=L/4096:FORJ=1TO4:L%=L:L$=CHR$(48+L%-(L%>9)*7):PRINTL$;:L=16*(L-L%):NEXT:RETURN
```

Modifications for BASIC 1

```
10 L=PEEK(134)+256*PEEK(135): T1=L: REM L IS CURRENT TOP OF MEMORY; T1 STOR
     ES
15 T = L - 319:REM WE RESET TOP OF STRINGS TO ENABLE SAFE INPUT OF CHARACTE
     RS
17 POKE134,LL%: POKE132,LL%: POKE130,LL%: POKE135,LH%: POKE133,LH%: POKE131
     ,LH%
20 FOR J = 59307 TO 59013 STEP -1: Q=PEEK(J): POKE L-1, Q: L = L-1: NEXT
40 KT = L+214: REM THIS IS THE KEYBOARD-TO-ASCII-TABLE POINTER IN RAM (RTS)
60 POKE L+158,KL%: POKE L+159,KH%
70 POKE L+187,KL%: POKE L+188,KH%
120 FOR J = 1 TO 80:IF PEEK (59227+J) <> K THEN NEXT: PRINT "[RVS]NOT FOUND"
     : GOTO 100
510 DATA 120,169,133,141,25,2,169,230,141,26,2,96
520 DATA 120,169,133,141,25,2,169,230,141,26,2,96
530 FOR J = L-24 TO L-1: READ X: POKE J,X: NEXT
540 L = L-24: REM NEW LOW MEMORY LIMIT = START OF ENTIRE ROUTINE
550 PRINT "[CLR][DOWN]SYS";L;"TURNS ON THE NEW KEYBOARD,AND
560 PRINT "SYS";L+12;"RETURNS TO NORMAL KEYBOARD
570 POKE L+2, (L+24) - INT((L+24)/256)*256: POKE L+7,(L+24)/256
```

REDEFINE KEYBOARD FOR 8032 ONLY:-

```
0 PRINT"[CLR][RVS]LOADS USER-DEFINED SPECIAL KEYBOARD AND PROTECTS IT IN TO
     P OF MEMORY
10 L = PEEK(52)+256*PEEK(53): T1=L: REM L IS CURRENT TOP OF MEMORY; T1 STORE
     S IT
15 T = L - 736: REM WE RESET TOP OF STRINGS TO ENABLE SAFE INPUT OF CHARACTE
     RS
16 LH% = T/256: LL% = T - LH%*256
17 POKE 52,LL%: POKE 50,LL%: POKE 48,LL%: POKE 53,LH%: POKE 51,LH%: POKE49,L
     H%
20 FOR J = 59168 TO 58453 STEP -1: POKE L-1, PEEK(J): L = L-1: NEXT
30 REM MOVE INTERRUPT ROUTINE $E455-$E720 TO TOP OF MEMORY - 1; L = LOWER LI
     MIT
40 KT = L+635: REM THIS IS THE KEYBOARD-TO-ASCII-TABLE POINTER IN RAM (RTS)
50 KH% = KT/256: KL% = KT - 256*KH%: REM LOW AND HIGH BYTES OF NEW TABLE
60 POKE L,32: POKE L+1,234: POKE L+2,255: REM JSR $FFEA TO UPDATE CLOCK
65 SR=L+105: SH%=SR/256: SL%=SR-256*SH%: POKE SR-5,SL%: POKE SR-4,SH%
66 REM PREVIOUS LINE ALTERS A JSR INSTRUCTION TO POINT TO OUR RAM ROUTINE.
70 POKE L+135,KL%: POKEL+136,KH%
80 REM BOTH REFERENCES TO THE KEYBOARD TABLE CHANGED TO POINT TO RAM, NOT RO
     M
90 PRINT"[CLR][RVS]ENTER *** WHEN YOUR KEYBOARD IS COMPLETE": REM COULD BE O
     THER EXIT S
100 PRINT"[DOWN]PRESS KEY TO BE CHANGED [RVS]OR[RVSO] ENTER
101 INPUT"ITS ASCII VALUE, LIKE THIS: V14";K$ :REM TO DISTINGUISH 3 FROM CHR$
     (3)
102 IFK$="***"THEN500
105 K=ASC(K$) : REM IF A VALUE HAS BEEN ENTERED, NEXT LINE FINDS CORRECT K
110 IF LEFT$(K$,1)="V" AND LEN(K$)>1 THEN K=VAL(MID$(K$,2))
120 FOR J = 1 TO 80:IF PEEK (59088+J) <> K THEN NEXT: PRINT "[RVS]NOT FOUND":
     GOTO 100
130 J = J + KT : REM J IS THE POSITION IN THE ROM TABLE AS RAM TABLE MAY VARY

140 REM WE NOW HAVE POSITION (=J) OF SOUGHT KEY IN RELOCATED TABLE IN RAM
200 PRINT"[DOWN]PRESS KEY TO REPLACE [RVS]OR[RVSO] ENTER
201 INPUT"ITS ASCII VALUE, LIKE THIS: V14";K$
202 IFK$="***"THEN500
205 K=ASC(K$)
210 IFLEFT$(K$,1)="V" AND LEN(K$)>1 THEN K=VAL(MID$(K$,2))
220 POKE J,K :REM REPLACE THE PREVIOUS KEY IN THE RAM TABLE WITH THE NEW ONE
230 GOTO 90 : REM  NOW DO THE NEXT CHARACTER
500 REM FINAL ROUTINE; POKE 2 ROUTINES TO CHANGE INTERRUPT ADDRESS
510 DATA 120,169,85,133,144,169,228,133,145,96
520 DATA 120,169,85,133,144,169,228,133,145,96
530 FOR J = L-20 TO L-1: READ X: POKE J,X: NEXT
540 L = L-20: REM NEW LOW MEMORY LIMIT = START OF ENTIRE ROUTINE
550 PRINT"[CLR][DOWN]SYS";L;"TURNS ON THE NEW KEYBOARD,AND
560 PRINT"SYS";L+10;"RETURNS TO NORMAL KEYBOARD
570 POKE L+2, (L+20) - INT((L+20)/256)*256: POKE L+6,(L+20)/256
580 REM TURN-ON ROUTINE WILL NOW LOAD THE CORRECT INTERRUPT ADDRESS.
590 PRINT"[DOWN]SAVE FROM"L"TO"T1
595 PRINT "             ($";: GOSUB600: PRINT " TO $";: L = T1   : GOSUB600: PRINT
     ")"
596 END
599 REM   ### ONE LINE DECIMAL TO HEX CONVERTER ###
600 L=L/4096:FORJ=1TO4:L%=L:L$=CHR$(48+L%-(L%>9)*7):PRINTL$;:L=16*(L-L%):NEXT
     :RETURN
```

Single-key entry of BASIC keywords &c. Special-purpose programmable keys can be implemented on the PET/CBM. The Shift-Stop key of course is a ROM example of this; it forces dload "*",8 in compressed form into the keyboard buffer. Commodore's VIC and the 'standard data entry' machine-code routine also have programmable keys. The following example illustrates a method of patching the IRQ vector, after moving it into RAM, to add extra keys. It does this by coding the right-shift key as ASCII 7, which is not otherwise used, then checking for this value when the keyboard is scanned. If it is found, a short routine converts the shifted key into a BASIC word and prints it. This version is BASIC 2; I've used addresses almost identical to those in ROM purely for ease of reference, moving the IRQ routine from E62E to 362E for instance. The principles are similar for any ROM, although 12" screen models tend to have longer decoding routines to process the automatic repeats. The left shift key operates with the shift-lock; for this reason the right shift is preferable as a control key. The keyboard has about 64 alphanumeric and punctuation keys. The routine that follows excludes some of the 76 BASIC 2 tokens, including +,-; ...,>,=,< which are single keys already, END, and GO.

    (i) If BASIC in in use, set the top-of-memory to $3000.

    (ii) Move E62E - E747 to 362E - 3747. This is the whole of the interrupt servicing routine and the decode table. We can add our extensions to this routine at 3748.

    (iii) Poke 36CF and 369E with 36 in place of E6. [These locations look at the ROM decode table; they are both LDA E6F7,X. After the poke they reference the RAM table].

    (iv) Poke $3702 with 7; this is the right shift key, appearing as the first of two zeroes in the decode table.

    (v) We put a patch at the point where the shift key is tested. The short piece of code following LDA E6F7,X /BNE xx/ processes the shift key, storing 1 in its flag. After our modification, right-shift stores 7 in the flag.

```
369F   JMP 3748   ;JUMP TO THE FIRST RAM ROUTINE AFTER THE DECODE TABLE

3748   BNE 3751
374A   LDA #01
374C   STA 98
374E   JMP 36B6   ;THESE 3 INSTRUCTIONS IMITATE THE ROM ROUTINE (BUT JMP, NOT BNE)
3751   CMP #07    ;LOOK FOR RIGHT-SHIFT
3753   BEQ 3758
3755   JMP 36A7   ;IMITATES THE ROM ROUTINE, RE-ENTERING TO COMPARE #FF
3758   STA 98     ;STORES RIGHT-SHIFT IN SHIFT LOCATION
375A   BNE 374E   ;BRANCH ALWAYS TO EXIT FROM SHIFT-FOUND PART OF ROUTINE
```

The keyboard with IRQ directed to this routine will behave normally, because #7 is processed by a shift-right, and appears similar to #1 from the point of view of shift-key processing. What is needed now is a further patch, within the coding which ORAs shifted keys with #80, to test for the right-shift.

    (vi) In BASIC 2, E6D2 has the BCC instruction testing for shift; we replace it:

```
36D2   JSR 375C   ;CALLS THE SECOND ROUTINE AFTER THE TABLE
36D5   NOP

375C   BCS 375F   ;IF SHIFT PRESSED, CARRY IS SET
375E   RTS
375F   LSR 98     ;TEST FOR #7 IN RIGHT-SHIFT
3761   BCS 3766
3763   EOR #80    ;LEFT-SHIFT. PUT IN UPPER-CASE
3765   RTS        ;AND RETURN
3766              ;RIGHT-SHIFT PROCESSING BEGINS.
```

```
3766   CMP #20
3768   BCC 3765
376A   SBC #1E
376C   CMP #2B
376E   BCC 3772
3770   ADC #09
3772   TAX
3773   LDY #FF
3775   DEX
3776   BEQ 3780
3778   INY
3779   LDA C092,Y
377C   BPL 3778
377E   BMI 3775
3780   INY
3781   LDA C092,Y
3784   BMI 378B
3786   JSR CA45
3789   BNE 3780
378B   AND #7F
378D   JMP CA45
```

At this point we have isolated the right-shift from the left, and can insert any routine which will serve our purposes. The example tests for reverse and cursor-control characters, with ASCII value less than 32, leaving these unchanged; it corrects for the 9 arithmetic tokens, already single keys; and it prints the Xth. BASIC reserved word, using logic largely taken from LIST.

    (vi) Finally, set the IRQ vector to 362E to drive the new routine. Poking $91 (=145) with $36 (=54) is convenient from BASIC.

8.8 Other firmware and hardware.

Reset switches   There are three (at least) reset switches usable with the PET/CBM; two are hardware, one is software. A reset switch provide an alternative to turning the machine off, then on, when a program has 'crashed'. A crash (i.e. completely unresponsive machine) is caused by the execution of an infinite loop. It will only happen in BASIC if the program has peeked, poked, SYSed, or USRed, although slow machine-code routines (notably memory freeing in BASIC<4) may give the appearance of a crash. Code like this: 027A LDA #00/ 027C BEQ 027C   or: 0300 JMP 0303/ 0303 JMP 0300 obviously gives an infinite loop; so do some pseudo-opcodes ending in 2 when written in hexadecimal. Usually, of course, the cause is more subtle than this. Typically, RAM code is overwritten accidentally, or a wrong location is jumped to, and the new code happens to have a loop somewhere. Incorrect stack handling can easily cause this type of problem; so can memory-move routines.

     How do hardware switches work? The earliest, for BASIC 1, made use of the reset vector at (FFFC) in all 6502-based machines. Normally, this vector is used when the machine is switched on, and jumps to a routine to initialise the whole of BASIC, poking in software values, and incidentally calculating the amount of RAM, and also configuring all the input-output chips so the keyboard and cassettes and so forth will   operate properly. However, if the so-called 'diagnostic sense' pin (bit 7 of PIA 1) is low, i.e. grounded, an alternative routine is entered, originally a test routine for BASIC 1. This feature has been retained in BASIC>1, but the alternative is now a 'call' entry to the monitor, printing *C in place of the break entry's *B. This is useful, because this entry retains most of the features of the program in RAM. So this switch requires two connections: one from pin 5 of the user port to ground (see the diagram at the end of Chapter 9. Pins 5 and 12 are the relevant ones). And another, connected from pin 22 to pin 25 on the J4 connector along the right-hand side within the PET/CBM. This puts the machine into the monitor.(If the 'diagnostic sense' pin is left low, switching on causes entry to the monitor, not BASIC. If it is disconnected, pin 22 to pin 25 will reset the machine into BASIC, clearing most RAM).Unfortunately this process is not hazard-free, since grounding reset is not safe (on the PET/CBM) for more than a few seconds. Moreover, the start of the resetting process alters the stack pointer irretrievably. The usual process on entering the monitor is to enter .X to return to BASIC, then CLR to set the stack (among other things). Machine-code is tidied by entering a meaningless command (usually .; is used) then changing SP to #F8 or (BASIC 1) #FA.

     A safer method (with fewer wires) uses the non-maskable interrupt (NMI) line, which has a vector at (FFFA). This was unusable in BASIC 1, but BASIC>1 sets this vector to print READY. In this case, pin 24 of J4 is momentarily connected to pin 25 of J4. Either of these methods, to be used routinely, require proper hardware, with a capacitor arrangement to debounce the connection. (Note that these pins are marked on the printed circuit board).*   The NMI method fails with the X2-type crash.

     Software uncrashing relies on the normal interrupt sequence for its effect. If the interrupt is off the method cannot work. And it too fails with X2 crashes. These restrictions are not very serious. The method is straightforward; we can redefine the Stop key as a reset key by slightly modifying the interrupt processing.

     An interrupt causes the program counter and status register to be pushed on the stack, and the program counter to be loaded with the contents of (FFFE). In the PET A,X, and Y are saved and a RAM address jumps to E685/E62E/E455. This is: JSR update clock & load A from E812 (to test Stop), then the remainder of the routine and RTI to return to the main processing.

     If the RAM address points to a routine of this sort:

```
JSR update clock/load E812
CMP #EF
BEQ +3
JMP remainder of interrupt processing routine
JMP machine-code monitor
```

then the software reset is operational.

This routine leaves three bytes from the interrupt on the stack. Naturally a different version is needed for each type of PET/CBM. BASIC 4 is slightly different from the

---

*Instructions for wiring both types (simultaneously!) are printed (e.g.) in Kb-Micro-computing, (J Strasma, Sept.'80). Some other hardware tricks of this sort are possible with the 6502; RDY (ready) can halt the 6502 when it is fetching opcodes, so that single instructions can be executed, for example.

others; the version here causes the clock to run at 5/6ths normal speed with 12-inch screen CBMs!*

BASIC 1				BASIC 2				BASIC 4			
JSR FFEA	20 EA FF			JSR FFEA	20 EA FF			JSR FFEA	20 EA FF		
CMP #EF	C9 EF			CMP #EF	C9 EF			CMP #EF	C9 EF		
BNE +3	D0 03			BNE +3	D0 03			BNE +3	D0 03		
JMP 040F	4C 0F 04			JMP FD11	4C 11 FD			JMP D472	4C 72 D4		
JMP E688	4C 88 E6			JMP E631	4C 31 E6			JMP E458	4C 58 E4		

EPROMS Unmodified PET/CBMs have 7 sockets for ROMs on their printed circuit boards; BASIC<4 uses 4, and BASIC 4 5, so that there are empty sockets spanning $9000-9FFF, $A000-$AFFF, and (BASIC<4 only) $B000-$BFFF. The sockets accept either 2K or 4K EPROMs, of type 2716 or 2532 respectively. An 'EPROM' ('Erasable Programmable Read-Only Memory') appears in use like a ROM; it can be read from, but not written to. As its name implies, its contents can be erased and replaced; to do this, an EPROM Programmer or 'Burner' has the required bytes loaded into it, and these are entered in a semi-permanent way into the EPROM with a relatively high voltage pulse of 27 volts or so. Erasure is performed by removing the opaque covering on top of the chip and exposing the chip to UV light from what has been called 'the world's most expensive ultra-violet lamp'. One of the first PET chips was Nestar's 'Toolkit'; it was also one of the most popular. It added commands to BASIC using a wedge. These are rather rudimentary trace, step and renumber facilities, tape append, 'find', and variable dump. Later BASIC extenders included 'Disk-o-Pro' for BASIC 2, which adds the disk commands found in BASIC 4, and 'Command-O' for BASIC 4, with screen and directory scrolling up and down, a 'print using', and search-and-replace.

'Power' is another well-known EPROM, but there are many more, including some complete packages (Visicalc, word processors). Most are designed for the $9000-$9FFF slot, with $A000-$AFFF a close second. Obviously it is impossible to run EPROMs which conflict in their requirements simultaneously. Some have alternative versions for several slots; few if any will relocate. Multiple sockets are available which accept several chips and allow switching between them, and this may be convenient (if expensive). Amongst the several dozen chips on sale are some of the 'Toolkit' utility type, intended to help with writing BASIC, and some providing similar help with machine-code. Others are designed for business use, easing input and output for example, for graphics use, some in association with hardware, and for tape or disk use. Some few are specific in nature, dealing (say) with matrix calculations. Unfortunately, documentation is often poor, and reviews are usually rush jobs which fail to mention bugs and pitfalls. The purchaser of EPROMs therefore should be wary. Moreover, as the technology to copy chips becomes more widely available, some of the incentive to produce good-quality firmware is lost or at least weakened.

Other types of EPROM - EAROM = Electrically Alterable ROM, for instance - are only important when hardware modifications allow the CBM to write to 'ROM', so that software can be held semi-permanently, in 'Instant ROM' and other products. This requires the write-enable line and a power supply connected to the 'ROM' package.

Other Black Boxes More ambitious add-ons include CP/M, a well-known standard microcomputer control program, designed for the Z80. The CBM has an external Z80, which runs the program in place of its 6502. This is a radical difference. At the time of writing, the U.K. company 'Small Systems Engineering' appears to have the only working version. Prestel (Videotex in the US) is often demonstrated on micros, and at least one system is available. At present this is (virtually) a receive-only system, but 2-way transmission may be feasible - Mullard's 'Lucy' chip is reported to be able to handle this at speed. Some multiple PET-to-disk systems exist, enabling users to share the use of the relatively expensive disks.

Industry, process control, and research Industrial applications of PET-type microcomputers range from dedicated (i.e. single) operation as an electronic instrument, to

---

*This can be programmed around by loading the accumulator from E812 independently of the clock update routine; LDA E812/ CMP E812/ BNE -8/ CMP #EF/ BNE +3/ JMP MONITOR/ JMP IRQ SERVICE. The E812 processing provides a simple debounce.

fairly large-scale process control. Full explanation of the hardware side of this is outside the scope of this book.* But a few examples illustrate what can be achieved with the aid of these machines. Most such applications are developed in industrial or academic laboratories, with some interchange of ideas between the two. Early examples include stepper motor control, one of the easier devices to program, and measuring and inspection machinery, such as balances and gauges. Data loggers, taking advantage of the video display, include equipment like transient recorders and spectrum analysers. Control equipment has been developed to handle the more mechanical side of research into animal behaviour. Smaller-scale models of test rigs have been constructed: a much-publicised motor-engine tester and a computer-controlled drill, again are typical of the sort of thing. There is a well-known standard computer application of mathematical theory to cutting up sheets of metal, paper, etc., which is the sort of optimisation process capable of calculation by the microcomputer. Recently, the computer press carried a story of a CBM equipped with hardware (via the user port) and software which were able between them to control a dairy packaging system. The engineering side of the packaging machinery was controlled by pneumatic valves.

Successful hardware of any degree of complexity requires considerable skill in design, partly because of the difficulties in estimating overall tolerances and cumulative errors in all the parts when they are put together. As for the software, I quote the Chairman of Research Machines Ltd: "It is virtually impossible to overstate the time taken to get software up and running ... a successful application within one year means you're doing well at the moment...".

Single on-off switches are easy to implement with machines like the PET; all that's needed is a set-up which protects the PET from excessive current or voltage while amplifying its signals. Chips which demultiplex (e.g. 3 wire to 8 wire convertors) are commonplace. Analogue-to-digital conversions and vice-versa are more difficult, because more lines are used up; so (say) temperature control is harder work, unless simple on-off controls are sufficient. Such devices may have to be polled (i.e. examined in turn), or controlled by regular timing, or perhaps use an interrupt. In addition, data conversion may be needed by or from pieces of equipment with non-PET data conventions. Of course, an off-the-shelf package of combined hardware and software, if it exists, may be able to deal with all these matters satisfactorily.

The program reads and displays the input from an analogue-to-digital converter. This one (a Siliconix LD130) converts voltages from 0 - 1 into binary coded decimal output, one digit at a time, so the hundreds, tens, and units figures are output individually. The result is .000 to .999. Bit 7, when set low, signals correct data in the byte; bits 4, 5, or 6 signal units, tens, or hundreds, depending on which one is high; and bits 0-4 hold the current value, 0 - 9, of units, tens, hundreds. Hardware programming differs from ordinary PET/CBM program in that the addresses which look like RAM in fact vary according to external events, so the style of programming has to reflect this. The comparatively long-winded programs which result are typified by this specimen.

```
TABLE  #40 #20 #10;TEST H,T, OR U
START  LDA E84F   ;AWAIT END OF
       BPL START  ;VALID PULSE
       LDX #00
WAITV  LDA E84F
       BMI WAIT   ;AWAIT VALID DATA
COMP   BIT TABLE,X;100s? [10s? 1s?]
       BEQ NEXT   ;IF NOT, BRANCH
       CLC
       ADC #30
       STA 8000,X;NUMBER ON VDU
       BNE WAITV
NEXT   INX
       CPX #03    ;ALL 3 DIGITS?
       BNE COMP   ;BRANCH IF NOT YET
       BEQ WAITV
```

---

*A small number of books deal with CBM-related hardware. The most recent is 'PET Interfacing' by J Downey and S Rogers; this is not an introductory text. 'Programming and Interfacing the 6502' (M DeJong) deals with the 6502, not specifically the PET/CBM, in some detail. Both these titles are listed as 'Blackburg Series' publications in the US. Of earlier books, Caxton Foster's 'Programming a Microcomputer:6502' is all about hardware, but mostly the KIM-1; and Zaks' '6502 Applications Book' has nothing specific on the PET/CBM.

The computing press has hardware articles; often, rather surprisingly, hardware is covered more reliably by journals of the Wireless World and Practical Electronics type, where, presumably, the readership and editorial staff expect a reasonable standard of accuracy. Sometimes outside contributors have a similar effect; for example CPUCN 3 #3 has an article on parallel to serial conversion by A Strutt & K Hobbs of ICI which is unusually well thought-out.

CHAPTER 9: GRAPHICS AND SOUND

9.1 PET/CBM Screens

Screen and character-generating ROM   All CBM machines to date have had a screen (or VDU = visual display unit) built in. Commodore's VIC ('Video interface chip' - or 'Volkscomputer' in Germany, to avoid an unfortunate pun) departs from this tradition, using, like Apple and Tandy, an external TV, so that colour is available. PET/CBMs have 8 inch or 12 inch screens, always black and white; a colour computer demonstrated by Commodore at a show hasn't subsequently resurfaced.

Confining ourselves to current models, we can see that the original, blue-white phosphor 8 inch screens were replaced by 8 inch green screens, which have now been replaced by 12 inch green screens, presumably to cut down on manufacturing hassle. All *new* models, whether 40 or 80 column, now have 12 inch screens. In the course of time, probably the 40-column models will be discontinued in favour of 40-column VICs. (This is a guess on my part). The internal circuitry has been tidied and modularised in the process. Its system is peculiar to Commodore.

The signal sent to the screen (or to an external monitor) has three components: horizontal position, vertical position, and indication whether or not to put a dot on the screen. Each character is made up of 8 dots by 8; typically only 7 by 5 make up the actual area holding the points of the character, so the edges of characters don't interfere too much with their neighbours. Nevertheless the appearance of reversed capitals can be improved by printing CHR$(100)s above them; and the 12-inch machines have the facility to change line separation using the CRT chip. CBM printers have 8 by 6 characters, and so cannot exactly reproduce the screen. Since each screen character is 8 dots deep, there are 8*25 = 200 scans of the screen before flyback to the start. As we shall see, the flyback is handled differently by the newer machines, making for some ROM incompatibilities with the older models.

External monitors can be connected to PET/CBMs to enable (say) a class to watch a demonstration program; the hardware is connected to the user port, which provides video, vertical, and horizontal signals from connections 2,9, and 10 respectively. See CPUCN, joint issue 1 and 2 or the 'Pet Revealed' for circuit diagrams (neither of which I've tested). These produce output suitable for monitors, not UHF TVs.

The actual pattern of dots making up a character is created by the character-generating ROM,*which is in the main board of the PET/CBM near the other ROMs. It converts any byte into a fixed pattern, in a manner similar to a look-up table. 256 separate patterns are stored within it, although it is possible to switch between several ROMs, and Commodore has its well-known pair of character sets available. POKEing 59468 with 12 or 14 switches the character generator into its upper case/ graphics mode or its lower/ upper case mode. The character sets are in fact very similar, except for the fact that A-Z when shifted produces graphics characters in the one case, and alphabetic characters in the other. Most (all but four) of the remaining characters are unaffected by the ROM poke; the chart on the following page shows the arrangement. If PRINT statements are being used, it is impossible to have *both* the full set of graphics and lower- and upper-case, as the chart shows, and experiment will prove.

---

*Commodore's VIC uses analogous, but different, principles. The 22-column version has 22 by 23 characters, fitting slightly less than 2 pages of RAM - 506 bytes of 512. The character generation is a RAM function, so user-definable character sets are quite easy (if laborious) to write. Each character's colour is controlled by 3 bits, from a byte in a 506-byte table. Only one background colour and one border colour can exist.

NEXT PAGE: Table of CBM 'ASCII' characters, in decimal/ hex order:-

Example: PRINT "$" and PRINT CHR$(36) print the dollar symbol.
         PRINT CHR$(65) prints 'a' or 'A' depending on the screen's mode.
         PRINT CHR$(19) and PRINT "[HOME]" (i.e. PRINT "▓") home the cursor.

*These control characters are available only on the 8032 and 12-inch 4032.
[2]Note that 96-127 and 224-255 appear as repeats of characters 32-63 and 160-191. So PRINT CHR$(98) prints a quote mark; however, the quotes flag is not set. There are 64 ordinary characters and 64 graphics/ upper case characters in all.
[3]8032 only.

Dec	Hex	Char		Dec	Hex	Char		Dec	Hex	Char		Dec	Hex	Char	
0	00			64	40	@		128	80			192	C0		
1	01			65	41	a	A	129	81			193	C1	A	
2	02			66	42	b	B	130	82			194	C2	B	
3	03	STOP		67	43	c	C	131	83	[D]LOAD & RUN		195	C3	C	
4	04			68	44	d	D	132	84			196	C4	D	
5	05			69	45	e	E	133	85			197	C5	E	
6	06			70	46	f	F	134	86			198	C6	F	
7	07	BELL*		71	47	g	G	135	87			199	C7	G	
8	08			72	48	h	H	136	88			200	C8	H	
9	09	TAB*		73	49	i	I	137	89	SET TAB*		201	C9	I	
10	0A	LINE FEED		74	4A	j	J	138	8A			202	CA	J	
11	0B			75	4B	k	K	139	8B			203	CB	K	
12	0C			76	4C	l	L	140	8C			204	CC	L	
13	0D	RETURN		77	4D	m	M	141	8D	SHIFT-RETURN		205	CD	M	
14	0E	TEXT*		78	4E	n	N	142	8E	GRAPHIC*		206	CE	N	
15	0F	SET TOP³		79	4F	o	O	143	8F	SET BOTTOM³		207	CF	O	
16	10			80	50	p	P	144	90			208	D0	P	
17	11	CURSOR DOWN		81	51	q	Q	145	91	CURSOR UP		209	D1	Q	
18	12	REVERSE		82	52	r	R	146	92	REVERSE OFF		210	D2	R	
19	13	HOME CURSOR		83	53	s	S	147	93	CLEAR SCREEN		211	D3	S	
20	14	DELETE CHR.		84	54	t	T	148	94	INSERT CHR.		212	D4	T	
21	15	DELETE LINE*		85	55	u	U	149	95	INSERT LINE*		213	D5	U	
22	16	ERASE END*		86	56	v	V	150	96	ERASE START*		214	D6	V	
23	17			87	57	w	W	151	97			215	D7	W	
24	18			88	58	x	X	152	98			216	D8	X	
25	19	SCROLL UP*		89	59	y	Y	153	99	SCROLL DOWN*		217	D9	Y	
26	1A			90	5A	z	Z	154	9A			218	DA	Z	
27	1B	ESCAPE*		91	5B	[		155	9B			219	DB		
28	1C			92	5C	\		156	9C			220	DC		
29	1D	CURSOR RIGHT		93	5D	]		157	9D	CURSOR LEFT		221	DD		
30	1E			94	5E	↑		158	9E			222	DE		
31	1F			95	5F	←		159	9F			223	DF		
32	20	SPACE		96	60	SPACE²		160	A0	SHIFT-SPACE		224	E0	REPEATS²	
33	21	!		97	61	!		161	A1			225	E1		
34	22	QUOTE	"	98	62	"		162	A2			226	E2		
35	23	#		99	63	#		163	A3			227	E3		
36	24	$		100	64	$		164	A4			228	E4		
37	25	%		101	65	%		165	A5			229	E5		
38	26	&		102	66	&		166	A6			230	E6		
39	27	'		103	67	'		167	A7			231	E7		
40	28	(		104	68	(		168	A8			232	E8		
41	29	)		105	69	)		169	A9			233	E9		
42	2A	*		106	6A	*		170	AA			234	EA		
43	2B	+		107	6B	+		171	AB			235	EB		
44	2C	,		108	6C	,		172	AC			236	EC		
45	2D	-		109	6D	-		173	AD			237	ED		
46	2E	.		110	6E	.		174	AE			238	EE		
47	2F	/		111	6F	/		175	AF			239	EF		
48	30	0		112	70	0		176	B0			240	F0		
49	31	1		113	71	1		177	B1			241	F1		
50	32	2		114	72	2		178	B2			242	F2		
51	33	3		115	73	3		179	B3			243	F3		
52	34	4		116	74	4		180	B4			244	F4		
53	35	5		117	75	5		181	B5			245	F5		
54	36	6		118	76	6		182	B6			246	F6		
55	37	7		119	77	7		183	B7			247	F7		
56	38	8		120	78	8		184	B8			248	F8		
57	39	9		121	79	9		185	B9			249	F9		
58	3A	:		122	7A	:		186	BA			250	FA		
59	3B	;		123	7B	;		187	BB			251	FB		
60	3C	>		124	7C	>		188	BC			252	FC		
61	3D	=		125	7D	=		189	BD			253	FD		
62	3E	<		126	7E	<		190	BE			254	FE		
63	3F	?		127	7F	?		191	BF			255	FF		

The earliest PETs followed other computers in giving capital letters primacy, so the shift key moves from upper case to lower case. CBM machines adopted the normal typing convention of shifting to upper-case. Their character-generating ROMs are consequently different, so that the chart on the previous page has its two sets of alphabetics arranged with 65-90 as upper case only, and 193-223 as lower-case or graphics. This leads to a confusion of terminology. We can talk about 'lower-case mode' and 'graphics mode' with anything except BASIC 1; 'standard' and 'alternate' character sets are often used to describe this earliest arrangement, where 'standard' means that shift gives lower-case, and 'alternate' shifts to graphics. I shall use 'graphics mode' (POKE 59468,12) and 'lower-case mode' (POKE 59468,14), hoping that business keyboard users, with graphics obtainable only by poking, will understand!

A couple of simple programs show how the screen memory and the character-generation interact. PET/CBM screens are wired so that consecutive RAM locations store the characters in the screen in the normal left-to-right then down sequence. The starting location is $8000, exactly midway in the 6502's memory map. 40 column machines store 40*25 (1000), and 80 column machines 80*25 (2000) bytes in this way. The address decoding is incomplete, so PEEKs and POKEs to locations outside the expected range produce echoes or 'images' in the screen. The 40 column machine uses 1000 of 1024 bytes, $8000-$83E7, to map the screen. (This is 32768-33767). The 24 bytes from $83E8-$83FF are normally unused.

In a 40-column machine, $8400 behaves just like $8000. An 80-column machine, of course, maps its screen to $8000-$8800. Since the starting address ($8000 = 32768) is identical in each machine,* the following simple program displays all 256 characters which the ROM can generate:

```
10 FOR J = 32768 TO 32768 + 255 :REM 256 VALUES NEED 256 SCREEN LOCATIONS
20 POKE J,K: K=K+1           :REM POKE 0,1,2,3,...,255
30 NEXT
```

This fills the top few lines of the screen with an entire character-set. We can look at both character sets by adding this line to switch between the two:

```
40 POKE 59468,12: FOR J = 1 TO 500: NEXT: POKE 59468,14: FOR J=1TO500:NEXT:GOTO40
```

This displays all the characters which can be generated, since no other values than 0 to 255 can be stored in the screen RAM, and only two modes exist. It may surprise business keyboard users to see all these graphics, which cannot be entered from the keyboard; the decoding process of the keyboard is rewritten (e.g. in the 8032) to remove these characters. Chapter 13 gives a machine-code routine to enable any character to be input from the keyboard. A table of CBM screen memory, showing decimal and hex values corresponding to graphics poked/peeked to the screen, on the next page shows the entire gamut of characters for BASIC>1. Where lower-case mode and graphics mode differ, the two alternative appearances are placed side by side. It is impossible for the pi symbol and the 4 by 4 chequered symbol to appear on the same screen, or for lower-case x and the heart symbol to co-exist; limitations like these should be borne in mind.

By careful synchronization, it is possible to watch characters changing modes. In fact, with machine-code, the screen can be made to display non-existent characters, made up of single lines from several different characters! However, the severe time requirements make this technique hard to use. This BASIC program[2]

```
10 FOR J = 1 TO 1000: PRINT "A";: NEXT: REM I.E. SHIFT-A. USE 2000 FOR 80-COLS.
20 X=59468: Y=12: Z=14          : REM ROM LOCATION AND ITS POKES
30 POKEX,Y:POKEX,Z:GOTO30       : REM RAPID SWITCH BETWEEN MODES
```

first fills the screen with a character which changes when the mode alters; I've put a shifted letter to ensure that the program will work with BASIC 1! Line 30 takes about 1/120th of a second to execute; BASIC 4 machines find this too fast, so slow line 30,

---

*VIC again is different: although the screen is memory-mapped to RAM, the actual location can vary, depending on the amount of RAM installed.

[2]Dr R Chiswell, in SUPA of March 1981, writes that people with certain forms of epilepsy (e.g. temporal lobe epilepsy) may have an attack induced by the program. 'Creative Computing' (Vol.6, #10; dated Oct.'78) has some information on video displays. This journal specialises in, or at least is biased towards, articles on graphics, games, and sound.

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
0	00	@	32	20	£	64	40	96	60	128	80	160	A0	192	C0	224	E0		
1	01	a A	33	21	!	65	41	97	61	129	81	161	A1	193	C1	225	E1		
2	02	b B	34	22	"	66	42	98	62	130	82	162	A2	194	C2	226	E2		
3	03	c C	35	23	#	67	43	99	63	131	83	163	A3	195	C3	227	E3		
4	04	d D	36	24	$	68	44	100	64	132	84	164	A4	196	C4	228	E4		
5	05	e E	37	25	%	69	45	101	65	133	85	165	A5	197	C5	229	E5		
6	06	f F	38	26	&	70	46	102	66	134	86	166	A6	198	C6	230	E6		
7	07	g G	39	27	'	71	47	103	67	135	87	167	A7	199	C7	231	E7		
8	08	h H	40	28	(	72	48	104	68	136	88	168	A8	200	C8	232	E8		
9	09	i I	41	29	)	73	49	105	69	137	89	169	A9	201	C9	233	E9		
10	0A	j J	42	2A	*	74	4A	106	6A	138	8A	170	AA	202	CA	234	EA		
11	0B	k K	43	2B	+	75	4B	107	6B	139	8B	171	AB	203	CB	235	EB		
12	0C	l L	44	2C	,	76	4C	108	6C	140	8C	182	AC	204	CC	236	EC		
13	0D	m M	45	2D	-	77	4D	109	6D	141	8D	183	AD	205	CD	237	ED		
14	0E	n N	46	2E	.	78	4E	110	6E	142	8E	184	AE	206	CE	238	EE		
15	0F	o O	47	2F	/	79	4F	111	6F	143	8F	185	AF	207	CF	239	EF		
16	10	p P	48	30	0	80	50	112	70	144	90	186	B0	208	D0	240	F0		
17	11	q Q	49	31	1	81	51	113	71	145	91	187	B1	209	D1	241	F1		
18	12	r R	50	32	2	82	52	114	72	146	92	188	B2	210	D2	242	F2		
19	13	s S	51	33	3	83	53	115	73	147	93	189	B3	211	D3	243	F3		
20	14	t T	52	34	4	84	54	116	74	148	94	190	B4	212	D4	244	F4		
21	15	u U	53	35	5	85	55	117	75	149	95	191	B5	213	D5	245	F5		
22	16	v V	54	36	6	86	56	118	76	150	96	192	B6	214	D6	246	F6		
23	17	w W	55	37	7	87	57	119	77	151	97	193	B7	215	D7	247	F7		
24	18	x X	56	38	8	88	58	120	78	152	98	194	B8	216	D8	248	F8		
25	19	y Y	57	39	9	89	59	121	79	153	99	195	B9	217	D9	249	F9		
26	1A	z Z	58	3A	:	90	5A	122	7A	154	9A	196	BA	218	DA	250	FA		
27	1B	[	59	3B	;	91	5B	123	7B	155	9B	197	BB	219	DB	251	FB		
28	1C	\	60	3C	<	92	5C	124	7C	156	9C	198	BC	220	DC	252	FC		
29	1D	]	61	3D	=	93	5D	125	7D	157	9D	199	BD	221	DD	253	FD		
30	1E	↑	62	3E	>	94	5E	126	7E	158	9E	200	BE	222	DE	254	FE		
31	1F	←	63	3F	?	95	5F	127	7F	159	9F	201	BF	223	DF	255	FF		

shifted — reversed — shifted

PET/CBM SCREEN MEMORY

for example with redundant spaces, to about 1/100th second. When the program runs, the screen fills with repeats of a character - this tedious part can be speeded up in various ways, e.g. by printing fewer, longer strings - then enters an infinite loop. A band, corresponding to the times when the character set is changed by a poke from line 30, but is not yet changed back, appears on the screen. Close examination of individual characters shows they are partly made up from shift-A, and partly from the spade graphics character. (I am referring to those characters which form the boundary of the bands). How does this happen? Line 30 runs in synchronization with the screen refresh. That is, every 60 times per second, or 50 with the 8032, the screen finishes scanning (the scan reaches the bottom) and an interrupt is generated, the identical interrupt to that which drives the keyboard input processing. After a short period of flyback, the screen is refreshed, scanning again from top to bottom. Our BASIC program changes the mode exactly twice during one complete scan, so the screen is separated into bands. If the timing of BASIC slows slightly, the bands will start to roll up, and vice versa. (Try pressing a key). The analogous process in machine-code works on the following lines: there are 8*25 = 200 lines of dots, all of which are scanned in (say) 1/50th second. Therefore one row of dots takes about 1/10 000 th second to be refreshed on the screen. This interval allows 100 clock pulses to occur, so a machine-code routine can replace some of the characters (not all 40 or 80!) before the next line of dots is scanned. The process must be repeated with each screen refresh to give a static image.

Alternative character-generating ROMs to those supplied can be made in EPROM form fairly cheaply, although in practice there does not seem to be much demand for them. The entire 'graphics mode' set can be changed, so that POKE 59468,12 makes the alternatives (e.g. Prestel) available, while retaining 'lower case mode' for use

when the system is loading and running normal programs.

PRINT, POKE and PEEK   The double usages of the screen memory and ASCII can be very confusing. It may be some consolation to recall that Commodore themselves seem to have been confused when designing the software for their printers. Let's start with the screen memory. We've seen that POKE puts a character on the screen, and that each poke corresponds to a character. In fact, there is a one-to-one relation, with the sole exception of space (CHR$(32)) and shift-space (CHR$(160)) which look alike but PEEK differently. There is no ambiguity about the screen: provided we know which mode it is set to, and provided we're not worried about shifted or unshifted spaces, the character's appearance tells us the value we shall peek from its RAM location. This diagram shows the relationship between the keys pressed and the screen:

BIT 7	BIT 6	BITS 5,4,3,2,1,0
1=REVERSED	1=SHIFTED	REPRESENT ANY CHARACTER
0=UNREVERSED	0=UNSHIFTED	FROM 0-63 IN SCREEN RAM

Because of this, bit 7 simply needs to be reversed to switch a character from reversed to unreversed and back. For example, EOR #$80 has this effect in CBM graphics. Not all computers use this system. Apple high resolution graphics use EOR #$FF, since every bit has to be reversed. Bit 6 selects either the unshifted or shifted part of the character set. In this case, EOR #$40 shifts and unshifts a character alternately. The remaining bits offer $2^6 = 64$ combinations, which are roughly divided into the alphabetic, numeric and punctuation symbols, and graphics. There is a maximum of 64 graphics; but reversing each of these extends the set, and in fact is the only way to complete some of the graphics subsets, as we shall see.

PRINTing has to convert ASCII characters into the screen form. True ASCII reserves the first 32 characters for control information, and PET took this idea over, although many PET screen editing functions bear little relation to any ASCII function. So, because of the screen memory arrangement, PRINT CHR$(65) or PRINT "a" has to poke 1 into the screen. To do this it simply drops the 6th bit from the ASCII value, after testing whether the character to be printed might be a control character. This is the reason for the repeats in the table of CBM 'ASCII' characters, two pages before this. Note that a reversed character cannot simply be printed; it must be preceded by the [RVS] key, or, what amounts to the same thing, the reverse flag must be set, $9F ($020E in BASIC 1) holding a non-zero value. This can be irritating when a graphics set is saved from the screen as strings: one of the manuals demonstrates graphics with a rocket which is drawn on the screen, then saved as BASIC by homing the cursor and typing 10" [return], 20" [return], and so on. What they don't say is that reversed characters, which are often necessary for a full effect, can't be saved in this way in a string. Instead the string must be punctuated with [RVS] and [RVSOFF] symbols.

How does PRINT work? Chapter 5 has information on this; we need consider only output to the screen. Typically this involves the kernel routine $FFD2, 'OUTPUT A CHARACTER'. This first checks for the device number; when 3 (i.e. screen) is found, the current A,X, and Y values are saved, and a set of routines entered, depending on whether the shift key was pressed, and so on. Control characters are tested by routines which are, in wide-screen CBMs, remarkably tortuous. Ordinary characters go by a different route to subroutines which put the correct byte into the screen (at last!) and update the screen cursor postions and so on, before recovering A,X, and Y and returning. It is not surprising that poking the screen direct in machine-code is the fastest way to transfer data to the screen. For this reason, all games and graphics programs, and routines which perform functions like reversing the screen or storing screens in RAM, directly load and save the RAM values in machine-code using load accumulator - store accumulator style operations. So why use the routine to print at all? The answer is: it is convenient. All the screen scrolling, calculation of new lines, cursor homing and movement, is easily done. Poke requires both machine-code and screen-position calculations; BASIC POKEing is not efficient.

The portion of PRINT which puts a byte into the screen differs between 8 inch screen PETs/CBMs and their later counterparts. The difference is in hardware, and it is reflected in the software. The routines (E7AC in BASIC 1, E6EA in BASIC 2, E606 in BASIC 4) are of two types; one waits until bit 5 of $E840 is zero before storing the accumulator's contents in the screen. The other, more recent, type doesn't wait, but slaps the characters in at full speed. (I quote Jim Butterfield). In the first case, the

object of the delay is to wait for the 'retrace interrupt' to be signalled, which means that flyback is taking place. In the earliest PETs the screen was blanked during this interval, and new characters written into the screen memory. When refresh took place, the new characters appeared. Direct poking caused 'snow', because the character generator couldn't tell whether some dots were supposed to be black or white. Hence this type of thing:

```
TAY        ;SAVE BYTE ...
LDA E840   ;AWAIT RETRACE INTERRUPT...
AND #20
BNE -7
TYA        ;AND RESTORE BYTE FOR SCREEN POKE.
```

BASIC 2 has this same formula in it; BASIC 4 has dropped it, at least in the 12 inch versions. A hardware rearrangement made it unnecessary. Because of this, BASIC 4 in the very earliest PETs will give a 'snowstorm' effect (well, it's not really that bad).

    With PRINT, a fair amount of time is wasted if retrace has to be waited for, because only a very small proportion of the scan's time is taken up with the interrupt. In fact, only a tiny number of bytes, perhaps ten, can be fitted into each of these flybacks. The fast-screen POKEs, discovered several years ago, and then recalled from some software when their harmful effects on certain machines became known, work by causing this delay not to happen. POKE 59458,62 speeds up screen writing by a factor of about 6, in BASIC 1 and BASIC 2 machines. This improves the appearance so much that it is pretty well compulsory with BASIC. Unfortunately, the hardware modification which helped take the 3032 into the 8032, and which made this type of poke redundant, is affected by the poke, so that software including the poke *should not be transferred to BASIC 4 machines*. The damage is not immediate; the screen picture collapses or diminishes, and eventually a curl of smoke comes from the machine ...(I quote Jim Butterfield again).

9.2 The CRT Controller chip

12 inch screen CBMs, but not 8 inch, contain a 6845 CRT (cathode ray tube) controller chip. Motorola's MC6845 is designed for raster-scan displays and can be configured for 'almost any' screen density; notably 80 by 25. It includes facilities for cursor control and light pen operation, not used by CBM. The manufacturer's data sheets are informative and include examples of parameter calculations used when initialising the chip. Confining ourselves to Commodore's implementation, we find the two RAM locations, wired to the address register and the register file, at $E880 and $E881 respectively. The first of these is an 'indirect' or 'pointer' register: its contents may be 0-17 (it has 5 bits only) and, depending on the value, the corresponding register, 0-17 is accessed, and a new parameter may be put in it. To see how this works, we find that a jump table in the CBM controls the CRTC. On switching on, the CRTC is initialised by a jump to $E018. This puts the machine into lower-case mode and separates the individual lines of text. The same effect is obtained by PRINT CHR$(14), 'Set text', or of course by SYS 57368 or JSR $E018 from BASIC or machine-code. Graphics mode is initialised by a neighbouring jump, $E01B, which can be performed by PRINT CHR$(142), 'Set graphic mode', or by SYS 57371, or by JSR $E01B.* A further jump table entry, $E01E, should be loaded before it is called with appropriate values for A,X, and Y; it is a user-definable entry point.

    When any of these three routines arrives at $E088, A and X are assumed to point to ROM or RAM, X being the high byte and A the low. The Y-register holds 12 or 14 decimal; this is poked into the location to control upper/lower case and graphics. Then 18 bytes, from the address pointed to up, are poked into the register file; this means that the address register is alternately poked with the register number. On the next page is a listing of a short BASIC program which exactly simulates the action of this. The order, from 17 to 0, mimics that of the machine-code, and the table of 18 bytes is identical to that for lower-case mode. Running the program therefore has no effect when in normal lower-case mode. However, if the screen is set to graphics mode, or lower-case mode with the lines next to each other (e.g. PRINT CHR$(142): POKE 59468,14), the program will make the screen revert to its switch-on appearance. The DATA values for graphics mode are: 0,0,0,0,0,16,0,0,7,0,37,25,0,49,15,41,40,49.

---

*These addresses relate to the 8032; at the time of writing I haven't definite evidence on 12 inch 4032s. In any case it's not hard to find the relevant code; for example, the reset vector can be followed until a reference to locations E880 & E881 is found.

CRT CONTROLLER PROGRAM IN BASIC (80 COLUMN CBM)

```
100 FOR J = 17 TO 0 STEP -1
110 POKE 59520,J
120 READ X: POKE 59521,X
130 NEXT
200 DATA 0,0,0,0,0,16,0,0,9,0,32,25,0,39,15,41,40,49
210 REM     PUT EACH DATA ITEM ON ITS OWN LINE FOR EASIER EDITING
```

There is little difference in setting for each mode. (These values are from tables at E72A - E73B and E73C - E74D for lower-case and graphics modes respectively). We can now, using this table of registers as a guide, investigate the chip further. Note that some combinations of inputs produce odd effects with the CRT; I've been told that it's unlikely that damage could result, but nevertheless the high-voltage equipment which operates the tube can emit noises of the sort you'd rather hear on other peoples' machines.

REGISTER #	REGISTER FILE DESCRIPTION	CBM LOWER CASE/	UPPER CASE VALUES	REGISTER BITS
0	HORIZONTAL TOTAL	49 ($31)	49 ($31)	0-7
1	HORIZONTAL DISPLAYED	40 ($28)	40 ($28)	0-7
2	HORIZONTAL SYNC POSITION	41 ($29)	41 ($29)	0-7
3	HORIZONTAL SYNC WIDTH	15 ($0F)	15 ($0F)	0-3
4	VERTICAL TOTAL	39 ($27)	49 ($31)	0-6
5	VERTICAL TOTAL ADJUST	0 ($00)	0 ($00)	0-4
6	VERTICAL DISPLAYED	25 ($19)	25 ($19)	0-6
7	VERTICAL SYNC POSITION	32 ($20)	37 ($25)	0-6
8	INTERLACE MODE	0 ($00)	0 ($00)	0-1
9	MAXIMUM SCAN LINE ADDRESS	9 ($09)	7 ($07)	0-4
10	CURSOR START	0 ($00)	0 ($00)	0-4*
11	CURSOR END	0 ($00)	0 ($00)	0-4
12	START ADDRESS (HIGH)	16 ($10)	16 ($10)	0-5
13	START ADDRESS (LOW)	0 ($00)	0 ($00)	0-7
14	CURSOR (HIGH)	0 ($00)	0 ($00)	0-5
15	CURSOR (LOW)	0 ($00)	0 ($00)	0-7
16	LIGHT PEN (HIGH)	0 ($00)	0 ($00)	0-5
17	LIGHT PEN (LOW)	0 ($00)	0 ($00)	0-7

Note that register 1 holds the horizontal display; this is 40, not 80, as might be expected. If this register is changed to say 41, by POKE 59520,1: POKE 59521,41 then the text will slope diagonally to the left, and the cursor moves in a crablike diagonal direction. A program called 'CBM 4032 C CHEE' reconfigures the 8032 as for a 40-column machine by (I presume) putting 20 into register 1. Register 9 controls the number of scans given to each character. 7 puts no blanks between adjacent lines; 9 puts 2. A value of 6 in this register loses the bottom line of dots, including lower-case decenders. A short program of the following type can be used to watch the effect of any register:

```
10 POKE 59520,R :REM REGISTER NUMBER; CHOOSE 0 - 17
20 FOR J = 0 TO 128 :REM OR CHOOSE OTHER LIMITS; TABLE ABOVE INDICATES MAXIMA
30 POKE 59521,J: GET X$: IF X$="X" THEN PRINT CHR$(14): END
40 GET X$: IF X$<>" " GOTO 40
50 NEXT
```

Each press of the space-bar will change the register-value. Entry of 'X' at the keyboard provides an emergency exit (just in case!) and returns to normal.

Register 6 controls the number of lines printed; if these exceed 25, garbage appears from higher up the screen memory. Register 13 alters the start address- the screen shifts left with wraparound. Register 12, holding 12_{10}, reverses the screen! So POKE 59520,12: POKE 59521,12 reverses the screen. Registers 4 and 7 between them control the position of the characters on the screen, rather like horizontal hold on a TV. I have been unable to get characters to come out in reverse, although I suspect this may be possible.

*Bit 5 sets blink period control, and bit 6 sets blink/non blink.

9.3 Graphics and the PET/CBM

BASIC graphics Commodore has retained its set of graphic characters in all its machines, including VIC. As we have seen, there are 128 graphics in total, or more if some of the alphanumerics or punctuation symbols are included. They have not added a high resolution facility, though hardware is commercially available which will do this. Some other machines have upgraded to high resolution graphics, such as the newer Sharps; and some, notably Apple, have had high resolution, and colour, of a sort, all along. PET/CBM users generally have to do without elaborate graphics; Commodore has allocated such activity to its VIC evolutionary branch. CBM graphics displays usually have a Prestel-like quality, apparently being made of a jigsaw of little bricks. The character-generation also has an annoying defect, causing some adjacent blocks of reversed characters not to connect properly, leaving small lines. Nevertheless there are advantages in the CBM approach. Unlike external TV sets, the monitor picture is stable. Moreover the graphics are fast, since only one or two thousand fill the screen. Apple high-resolution pictures need 8K of RAM storage. It is also possible to replace the character-generator with a fast EPROM containing (say) Prestel characters or 128 user-defined graphics (the other 128 being their reversed forms).

'Cross-reference to CBM graphics characters' - see the table on the following page - groups the individual graphics in a helpful way. There seems to be no method in the ordering of these characters in ROM. Note that, when reverse is taken into account, the sets of graphic type are all complete, with the exception of the shaded blocks.

Programmers unused to the graphics set, or perhaps looking for ideas on how best to combine graphics characters, could to worse than experiment with a short BASIC program like this:

```
10 POKE 59468,12: INPUT "LOWER CASE SET (Y/N)"; YN$: IF YN$="Y" THEN POKE
   59468,14
20 INPUT "CHARACTER STRING"; X$
30 FOR J=1 TO 5: X$=X$+X$: NEXT: REM INCREASE LENGTH OF STRING FOR SPEED
40 FOR J=1 TO 1000/LEN(X$): PRINT X$;: NEXT: RUN
```

This accepts input of a short string of graphics characters, then fills the screen with repetitions of the string. Interesting optical effects can be found; for example, the diagonally-shaded squares in lower-case mode produce a herringbone pattern which displays a well-known optical illusion. A string with format xxxyzzy may work well, and so on. Obviously, if the input string is 2 or 4 characters long, the result will be rectilinear, since each line will exactly repeat the previous line. Otherwise, the pattern naturally has a diagonal symmetry. Note that the program is designed for a 40 column machine; line 40 prints about a thousand individual characters. The 8032 is less amenable to graphics than its earlier counterpart; to display the full range, input a set of numbers (see the table) and convert them by CHR$ into a printable graphic equivalent. Also change the constant in line 40 to 2000. Business keyboard users will find a number of graphics hard to obtain; the rather useful lines, permitting boxed formats, for example like the screen pictured below, which is authentic apart from the figures, aren't obtainable by keyihg-in. This is because shift-! through shift-? in the ASCII table have been combined typewriter-fashion: shift-1 becomes ! rather than a T-shaped connecting graphic symbol.

CROSS-REFERENCE TO CBM GRAPHICS CHARACTERS

KEY:	sh-$	sh-r	sh-f	sh-@	sh-c	sh-d	sh-e	sh-#
CHR$:	228	210	198	192	195	196	197	227
POKE:	100	82	70	64	67	68	69	99

KEY:	sh-%	sh-t	sh-g	sh-b	sh-]	sh-h	sh-y	sh-'
CHR$:	229	212	199	194	221	200	217	231
POKE:	101	84	71	66	93	72	89	103

KEY:					REVERSE, CHR$(18), THEN-			
KEY:	sh-$	sh-/	sh-9	sh-"	sh-8	sh-7	sh-#	sh-space
CHR$:	228	239	249	226	184	183	163	160
POKE:	100	111	121	98	248	247	227	224

KEY:					REVERSE, CHR$(18), THEN-			
KEY:	sh-%	sh-4	sh-5	sh-!	sh-6	sh-*	sh-'	sh-space
CHR$:	229	244	245	225	182	170	167	160
POKE:	101	116	117	97	246	234	231	224

KEY:	sh-o	sh-p	sh-:	sh-1	sh-v	sh-[	sh-m	sh-n
CHR$:	207	208	186	204	214	219	205	206
POKE:	79	80	58	76	86	91	77	78

KEY:	sh-=	sh--	sh-Ø	sh-.	sh-1	sh-2	sh-3	sh-+
CHR$:	189	173	176	174	177	178	179	171
POKE:	125	109	112	110	113	114	115	107

KEY:	sh-<	sh->	sh-,	sh-;	sh-?			
CHR$:	190	188	172	187	191			
POKE:	126	124	108	123	127			

KEY:	sh-k	sh-j	sh-u	sh-i	sh-w	sh-q		
CHR$:	203	202	213	201	215	209		
POKE:	75	74	85	73	87	82		

KEY:	sh-)	sh-←	sh-&	sh-(	sh-\			
CHR$:	169	223	166	168	220			
POKE:	105	95	102	104	92			

KEY:	sh-a	sh-s	sh-z	sh-x	sh-↑			
CHR$:	193	211	218	216	222			
POKE:	65	83	90	88	94			

NOTES: (i) There are ambiguities in many of the CHR$ figures - CHR$(227) or CHR$(163) might equally well be chosen. I've preferred the values with a constant difference of 64 or 128 from the screen POKE/ PEEK value.

(ii) As the characters are made of 8 dots by 8, a line cannot appear exactly in the centre of any character; some characters, when positioned as neighbours, will not exactly line up together.

(iii) The table has more than 64 entries, because some appear twice. Note that the lower-case mode special graphics – chequered square, diagonally shaded squares, and square-root or tick sign – have not been included. The full 128 graphics characters are obtained by reversing all those in the table, by PRINTing the reverse character first, or by POKEing the values listed here + 128.

'Digital clock' is a reasonably short specimen graphics program; it converts TI$ into larger characters, modelled on a 7-segment LED or liquid-crystal display. Two type-faces, ordinary and 'modern', are available! This program is in BASIC and is quite slow, so that 1 second is sometimes not enough time to allow the time to be updated, so it will skip a second. Also, of course, the program does nothing else but 'draw' the clock. The subroutine starting at line 10 draws a single numeral; it can be used to print any other numerals, where a display somewhat larger than normal is wanted, on a similar basis.

Briefly, we wish to simulate diagram 1. A complete square is not available in the graphics set, so a layout like that in diagram 2 is needed, in which 5 adjacent squares control the final appearance. I've used the order D,B,A,C,E,F of diagram 3; in this way, the cursor is always left in a position to start the next number, if there is one.

Diagram 1   Diagram 2   Diagram 3

If we make up a table with 10 rows, one for each digit, we can write out the ASCII values required in segments A - E. Zero for example needs A, B, C, D, and E, or, in ASCII values, 164,165,165,204 and 165. The five lines of subroutine in the program each process one of these segments, using a logical formula to embrace every value of numeral X.

A decimal point can be allowed for;

5000 IF MID$(X$,J,1)="." THEN PRINT "▓";: GOTO 5020   shows the sort of thing.

DIGITAL CLOCK

```
0 GOTO 1000
10 PRINT CHR$(204 + 172*(X=1 OR X=4 OR X=7) + 40*(X=3 OR X=5 OR X=9)); "[LEFT][
UP]";
20 PRINT CHR$(204 + 172*(X=1) + 40*(X=2 OR X=3) + 39*(X=0 OR X=7)); "[LEFT][UP]
";
30 PRINT CHR$(164 + 132*(X=1 OR X=4)); "[DOWN]";
40 PRINT CHR$(165 + 133*(X=5 OR X=6)); "[LEFT][DOWN]";
50 PRINT CHR$(165 + 133*(X=2));
60 RETURN
1000 PRINT "[CLR][DOWN]"
1010 X$ = TI$
1020 FOR I = 1 TO 6 STEP2
1025 FOR K = 0 TO 1
1030 X = VAL( MID$(X$,I+K,1) ); GOSUB 10
1040 NEXT K: PRINT" "; NEXT I
1050 PRINT "[HOME][DOWN]"
1060 GOTO 1010
READY.
```

SIMILAR ROUTINE WITH COMPUTER-STYLE TYPEFACE:

```
10 PRINT CHR$(204 + 172*(X=1 OR X=4 OR X=7) + 29*(X=3 OR X=5 OR X=9)); "[LEFT][
UP]";
20 PRINT CHR$(204 + 172*(X=1) + 40*(X=2 OR X=3) + 24*(X=0 OR X=7)); "[LEFT][UP]
";
30 PRINT CHR$(164 + 132*(X=1 OR X=4)); "[DOWN]";
40 PRINT CHR$(180 + 148*(X=5 OR X=6)); "[LEFT][DOWN]";
50 PRINT CHR$(180 + 148*(X=2));
60 RETURN
```

Special features of BASIC 4 screen handling BASIC 4 machines are more complex than their predecessors. There are currently three distinct models: the 8032, which has 80 columns; the 8" screen 4032 (and 4016, 4008), which is no longer manufactured, and the 12" screen 4032 (and 4016, 4008). The 80-column machine has the most features, largely because a chunk of RAM previously used for screen-line tables is not needed. This enables it to be equipped with several features not obtainable on any other machines, even those with BASIC 4, notably a definable screen 'window'. These differences are made possible by varying the ROM which deals with E000 - E7FF. The presence of the CRT controller chip distinguishes 12" screen machines from the earlier 8" models.

Text/ graphics 12" machines have machine-code which *both* sets the CRT and puts the character-generator into the appropriate mode. For 'graphics', the lines are moved so as to be adjacent, and upper-case/ graphics mode is selected; in 'text', or 'lower-case' mode, the process is reversed. PRINT CHR$(14) selects text mode; the value 14 was presumably suggested by the use of POKE 59468,14 to select lower-case mode. PRINT CHR$(142) sets graphics. The ROM routine can be called directly, with SYS 57368 or SYS 57371. The [ESCAPE] key allows yet another variation; 'n' is the 14th letter of the alphabet, so PRINT "[ESCAPE]UNSHIFTED-N has the effect of PRINT CHR$(14), and PRINT "[ESCAPE][RVS]UNSHIFTED-N acts like PRINT CHR$(142).

8032 screen window One screen window only may be defined at any one time; however the redefinition time is small, so there is little difficulty in apparently generating such windows at will anywhere on the screen. When a window is defined, four RAM parameters are set: locations $E1 and $D5 (225 and 213 in decimal) hold the screen positions of the bottom and right of the window; $E0 and $E2 (224 and 226) hold the top and left. The top and left parameters have a minimum value of zero, corresponding to the topmost line and leftmost screen position. The bottom has a maximum of 24, and the right a maximum of 79. If these values are exceeded, garbage will appear on the screen when it scrolls, and information at the rightmost end of lines will be lost. If the bottom parameter is less than or equal to the top, or if the right is less than or equal to the left, a single row or column only is printable. A window must have a certain minimum width to be usable. For example, RUN[RETURN] needs at least 4 columns. A narrower window will not permit the command RUN to operate.
    The parameters may be poked into the locations given, or the specially allocated characters may be printed: PRINT CHR$(15) makes the cursor's current position into the window's top-left, and PRINT CHR$(143) sets the bottom-right.
    A window is erased by two consecutive [HOME]s, which are counted in RAM location $E8 (=232 in decimal).

40 to 80 column interconversion 40-column programs can be run on the 80-column machine by redining the screen with the CRT chip, as previously mentioned. Generally the reverse process is impossible, because 80-column software requires 2000 bytes of screen RAM, which the smaller-screen machines do not have. However, by editing the larger-screen output, provided its total storage requirement isn't too large, conversion to the smaller format may be possible. 40-column software may run without modifications on 80-column machines: if only PRINT statements are used, and if these are terminated with carriage returns, the output will align itself down the left half of the screen. When PRINT is followed by ';' ,relying on the end of the screen to force printing on the next line, its line will extend across the screen. Direct pokes into screen RAM fill the top half of the screen. Either of these latter possibilities can be avoided by CRT reconfiguration.

Other screen-editing characters in BASIC 4 Apart from TABs, BASIC 4's new screen editing commands are:
PRINT CHR$(21)	Delete line from screen	PRINT CHR$(149)	Insert line into screen
PRINT CHR$(22)	Erase line up to end	PRINT CHR$(150)	Erase line from start
PRINT CHR$(25)	Scroll screen up	PRINT CHR$(153)	Scroll screen down

These may be easier to use if a string is defined to store the appropriate characters; for example, SD$=CHR$(153) is mnemonically helpful in PRINT SD$.

Using TABs PRINT CHR$(137) sets a tab position; PRINT CHR$(9) moves the cursor to the next tab position, or to the end of the line. However, if a tab is already set at that position, PRINT CHR$(137) unsets it. Tab data is stored in ten bytes of RAM, just below BASIC. (See Chapter 15's RAM map). Each of the eighty bits may be 1 or 0; and 1 denotes a tab setting. The first byte stores tabs for columns 0-7, the second for 8-13, and so on. However, the bits are arranged in the reverse order, so poking the first byte with (say) 2 sets a tab near the left of its set of columns.

Machine-code graphics As we've seen, BASIC is liable to be slow when dealing with graphics. In this section we'll look at machine-code graphics, which enable graphics effects to be realised in a far faster manner. This will have to be skipped by those not yet familiar with machine-code; nonetheless many of the examples can be entered and run by inexperienced programmers.

Machine-code is chiefly used (with graphics) to put characters directly into screen RAM, rather than PRINTing them with FFD2 or a similar output routine. The beginner, to understand this idea, can enter this short program into the machine with the monitor:

```
Enter SYS 4; the monitor now displays the program counter and other registers.
Enter M 033A 0342; two lines from the machine (16 bytes in all) are printed in
two lines. This is the start of the second cassette buffer; in BASIC<4 it is
inviolate, unless the second external cassette port is used; in BASIC 4 it
stores some disk data, but in our example this isn't important. Now type in:
M 033A A2 00 8A 9D 00 80 E8 D0
M 0342 F9 60                        the remaining symbols are unimportant.
Enter X to return to BASIC.
```

$033A (=826) is now the starting-point for the following machine-code:

```
LDX #00     ;load register X with value zero
TXA         ;transfer contents of X to A
STA 8000,X  ;store accumulator contents into address $8000 + offset in X
INX         ;increment X
BNE -7      ;if X is non-zero, branch back 7 (count from RTS) to TXA
RTS         ;return when X is zero - after 256 loops.
```

Now SYS 826 pokes 256 values, from 0-255, into the top of the screen. They should correspond to the table of screen poke values a few pages back. Note the greatly increased speed with which characters are printed to the screen. This program is short because its values are computed; they needn't be looked up. The next example pokes the word 'hello' into the screen, in lower- or upper- case depending on the mode:

```
M 0350 A2 (04) BD 5B 03 9D (23 81)      04 = no. of chrs. to print -1.
M 0358 CA D0 F7 60 (08 05 0C 0C         23 81 = start address in screen = $8123  here.
M 0360 0F    -- any --                  = Table of bytes.
```

When this routine is entered, SYS 848 prints 'hello', starting at $8123. The table of 5 bytes holding 'hello' appears after 60, which is the RTS opcode. Again, the beginner is recommended to try this; it is quite easy to understand. The values ringed can be changed freely, and the result examined.

Screen reversal and flashing. These effects are easy to get in machine-code, subject to the usual problem of managing RAM so that the routine doesn't occupy space taken up by other routines. We have already seen that the high bit of screen RAM characters detemines whether the character is reversed or not. To reverse an entire screen, therefore, all we need to do is scan the entire screen RAM, replacing every character by its equivalent with the high bit reversed. If we repeat the process the screen will return to its previous condition. Note that this method reverses *all* characters; if they are reversed already, our routine will return them to the unreversed state.

```
        LDA #80
        STA 01
        LDA #00
        STA 00      ;indirect address (00) points to $8000 now
L1      LDY #00     ;this loop processes one page (256 bytes) of screen
L2      LDA (00),Y  ;this loop uses Y to count from 0-255
        EOR #80
        STA (00),Y  ;poke reversed value back into the screen
        INY
        BNE L2
        INC 01      ;indirect address (00) points to $8100, $8200, etc.
        LDA 01
        CMP #84     ;stop when $8400 reached; 80-column machines use #88
        BNE L1
        RTS
```

This machine-code (not quite identical - a tighter version) reverses the screen:

```
M 033A A0 00 84 01 A2(83)86 02    ——>87 for 80-column machine.
  0342 B1 01 49 80 91 01 88 D0
  034A F7 CA 30 F2 60 -any---
```

So SYS 826 reverses the screen (in 1/50th second or 1/25th with 80-columns), and 10 FOR J = 1 TO 10: SYS 826: NEXT flashes the screen ten times, leaving it unreversed. This code is relocatable (i.e. can be entered unchanged anywhere in RAM); locations 1 and 2 are used, so USR jumps will no longer work until their correct jump address is loaded.

Is it possible to reverse only part of the screen? This too is quite easy. Let's suppose we have the starting address (e.g $8050) of the region to be reversed, and let's suppose we want a certain number of bytes (<256) after the starting address to reverse; for example, 40 or 80 bytes will reverse one line of text. The following short routine, also relocatable machine-code (for a change, I've put it at the start of the first cassette buffer, starting $027A), and demonstration BASIC driver program is one way of doing this:

```
M 027A A4 00 88 B1 01 49 80 91
  0282 01 98 D0 F6 60 --any--
```

```
1000 POKE 0,20            :REM 0 HOLDS NUMBER OF CHARACTERS, SAY 20, AS HERE
1010 POKE 1,80: POKE 2,128 :REM (01) POINTS TO $8050
1020 SYS  634             :REM REVERSE 20 CHARACTERS FROM $8050 ON
```

Different effects can be obtained by modifying the core of all these routines, which is   LDA from an address/ EOR #80 to switch the high bit/ STA back into address. For example, AND #7F unreverses the entire screen; ORA #80 puts every character in reverse form; EOR #40 switches all shifted characters to unshifted, and vice versa; INC address (e.g. INC(01),Y /NOP /NOP/ NOP/ NOP) replaces every character by the next character in the screen RAM table.

Note that all these examples use purely software methods. When the screen scrolls, the reversed text will scroll up with it, leaving normal text.

<u>Using switches</u>. With the help of the interrupt routine, we can call up machine-code routines with a simple poke; this provides a convenient way to call a batch of routines. For example, we may have ten stored screens of data; poking a preset location with 1-10 can automatically display any of them. To show the method, I'll assume that the routine at the top of this page is present in RAM. We can use 034F as the key location. Then enter:

```
M 0350 AD 4F 03 D0 03 4C 55 E4      BASIC 2: 2E E6, BASIC 1: 85 E6
  0358 20 3A 03 10 F8 --any--
```

Now enter .R, to display the registers, and change IRQ from E455 or E62E or E685 to 0350. The IRQ vector isn't changed until a GO command, so enter .G 0004 to BRK. Now, this machine-code is processed at every interrupt:

```
0350 LDA 034F
0353 BNE 0358
0355 JMP E455; OR E62E OR E685
0358 JSR 033A;executes subroutine at 033A, in this case screen reverse
035B BPL 0355;unconditional branch, because of 033A's method of operation.
```

When $034F (=847) is POKEd with any non-zero value, the screen is reversed at each interrupt; therefore the screen flashes, and processing also slows down a great deal because there are 50 or 60 interrupts per second, and our routine takes 1/25th or 1/50th of a second to run! POKE 847,0 stops the flashing. The routine at $0350 is easy to extend so that it perhaps reverses the screen once, or reverses it n times when n is poked into $034F. The real point, however, is the relative ease by which subroutines can be called like this; this is as true in machine-code as BASIC. A value of 0 in a location might signal that nothing is to be done; values from 1-16 might draw object number 1 in any of sixteen preset positions on the screen; values of 17-31 might do the same for another object, and so on. If necessary the interrupt can test several locations. 'Space Invaders' in its PET/CBM version uses a technique like this. Having seen a few methods at work, we can consider some of the published work on graphics.

Published utilities. 'Compute!' (March '81; Vol.3, #3) has a six-page article by D Malmberg including machine-code (long) and BASIC enabling one rectangle at a time to be filled with a character, reversed, or made to flash; and also to be shifted elsewhere bodily, shifted continuously or 'made to grow or shrink in size'. This last feature means that, as a rectangle moves, its previous incarnations are left on the screen, not erased. D Simons (CPUCN Vol.3, #2) has a long routine, for BASIC 2, which intercepts BASIC, providing various screen facilities, including the interchange of two screens in RAM, and vertical-bar graphics. 'Printout' (Jan.'81) has a few routines, taken from 'PET User Notes'. CCN (Vol.3, #8) has some machine-code, without, however, instructions on using it. Probably, if you have access to back issues of journals, you can find more. Don't expect much though; you may be disappointed. A different style of screen image processing is represented by 'systematic routines', as I've called them, for lack of a better name. These include 'SET', which plots double-density dots on the screen (see Chapter 5).

Systematic machine-code utilities   Three pages before this you'll find a table entitled 'Cross-reference to CBM graphics characters'. From the layout, notably of the topmost rows, it is clear that the completeness of the graphics character sets enables some progress to be made towards high resolution graphics. SET (in Chapter 5) exploits the fact that all sixteen combinations of squares with internal quadrants exist on the PET/CBM; the machine-code has a table of the appropriate values built in. Similarly, a CBM manual has a demonstration program including a histogram of US national income, in which the horizontal bars include, at the end, fractions of a square, as listed in the fourth row of the graphics chart. D Simons' 'Super BASIC' (see reference above) includes a routine to plot vertical bars in the same way. And some EPROMs, e.g. the 'PicChip', include routines to approximate curved lines with short segments, taken from the first and second rows of the chart. To show the methods which such programs use, let's write a routine to plot vertical bars in histogram-fashion, to the nearest 1/8th of a square, i.e. including for 0-7 rows of dots on top of each column of solid characters. For this, the third row in 'Cross-reference' is needed. I shall assume that the starting-point in the screen (e.g $83C0, the bottom-left of a 40-column screen) is stored in the two bytes ($01), and that the height of the column is stored in $00, so that a 'height' of 20 means 2 solid squares (making 16 rows of dots) topped by CHR$(226), adding the final 4 rows of dots.

```
        LDY #00
   L1   LDA 00
        CMP #08
        BCC +23     ;exit routine when this value is zero to seven at L2
        SBC #08
        STA 00
        LDA #A0
        STA (01),Y; put reverse-space into screen
        LDA 01
        SBC #28     ;subtract 40₁₀. 80-column machines use #50 instead.
        STA 01
        LDA 02
        SBC #00
        STA 02
        BCS L1      ;unconditional branch back to continue
   L2   TAX         ;A and X hold value 0-7, which now becomes the table's offset
        BEQ L2      ;don't plot if zero...
        LDA TABLE,X;otherwise, load Xth value from table (holds row 3 poke values).
        STA (01),Y;store final few rows of dots
   L2   RTS
        TABLE #64 #6F #79 #62 #F8 #F7 #E3  ;in decimal,100,111,121,98,248,247,227
```

The routine relocates, and works with all BASICs (in graphics mode). It looks like this:-

```
M 027A A0 00 A5 00 C9 08 90 16
  0282 E9 08 85 00 A9 A0 91 01
  028A A5 01 E9 (28) 85 01 A5 02        ➤ 50 with 80-column machines.
  0292 E9 00 85 02 B0 E4 AA F0
  029A 05 BD A0 02 91 01 60 64
  02A2 6F 79 62 F8 F7 E3 -any-
```

So   POKE 0,H:POKE 1,P: POKE 2,131 ;SYS 634   draws a column of height H on $8300 + P.

Using 'SET'. The disassembled version of SET is too long for inclusion here; its method, briefly, involves (i) Calculating the position in the screen which corresponds to X-coordinate and Y-coordinate; (ii) Loading the character at that screen position; (iii) Modifying it, by ORing the offsets in the table of 16 characters together; (iv) Replacing the character with a new one, with one quadrant changed (or nothing changed if the dot already exists).

Instructions for ROM modifications and relocation appear in Chapter 5. Note that the X- and Y-coordinates, stored in $00 and $01, are changed in the course of the routine's execution, and must be entered afresh for each plot. Any characters in the screen which are not in the lookup table (i.e. everything except the sixteen graphics characters which are part of the set of quadrants) causes nothing to happen, so that leetering on graphs is ignored. Obviously, because locations 0 - 2 are utilised, it is necessary if USR subroutines are called from BASIC to poke the jump byte ($4C=76) into 0, and the indirect address into (01). Its worth noting that the character generating ROM has defects which show up in this instruction. Reverse- shift-> , shift-; and shift-? do not abut correctly to reverse-space, leaving little vertical lines.

Demonstration programs. 'Conic sections' is a mathematical routine, which draws a conic section from 6 general parameters. It includes a subroutine to enlarge and reduce the scale on which the conic section is plotted; the limits on the X-axis are shown on the screen. In this way (with luck) a conic section can be viewed at a reasonable scale. It is interesting to see how the two branches of a hyperbola appear, when reduced in scale, as intersecting straight lines. Some equations, of course, are incapable of being plotted, having only imaginary solutions. The program simply calculates 80 solutions to a quadratic, scanning from left to right, and plotting the result if it exists, and fits into the current screen's limits. Line 508's function is the sign; its function is mainly cosmetic. The scaling factors have to convert any range of X values into 0 - 79. For example, X limits of 20 to 50 are transformed so that X=20 becomes 0, X=50 becomes 79; at 1/10 scale the transformation is recalculated so that -115 to 150 is the range for which X becomes 0 - 79. An 80-column machine requires a few changes, to lines 10 and 510.

DOUBLE DENSITY CONIC SECTIONS PLOTTER   Feb 81

```
0 PRINT "[CLEAR][REVS]NOTE[RVSO]   NEEDS 'HIRES $033A' TO BE LOADED": WAIT 158,1: RE
    M WARNS
2 GOTO 500
10 FOR I = 0 TO 79: X = X1 + I*SF
20 IF C=0 THEN A1=FNA1(X): A2=FNA2(X): Y=-A2/A1: GOTO 56
30 A1 = FNA1(X)/C
31 A2 = FNA2(X)/C
35 IF A1*A1 < 4*A2 THEN NEXT: GOTO 110: REM FASTER THAN 'GOTO 100'
40 A3 = SQR(A1*A1-4*A2)
50 Y = (-A1+A3)/2.6 :REM DIVISOR 2.6 IS A SCALE CORRECTION (FOR ROUND CIRCLES!)
51 Y =25 + Y/SF
52 IF Y>1 AND Y<50 THEN POKE 0,I: POKE 1,Y: SYS 826
55 Y = (-A1-A3)/2.6: REM THIS IS THE OTHER SOLUTION O THE QUADRATIC
56 Y = 25 + Y/SF
58 IF Y>1 AND Y<50 THEN POKE 0,I: POKE 1,Y: SYS 826
100 NEXT
110 INPUT"[HOME][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][DOW
    N][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN][DOWN]0 (EXIT)/ ENLARGEMENT
    FACTOR";A1
115 IF A1 = 0 GOTO 500
120 S2 = (X2+X1)/2: S3 = (X2-X1)/2
130 X1 = S2-S3/A1: X2 = S2+S3/A1: GOTO 510
500 PRINT"COEFFICIENTS OF A,B,C,D,E,F :":INPUT A, B, C, D, E, F
502 INPUT "INITIAL LIMITS OF X1,X2 ARE";X1, X2
504 DEF FN A1(X) = (B*X + E)
506 DEF FN A2(X) = (A*X*X + D*X + F)
508 DEF FN SG(X) = - (X<0)*45 - (X>=0)*43
510 SF = (X2-X1)/79 : REM SCALE FACTOR
512 PRINT "[CLEAR]"; A; "[LEFT]X^2"; " "; CHR$(FNSG(B)); " "; MID$(STR$(B),2); "XY";
513 PRINT " "; CHR$(FNSG(C)); " "; MID$(STR$(C),2); "Y^2"; " ";
514 PRINT CHR$(FNSG(D)); " "; MID$(STR$(D),2); "X"; " "; CHR$(FNSG(E)); " ";
515 PRINT MID$(STR$(E),2); "Y = "; -F
518 PRINT "FOR X="; SGN(X1)*INT(ABS(X1)+.1); "TO"; SGN(X2)*INT(ABS(X2)+.1); ":"
520 GOTO 10
```

    Extensions. An obvious addition to SET is a command to plot 'straight lines', i.e. small squares as nearly straight as the resolution will allow. There is insufficient space here to go into details. (CPUCN Vol.3,#2 has a 40-col. routine by D Middleton). The algorithm is reasonably straightforward; something like this is required:

    (i) arrange the end-points a,b and c,d so that a is less than or equal to c.

    (ii) Test a=c and b=d; if true, plot one point only and exit.

    (iii) Is ABS(d-b) > ABS(c-a)? If so, the gradient, either up or down, is steeper tha 1. Branch to one of two routines depending on the gradient:

    (iv) Gradient<=1. Every horizontal position will have a corresponding point; some verticals may be duplicated, like this:

Plot a,b. Increment a; if it exceeds c, exit.
Calculate the next b = b + x-increment*(d-b)/(c-a).
Go back to plot a,b again.

    (v) Gradient > 1. Test for a vertical line: if found, draw it with its own sub-routine. 'Vertical' includes nearly vertical lines, which otherwise will be too short.

    Increment vertical positions, not horizontals, like this:

Plot a,b. Increment b, calculate the nearest a, and continue until b exceeds d.

    'SET' is slower than it need be: a lookup table for screen lines saves time in performing calculations, but occupies more space. The slowest part of the routine is the search for one of the sixteen characters. With CBM ROMs this process is inevitable because of the unordered arrangement of the relevant graphics screen form. A system in which the screen value corresponded to the graphic's appearance (e.g. POKE 0 giving a blank, POKE 1 a single quadrant in the top left, etc.) would be faster.

    Demonstrations. Finally, a few more simple BASIC demonstration programs. Some of the results are illustrated on the next page.

HIGH RESOLUTION GRAPHICS DEMONSTRATIONS (ASSUMING SYS 826)

```
400 FOR J = 6 TO 2 STEP -1
410 FOR X = 0 TO 79 STEP J: FOR Y = 0 TO 49 STEP J: POKE 0,X: POKE 1,Y
420 SYS 826: NEXT Y, X, J: END
500 FOR J = 60 TO 2 STEP -1
510 FOR X = 0 TO 79 STEP J: FOR Y = 0 TO 49 STEP J: POKE 0,X: POKE 1,Y
520 SYS 826: NEXT Y, X, J: END
600 DEF FN Y(X) = 25 + X*SIN(X/3)/10
610 FOR X = 0 TO 79: Y = FNY(X): IF Y < 0 OR Y > 255 THEN 630
620 POKE 0,X: POKE 1,Y: SYS 826
630 NEXT
640 END
700 DEF FN Y(X) = 25 + SIN(X/10)*COS(X/10)*25
710 FOR X = 0 TO 79: Y = FNY(X): IF Y<0 OR Y>255 THEN 730
720 POKE 0,X: POKE 1,Y: SYS 826
730 NEXT:END
800 DEF FNY(X)=X*J
810 FOR J = 0 TO 2 STEP .2
820 FOR X = 0 TO 79: Y = FNY(X): IF Y<0 OR Y>255 THEN 840
830 POKE 0,X: POKE 1,Y: SYS 826
840 NEXT X,J
850 END
900 INPUT "NO.OF PETALS";M: INPUT "STEP SIZE .01-10";SP
910 IF M = INT(M/2)*2THEN M = M/2
920 FOR TH = 0 TO 100 STEP SP
930 S = SIN(TH*M)+.2
940 X = S*COS(TH):Y = S*SIN(TH)
950 X = 40+30*X: Y = 25+20*Y
960 POKE 0,X: POKE 1,Y: SYS 826
970 NEXT
```

9.4 Dumping PET/CBM graphics to a printer   'DUMP', in Chapter 5, has a program to scan and print the screen to non-CBM printers; graphics characters are printed as '#' or some other symbol, to show that they exist, but cannot be printed. The BASIC routine below will print screen produced by SET to any printer; the top and bottom halves of each character are printed as ' ',' *','* ', or '**'.

LISTING OF 'HI RESOLUTION' PLOTTING PROGRAM

```
0 PRINT"LOAD SCREEN WITH PICTURE TO BE PRINTED IN LINE 00": END
10 OPEN 4,4: CMD4: REM PRINTER OPENED AND READY
20 SCREEN = 8*4096: REM START OF SCREEN
30 FOR V = 0 TO 24: REM SWEEP VERTICAL POSITIONS OF SCREEN
40 FOR H = 0 TO 39: REM SWEEP HORIZONTALLY AND PRINT HIGH HALF OF CHARACTER
50 CH = PEEK (SC+ 40*V + H) : REM ASCII VALUE OF CHARACTER ON SCREEN
60 IF  CH=32 OR CH=98 OR CH=108 OR CH=123 THEN PRINT "  "; :GOTO 100
70 IF CH=124 OR CH=225 OR CH=254 OR CH=255 THEN PRINT " *"; :GOTO 100
80 IF  CH=97 OR CH=126 OR CH=127 OR CH=252 THEN PRINT "* "; :GOTO 100
90 IF CH=160 OR CH=226 OR CH=236 OR CH=251 THEN PRINT "**";
100 NEXT H
110 PRINT
140 FOR H = 0 TO 39: REM SWEEP HORIZONTALLY AND PRINT LOW HALF OF CHARACTER
150 CH = PEEK (SC+ 40*V + H) : REM ASCII VALUE OF CHARACTER ON SCREEN
160 IF  CH=32 OR CH=124 OR CH=126 OR CH=226 THEN PRINT "  "; :GOTO 200
170 IF CH=108 OR CH=127 OR CH=225 OR CH=251 THEN PRINT " *"; :GOTO 200
180 IF  CH=97 OR CH=123 OR CH=236 OR CH=255 THEN PRINT "* "; :GOTO 200
190 IF  CH=98 OR CH=160 OR CH=252 OR CH=254 THEN PRINT "**";
200 NEXT H
210 PRINT
250 NEXT V
260 CLOSE 4: END
```
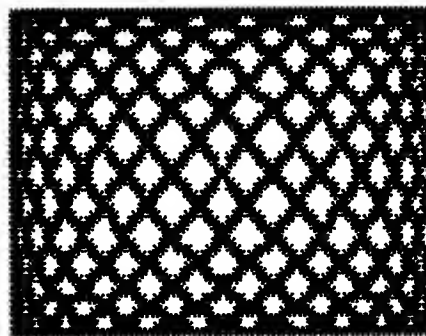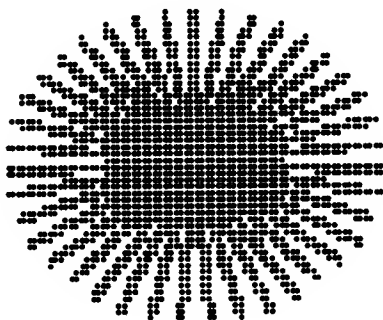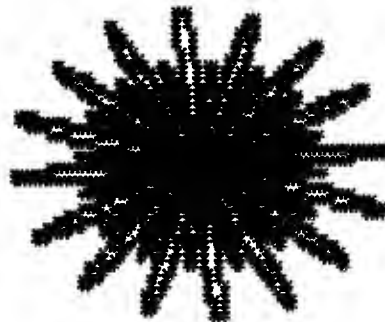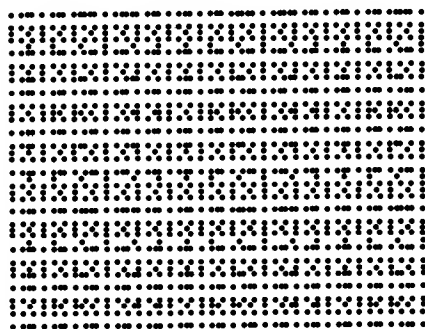
Users with Commodore printers can of course reproduce the entire graphics set on paper. This machine-code program, 'Keyprint', is the BASIC 4 version of a routine which has appeared twice in 'Compute!', in March '81 (Vol.3,#3) for BASIC 1, and in Nov./Dec.'80 (Vol.2,#4) for BASIC 2.

'KEYPRINT' (BASIC 4)

```
.:   027A 78 A9 02 85 91 A9 85 85        ;Points to $0285 here
.:   0282 90 58 60 A5 97 C9 2F DO;  Charactr (eg. backslash).
.:   028A 03 20 91 02 4C 55 E4 A9        ;Points to $0291 & IRQ
.:   0292 80 85 20 A9 00 85 1F A9
.:   029A 04 85 B0 85 D4 20 D5 F0
.:   02A2 20 48 F1 A9 19 85 22 A9
.:   02AA 0D 85 21 20 D2 FF A0 11
.:   02B2 AE 4C E8 E0 0C D0 02 A9
.:   02BA 91 20 D2 FF A0 00 B1 1F
.:   02C2 29 7F AA B1 1F 45 21 10
.:   02CA 0B B1 1F 85 21 29 80 49
.:   02D2 92 20 D2 FF 8A C9 20 B0
.:   02DA 04 09 40 D0 0E C9 40 90
.:   02E2 0A C9 60 B0 04 09 80 D0
.:   02EA 02 49 C0 20 D2 FF C8 C0
.:   02F2 28 90 CB A5 1F 69 27 85;  Columns (#28 or #50).
.:   02FA 1F 90 02 E6 20 C6 22 D0
.:   0302 A6 A9 0D 20 D2 FF 4C CC
.:   030A FF
```

Notes:

[1] Has no test for 'stop'; if inadvertently started, switch off the printer.

[2] Designed for upper case with graphics mode.

[3] The marked byte controls the width of the output: change this to #$50 (80 in decimal) with 8032 machines, else you'll get 50 lines of 40 characters.

[4] The other marked byte controls the character which starts the print at any time during program running. The backslash character is #$2F in location $97 with 8032 machines, but #$45 in others, because of differences in the keyboard organisation.

[5] Calling $027A (by SYS 634) redirects the IRQ vector so every sixtieth of a second backslash is tested for, and, if found, the program stops while the entire screen is dumped. Obviously, if the IRQ vector is reset, by you or by tape activity, or if the interrupt is off, 'keyprint' won't work.

[6] This is positioned in cassette buffer #1, for compatibility with disks (which in BASIC 4 use parts of buffer #2) and external cassettes (device 2). A few early addresses need changing to relocate the routine.

9.5 Animation

Before examining programming methods, let's briefly look at some of the stock-in-trade methods of animators.*

Broadly speaking, the object is to get an impression across at least expense. In practice this means using as much repetition as possible, as few elaborate drawings, and as few frames-per-second as looks reasonable. Significant features are enlarged, e.g. head, nose, eyes; less inportant features are suppressed. The overall figure must have its configuration of (say) arms, legs and body correct with respect to the centre of gravity, if motion is to be suggested. Too many 'in-betweens' should be avoided: a stylised face may have only a profile or full-face, perhaps a 3/4 face. Symbols will obviously be needed: 'Microchess' successfully used very stylised chess pieces in its PET version. 'Space invaders' has led to the acceptance of things like busloads of people and orange lawnmowers whizzing around screens. An article by an advertiser, D Ross ('Creative Computing',Jan.'81) lists rules of thumb for eyecatching animations,

---

*Computer graphics can produce very impressive (and expensive) effects; some simulators for air and sea pilots have real-time displays in colour of considerable realism. Some west-coast US universities, and the New York Tech, are renowned for their work in this field. See (e.g) 'Principles of Interactive Computer Graphics' (Newman & Sproull, McGraw-Hill), which is however heavily mathematical. Work of this type, which may involve data transfer rates of many megabytes per second, is outside the capacity of present microcomputers.

largely based on the idea that any motion attracts attention. ("'If it moves, salute it' is a biological imperative not confined to the quarter-deck"- G Spencer Brown).
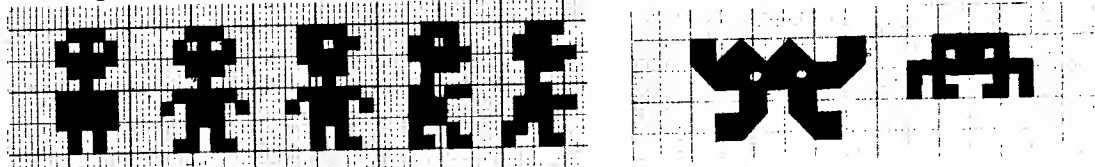
Paraphrasing this and other articles gives 'rules' something to this effect:

    i. Always have something on the screen (i.e. don't just clear it).

    ii. Always have movement of some kind, blinking or flashing text, etc.

    iii. Vary the speed (but keep it at at least medium-pace) and vary styles of lettering - large, small, overwriting, rolling right-to-left, 3-D - and their relations to objects. Words may appear on a board, within a speech balloon, from an 'alphabet soup', or shot from a gun.

    iv. Even apparently dull things may be animatable; Ross quotes a 'frypan with a flickering flame' and 'bread popping from a toaster'. The BASIC loader of Supermon puts numbers in the top left of the screen, so you have something to watch.

    <u>Animation by replacement of the screen</u>. An obvious method to achieve motion is to rewrite the screen at intervals of a fraction of a second. With a 32-K 40-column machine, a maximum of about 30 different screens can be held in RAM simultaneously; this is more than enough to provide good animations of such things as engines. The worst part of such a program is the effort of 'drawing' and storing the individual screens. Once they have been stored in RAM - it's sensible to practise storing them to disk <u>before</u> all the work of entering them - they can be displayed by a program like the following machine-code. I've assumed that the screens are arranged in 1-K sets starting just below screen RAM; adjustments for non-32K and non-40 column machines aren't too difficult. The start of each screen therefore is a block of RAM starting at a page. For convenience I've arbitrarily numbered the screens in descending order, starting 1. The object is to access any screen easily; let's use the idea of a switch, so that a POKE into some key location with 3, say, instantly displays screen 3. Then it's easy to control the animation in either machine-code or BASIC; for example BASIC needs only something like: 10 FOR J=1 TO 10: POKE 634,J: NEXT: GOTO 10 to show 10 screens in sequence.

| RAM→ | ... | $7000 | $7400 | $7800 | $7C00 | $8000-$83FF |
| | ... | #4 | #3 | #2 | #1 | SCREEN DISPLAY |

This machine-code (I've omitted the initialisation routine to direct the IRQ vector to $027B) fits the bill. It stores the addresses 'from' and 'to' in the random-number area in the zero-page.

```
027A 00          BRK            ;Holds key byte 0,1,2,...
027B AD 7A 02    LDA $027A      ;Load key byte
027E F0 27       BEQ $02A7      ;Exit if it's zero
0280 0A          ASL
0281 0A          ASL            ;Key byte now 4,8,12,...
0282 49 FF       EOR #$FF       ;Flip bits...
0284 38          SEC            ;+1 for 2's complement
0285 69 80       ADC #$80       ;Gives #80 minus 4*screen#
0287 85 89       STA $89        ;High byte of start address
0289 A9 00       LDA #$00
028B A8          TAY            ;Initialise offset Y
028C 85 88       STA $88
028E 85 8A       STA $8A        ;Set low bytes to zero
0290 8D 7A 02    STA $027A      ;And reset key byte off.
0293 A9 80       LDA #$80
0295 85 8B       STA $8B        ;Now (88)=7C00 etc; (8A)=8000
0297 A2 04       LDX #$04       ;Counter for 4 pages
0299 B1 88       LDA ($88),Y
029B 91 8A       STA ($8A),Y    ;Transfer loop for 1 page
029D 88          DEY
029E D0 F9       BNE $0299
02A0 E6 89       INC $89
02A2 E6 8B       INC $8B
02A4 CA          DEX
02A5 D0 F2       BNE $0299
02A7 4C 2E E6    JMP $E62E      ;Continue interrupt. [E62E BASIC 2]
```

```
027A 00 AD 7A 02 F0 27 0A 0A
0282 49 FF 38 69 80 85 89 A9
028A 00 A8 85 88 85 8A 8D 7A
0292 02 A9 80 85 8B A2 04 B1
029A 88 91 8A 88 D0 F9 E6 89
02A2 E6 8B CA D0 F2 4C 2E E6 ─────→ BASIC 4 : 55 E4.
```

Test the routine by displaying the registers from monitor (type .R) and changing IRQ to 027B. Enter .G 0004, which substitues the new IRQ for the old. If 027A did not hold zero, the screen will immediately fill with some lower part of RAM.

    The appearance of this type of animation can be improved with a form of 'in-betweening'. A screen is not simply moved bodily in one movement. Instead, each screen replaces its predecessor in two stages. The first compares the two screens, and puts blanks in all locations which are not identical. Then it shifts the new screen. The effect is to simulate movement more accurately, by keeping the fixed parts of the image but temporarily deleting the moving parts. Thus part A does not instantly reappear in position B, but only after a very short delay. This is worth trying, although with some images which rely on reversed graphics there may be too much flickering.

Table of screen memory locations

Screen line	Start of line			
number	40 columns		80 columns	
0	$8000	32768	$8000	32768
1	$8028	32808	$8050	32848
2	$8050	32848	$80A0	32928
3	$8078	32888	$80F0	33008
4	$80A0	32928	$8140	33088
5	$80C8	32968	$8190	33168
6	$80F0	33008	$81E0	33248
7	$8118	33048	$8230	33328
8	$8140	33088	$8280	33408
9	$8168	33128	$82D0	33488
10	$8190	33168	$8320	33568
11	$81B8	33208	$8370	33648
12	$81E0	33248	$83C0	33728
13	$8208	33288	$8410	33808
14	$8230	33328	$8460	33888
15	$8258	33368	$84B0	33968
16	$8280	33408	$8500	34048
17	$82A8	33448	$8550	34128
18	$82D0	33488	$85A0	34208
19	$82F8	33528	$85F0	34288
20	$8320	33568	$8640	34368
21	$8348	33608	$8690	34448
22	$8370	33648	$86E0	34528
23	$83C0	33688	$8730	34608
24	$83E8	33728	$8780	34688

9.6 Pen Plotters. Plotters are not common peripherals; they are used for computer-aided design, and are not often found in the micro world. The best plotters are large pieces of equipment, either 'flat-bed' or 'rotary'; the latter use wide rolls of paper. Benson is one manufacturer of this type of equipment; Calcomp is another. Smaller scale plotters are available from, for example, Hewlett-Packard and Houston Instru-

ment make smaller models, desk-top size. The principle of these machines is to attach a pen to a carrier which is movable in perpendicular directions under program control. Typically, two stepper-motors drive the carrier. An 'unintelligent' plotter can move its pen only in steps; 'intelligent' plotters can 'home' and store coordinate positions. They may have other features, such as a set of alphanumeric character plotting routines in ROM. The precision of the motors controls, to some extent, the maximum plotting-speed obtainable. And the step-size controls the fineness of the resulting drawings, which, because of the stepwise nature of the plotting process, inevitably have a slightly serrated appearance. Typically, several step-size and step-rate combinations can be selected. For example, a small Houston 'HiPlot' can plot either 240 steps per second at .01 inch step-size, or 480 steps per second at .005 inch step-size. Thus the fastest rate of drawing is about 2½ inches with this model. This parameter is rather important; a complicated drawing can take a lot of time. Moreover, the baud rate is also important. Suppose a CBM has an interface set at 600 baud; this means 600 *bits* (not bytes) per second. If the programming system is such that one byte generates one movement of a motor, then a maximum of only about* 75 bytes can be sent per second; this is fine for (say) a daisywheel printer, but confines a plotter to perhaps an inch per second at most.

To illustrate the programming methods used with plotters, I'll take the 'Hiplot' as an example. This machine is controlled by only ten commands; 8 of these are directions as shown in the diagram, and the remaining two commands move the pen down to the paper and lift it from the paper. All other positioning, for example of the pen before plotting starts, is done manually. Each direction on the diagram is labelled with the character which, when sent from the computer, causes one step to be plotted in that direction. Obviously, each motor can step in the positive direction, or in the negative direction, or not at all. Thus there are 3*3 = 9 combinations. There is no particular command for no-movement-at-all.
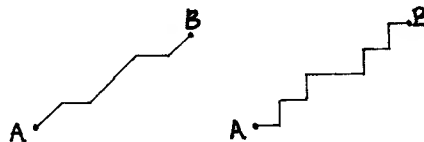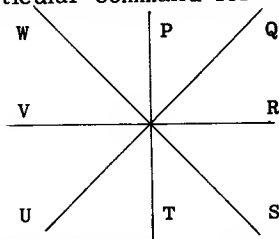
The 45° lines are of course generated by simultaneously activating both motors. This is useful, because some of the jaggedness of lines can be taken out. When plotting a straight line, for example, the appearance can be improved by building it from 45° lines with either horizontal or vertical lines, rather than drawing it only with lines parallel to the x- and y- axes. The diagrams show the difference. The program on the next page draws the best straight line between two points in this way. Note that it uses the notation NW\$, E\$, SE\$, and so on as a rather obvious mnemonic. These strings have to be initialised elsewhere in the program, by

     1000 n$="P":e$="R":s$="T":w$="V":ne$="Q":se$="S":sw$="U":nw$="W":u$="Y":d$="Z"

Lines 107 and 109 test the gradient of the line which is to be drawn; those with gradient < 1 are drawn by program lines 110 − 195, and steeper steeper straight lines are dealt with by the part of the program starting at 300. Note that steep lines require N\$ or S\$, while gentle lines use W\$ or E\$. Logical file #4 is assumed to be open to the plotter.

Circles can be difficult to program. The standard algorithm, which uses the minimum of trigonometrical calculation, is:

```
500 REM Q=DEGREES SUBTENDED BY EACH STRAIGHT-LINE SEGMENT. EG Q=10 PLOTS A
    36-SIDED FIGURE
510 G=R: H=0: REM R=RADIUS. G AND H ARE INTERMEDIATE VALUES
520 N=360/Q : REM N=NUMBER OF SIDES=NUMBER OF REPETITIONS OF LOOP
530 F=COS(Q*[PI]/180): I=SIN(Q*[PI]/180):REM TRIG PARAMETERS
540 FOR J = 0 TO N
550 C=G*F-H*I: A=G*I+H*F :REM THESE ARE THE X- AND Y-COORDINATES OF THE NEXT PT.
560 REM DRAW THE STRAIGHT-LINE SEGMENT TO THE POINT X=C,Y=A
570 G=C :H=A
580 NEXT J
```

*In practice, 1 byte may be transmitted with 10 bits (say); hence the vagueness.

```
99 REM  **************************************************************
100 REM ** SUBROUTINE TO PLOT LINE BETWEEN POINTS, GIVEN X AND Y DISTANCES
101 REM **************************************************************
105 XP=0: YP=0
106 IF XD=0 THEN M=1E9: GOTO 300
107 M=ABS(YD/XD)
109 IF M>1 THEN 300
110 IF XD>0 THEN X$=E$: Y$=NE$: IF YD<0 THEN Y$=SE$
120 IF XD<0 THEN X$=W$: Y$=NW$: IF YD<0 THEN Y$=SW$
130 XD=ABS(XD): YD=ABS(YD)
160 FOR S=1 TO XD: PRINT£4,X$
170 IF M*S>YP THEN PRINT£4,Y$: YP=YP+1: S=S+1: IF S<XD GOTO 170
180 NEXT
190 FOR J=0 TO 1E5: IF YP-1<YD THEN PRINT£4,Y$: YP=YP+1: NEXT
195 RETURN
300 IF YD>0 THEN Y$=N$: X$=NE$: IF XD<0 THEN X$=NW$
302 IF YD=<0 THEN Y$=S$: X$=SE$: IF XD<0 THEN X$=SW$
304 XD=ABS(XD): YD=ABS(YD)
305 FOR S=1 TO YD: PRINT£4,Y$
310 IF S>M*XP THEN PRINT£4,X$: XP=XP+1: S=S+1: IF S<YD GOTO 310
320 NEXT
330 FOR J = 0 TO 1E5: IF XP-1<XD THEN PRINT£4,X$: XP=XP+1: NEXT
340 RETURN
```

Pattern plotting Mathematical curves and drawings, either as single long lines (e.g. Lissajou figures - see 'SET') or as repeated plots (straight line segments mimicing string and nails/ repetitive drawings in which parameters are slightly varied/ etc.) have been fairly popular. They may also be useful in mathematical education. Curves with plottable formulas include these following examples. These equations are all parametric, so pairs of values are generated, and can be plotted immediately. The scale of course has to be adjusted so the drawing is aptly sized.

Trisectrix: x=cosa + cos2a   Cycloid: x=a + sina   Cardioid: x=2cosa + cos2a
        y=sina + sin2a   y=1 - ncosa   y=2sina + sin2a

Folium: $x=t/(1+t^3)$ Epicycloid: x=mbcosa - bcosma   Strophoid: $x=(t^2-1)/(t^2+1)$
       $y=t^2/(1+t^3)$ y=mbsina - bsinma $y=t(t^2-1)/(t^2+1)$

There are of course innumerable equations of functions of y in terms of x, which are instructive to plot; for example:

Catenary: y=c cosh(x/c)   Normal: $y=e^{-(x-a)^2}$ Damped sine: $y=e^{-kx}$(Asinwx + Bcoswx)

The specimen below, and that at the foot of the page before last, were constructed by a different principle; four separate types of 'tile', used as building-blocks and drawn next to each other in random sequences chosen by the computer, make up
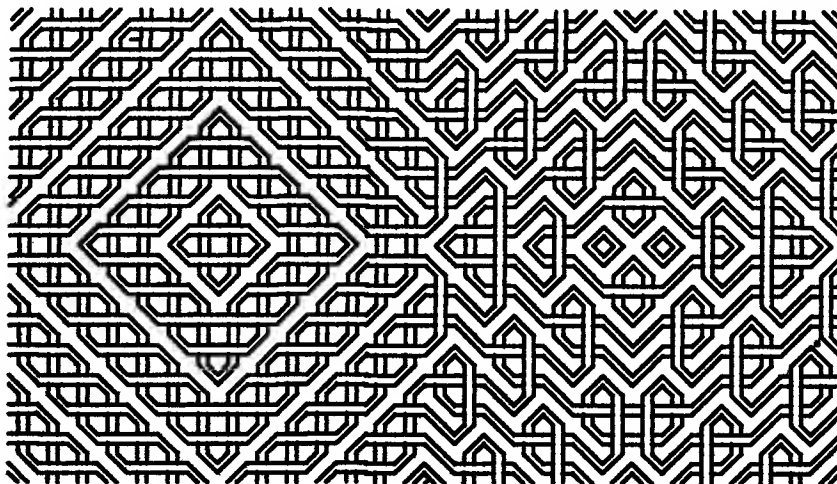
the entire drawing. This example plots 'tiles' of these designs; when drawn as neighbours, their lines must always link, forming an Islamic-style abstract pattern.
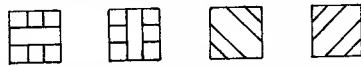
The program is too long for reproduction here; it has four subroutines, one for each 'tile', and a routine to convert the random sequence of five or so 'tiles' into a symmetrical array. The actual loop which performs the plot is this:

```
3500 REM ****  NOW PLOT SYMMETRICAL PATTERN USING CA%(,) ARRAY DATA  ***
3510 V = 1 :REM 'LOOP'
3520 FOR H = 1 TO 2*HRIZ
3530 ON CA%(H,V) GOSUB 100,200,300,400
3540 NEXT H
3550 V=V+1: IF V>2*VERT THEN END
3560 PRINT£4,PU$: FOR J=1 TO 3*S: PRINT£4,SW$: NEXT
3570 FOR J=1 TO 3*S*(2*HRIZ-1): PRINT£4,W$: NEXT
3580 GOTO 3520
```

Lines 3560-3570 move the pen back from the end of one line of tiles to the start of the next.

Three dimensional drawings. Perspective drawings are possible, but the programs are mathematically difficult. Each corner has to be entered as three co-ordinates and their positions on plotting calculated so that the geometrical shape is projected on a plane. The picture of a CBM below was plotted like this; in fact it is a stereo plot, the original being a stereoscopic pair in red and blue. When viewed through two filters, red for one eye and blue the other, this creates a stereo image. Several black and white films have been made using this colour separation technique. Again, the program is too long for reproduction. Readers interested in this will need to consult textbooks on the theory of perspective projections; matrix arithmetic is usually the preferred way to store data and process it. There are some snags: one is the hidden-line removal problem, which tries to deal with lines which would be blocked out by an opaque object, but which the computer may not recognise as unwanted. (The CBM picture had no 'corners' input of its far side, which evaded the difficulty). Another snag may involve the conceptual framework of the projection: the arithmetic may
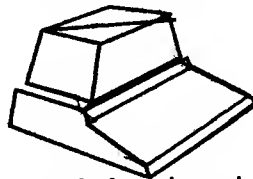
assume that the object is fixed, and the viewpoint moves around it; or the object itself may be rotated. If the method isn't suited to your needs, you may find it necessary to calculate angles and distances in order to make the image the correct size or make a series of images bear the correct relationships to each other.

Pseudo-perspective drawings - axonometric or isometric - are easier because there is much less calculation involved. As an example, consider what is required to

Axonometric

Isometric
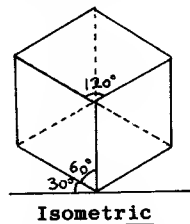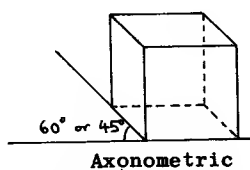
plot a mathematical function in 3-dimensions. The aim is to plot a function to produce an image of the sort indicated in the sketch. How can this be done? The simplest method is to ignore the hidden-line problem, simply plotting cross-sections of the curve from left to right for a range of values. Taking account of hidden lines makes the programming, and the running time, longer. The easy programming
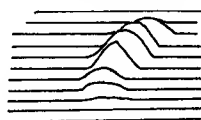
solution is to adopt a modified axonometric
projection, exactly like this diagram *except*
that the horizontal y-axis is drawn to coin-
cide with the z-axis. Now, for each point
x_n, a set of values $z(x,y)$ are found, and
plotted as dots provided that each value
exceeds the previous value. In this way a
column of dots builds up, scanning from
left to right, which takes the form of the
type of drawing we have in mind. This method is fine with a VDU, but not very good
with a plotter, because more time is spent moving the pen than actually plotting. So
a more elaborate method must be used, which stores the maximum value plotted so far
at every x_n, and plots a continuous line from left to right, except where it is inter-
rupted by some larger value.
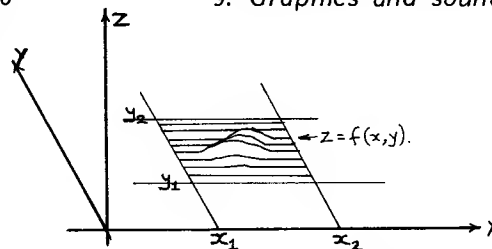
```
      FOR Y = Y1 TO Y2 STEP YS
      FOR X = X1 TO X2 STEP XS   :REM XS,YS MEANS X-STEP SIZE, Y-STEP SIZE
      Z = FN(X,Y) + Y            :REM EXTRA Y GIVES 'PERSPECTIVE', INCREASING THE
                                  'HEIGHT' OF FAR POINTS
      N=(X-X1)/XS                :REM NTH COL. OF PLOT; N=0,1,2,...
      IF Z<MAX(N) GOTO L1        :REM DON'T PLOT SEGMENT IF VALUE < MAXIMUM
      MAX(N)=Z: PLOT SEGMENT     :REM SAVE NEW MAXIMUM AND PLOT
L1    NEXT X: NEXT Y
```

This schematic BASIC program shows the method.

9.7   Sounds and the PET/CBM

Introduction   Computer synthesized music has achieved considerable success, notably
with organ-like sounds and special effects; ordinary orchestral instruments remain
resistant to synthesis, because of the extremely complicated waveforms which they
generate. Before considering microcomputer music, let's look at a typical note-synthes-
ising method. We may distinguish three stages in the life of a note: attack, sustain,
and decay. The attack - the period in which the note is becoming established - has a
spiky and irregular waveform because of the note's instability. This part is difficult
to synthesize. The sustained part of the note has a steady waveform, including
harmonics characteristic of the instrument. Finally, the decay also has the same wave-
form, but harmonics tend to disappear as the note attenuates.
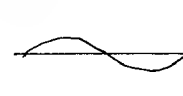
        Attack   **Sustain**   **Decay**

To simulate these stages, we can hold a table of, say, 256 bytes per note. Each byte
may have a value of 0-255; and all the bytes together provide samples of the wave-
form from the start of a wave to its end (i.e. 1 wavelength). If we cycle through the
'attack' table a number of times, then the 'sustain' table, and finally the 'decay', the
entire note is simulated in this manner:
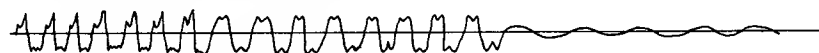
    Start ←—Attack —→ ←— Sustain ————————→ ←——— Decay ——→

If the digital values generated are converted to analogue signals by a digital-to-anal-
ogue converter ('DAC'), an approximation to the original sound will result. Speech
is synthesized from a vocal cord analogue (buzzing sound) with perhaps 3 bandpass
filters for each of the major formants of speech sounds. Chips are available, and are
used commercially in some products, to produce sounds of specified waveform, frequen-
cy, and envelope shape, to synthesize speech, and so on.

     New, 12 inch screen CBMs are all equipped with an internal speaker (of very
low volume). This provides, as we'll see, an easy way to generate tones, clicks,
squeaks, and so on; it relies on a square wave, produced, in the CBM's case, by the
shift register, which periodically shifts a bit to be amplified by the speaker. This has
only two positions, out and in, so the sound is cruder than that produced by digital
to analogue conversion.

The frequency produced by these methods can be calculated easily enough for any particular case, but what is the corresponding note? Several musical scales exist, including the scientific ('Just') scale with middle C of 256 Hz, and two equal-tempered chromatic scales, of which the American standard has middle C of 261.63, and the International standard a middle C of 258.65. Chromatic scales have 12 notes before the octave repeats, and their frequencies have a constant ratio of $2^{1/12}$ (1.05946...). An abridged summary of the full range of notes for each scale follows:-

NOTE	EQUAL TEMPERED CHROMATIC SCALE		SCIENTIFIC SCALE
	FREQUENCY: ASA	INTERNATIONAL PITCH	
C_4	261.63	258.65	256
$C\#_4$	277.18	274.03	
D_4	293.66	290.33	288
$D\#_4$	311.13	307.59	
E_4	329.63	325.88	320
F_4	349.23	345.26	341.33
$F\#_4$	369.99	365.79	
G_4	392.00	387.54	384
$G\#_4$	415.30	410.59	
A_4	440	435	426.67
$A\#_4$	466.16	460.87	
B_4	493.88	488.27	480

Sounds with microcomputers   Hal Chamberlin is one of the most well-known authorities on music generation by computer; see Byte, Sept.'77, or his more recent book, 'Musical Applications of Microprocessors' (Haydon/ Wiley). Both user-port techniques for the PET appear to have originated from him. (A commercial product for 4-voice sound synthesis, by MTU, is his. Another well-known product is the 'Visible music monitor', or VMM, by AB Computers. Each has software with a digital-to-analogue converter). VlC has a 4-voice synthesizer built in. Other machines, e.g. Apple, have speakers built in as standard. As we'll see, the usual PET system uses a different system of operation which is probably easier to work. Let's first look at square-wave generation on the PET, using extra hardware. The programming is identical to that for the wide-screen models, which have the same circuitry ready supplied.

Square-wave generation with the VIA's shift register   Before the specimen programs, here are three alternative methods to make the VlA's square waves audible. They are arranged in order of ease of implementation. I don't recommend users without hardware experience to try this, and can't accept responsibility for disasters which may result. (Not that anything untoward is likely to happen, in fact).

(i) It's not widely known that the PET's sound can be amplified without any connections at all, except one to the M pin, corresponding to CB2, which is the right-most-but-one pin on the underside of the user port. (This port is next to the lEEE port - check in the manual). If a single connection to this pin is taken as a wire near a radio, its radio frequency signals will be picked up, and the sound broadcast. The result is rather noisy, in the technical sense, but is better than nothing.

(ii) The same single wire can be attached to the radio's volume control. Assuming this is a thumbwheel, connect the output to A or B, enabling the volume control to operate. (Not C, which doesn't).
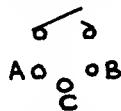
(iii) A simple amplifier circuit, of three components plus a small speaker, gives very adequate sound. The circuit diagram shows how the components are arranged; a sketch on the following page shows how they appear, in unmounted form, with the computer. lt is of course possible to buy amplifiers as chips, which makes very neat and compact units available.
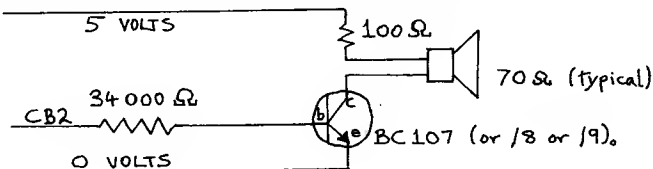
The signal may be reduced by increasing the smaller resistance from 100 ohms. The larger resistor's value is set at 34000 ohms since the gain is of the order of 200, so 200 x (100 + 70) = 34000.
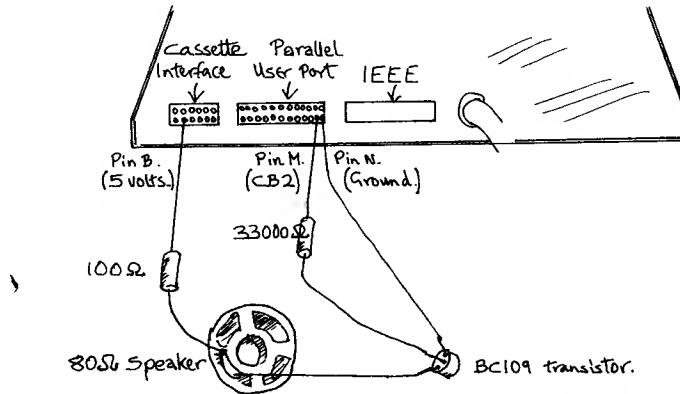
Programming the speaker is a fairly easy matter. The usual method is to use the VIA's Shift Register, at $E84A. A shift register moves one byte to the right at regular time intervals determined by a clock. In our case, the location $E848, which is the low byte of timer 2, controls the rate of shifting. Also, the Auxiliary Control Register, which has 8 alternative shift-register settings, is set for 'Free Running Output Mode'. Let's first look at a machine-code example; in fact the routine which BASIC 4 uses to tinkle its internal bell.

The jump-table address (E02A in the 8032) causes the chimes to ring once; the jump is to E6A7. To ring the bell twice, call E6A4, which has the command JSR E6A7, and consequently tinkles the bell, then drops through to tinkle it again. The bell-ringing routine is like this:

```
      LDA #$10
      STA $E84B ;AUX.CTRL.REG. ('ACR') INTO FREE RUNNING OUTPUT MODE, T2 CONTROLLED
      LDA #$0F
      STA $E84A ;SHIFT REGISTER HOLDS 0000 1111
LOOP  LDA xxxx  ;LOAD A WITH SOME TABLED VALUE
      STA $E848 ;PUT A INTO TIMER 2 (LOW)
      . . .
EXIT  LDA #$00
      STA $E84A ;SHIFT REGISTER HOLDS 0000 0000
      STA $E84B ;ACR HOLDS 0000 0000
```

The values the CBM loads are 0E/ 1E/ 3E/ 7E/ 3E/ 1E/ 0E in turn. The delays before shifting are in proportion to 1:2:4:8:4:2:1 (i.e. spanning 3 octaves, because the frequencies are halved by the delay's doubling). The rationale is this:

(i) Bits 2,3, and 4 between them control the shift register; if for example they hold 000, no shifting occurs; 101 causes just 8 bits, the present contents of the shift rgister, to be shifted under control of T2; and – our sound generator – 100 causes the shift register to shift in 'free run' output under the control of T2. This means that T2, which is decremented at every clock cycle, causes 1 bit of the shift register to be output when ever T2 becomes zero. T2 is automatically reloaded in this mode; so is the shift register. ACR is E84B (59467).

(ii) T2 has a high byte and a low byte. As we shall see, only the low byte is usually used for tone generation. Its location is E848 (=59464).

(iii) The shift register itself is E84A (=59466). Its pattern of bits determines the frequency of the note and its timbre.

To see how these three VIA registers cause a square wave to be generated, consider these illustrations of the shift register at regular intervals as T2 times out. A bit is set high or low (1 or 0) on CB2 at each of these intervals, and the resulting wave train appears as shown; it has only two amplitudes, and is a square wave (with disturbances depending on mechanical and electrical error).

```
Start:  10110011
T2 out: 01011001  CB2=1
T2 out: 00101100  CB2=1
T2 out: 00010110  CB2=0
T2 out: 00001011  CB2=0
T2 out: 00000101  CB2=1
```

Many of these wave trains corresponding to SR's bit patterns sound identical to waves originating from other bit patterns which at first sight appear different. For example,

01 (0000 0001) and $80 (128 decimal). 1000 0000 sound identical, because each produces a single pulse. More subtly, $0F = %0000 1111, changes the note's apparent frequency because the ear picks up the period of the wave as half that of, say, 1110 1100. This applies also to 0011 0011 and 0101 0101 and their variants, in which the frequency shifts (of an octave each) are caused by the repetitive bit pattern. Note also that patterns which we might call 'inversions' - e.g. 1111 1000 and 0000 0111 - sound alike to the ear; this is a fact about the psychology of perception, rather than physics. We can list all the fundamentally different bit patterns; they are

RESULTS OF M ZIMMERMANN'S 'MUSIC GENERATOR' PROGRAM (FEB 81 BYTE)

```
 1  0 0 0 0 0 0 0 1
 3  0 0 0 0 0 0 1 1
 5  0 0 0 0 0 1 0 1
 7  0 0 0 0 0 1 1 1
 9  0 0 0 0 1 0 0 1
11  0 0 0 0 1 0 1 1
15  0 0 0 0 1 1 1 1
17  0 0 0 1 0 0 0 1
19  0 0 0 1 0 0 1 1
21  0 0 0 1 0 1 0 1
23  0 0 0 1 0 1 1 1
27  0 0 0 1 1 0 1 1
37  0 0 1 0 0 1 0 1
43  0 0 1 0 1 0 1 1
45  0 0 1 0 1 1 0 1
51  0 0 1 1 0 0 1 1
85  0 1 0 1 0 1 0 1
```

Zimmerman's 'Byte' article also includes Fourier Analysis methods, routines being given for the BASIC 1 PET.*

The great advantage of this method of tone generation is that, once started, it continues. Apple's system requires the machine to continually tweak the speaker, so no other processing can be carried on. To show how this can be used, the following short machine-code program uses the interrupt to help play a tune. (It is written for BASIC 2; other BASICs need their correct IRQ in place of E62E). The overhead in terms of time is tiny; the space overhead depends on the table of notes. This example 'plays' the zero-page, producing 1 note with each interrupt; you should therefore be able to hear repetition of the 'tune' each 4 seconds or so. Load the routine, and point the IRQ to $0300 (e.g. SYS 4/ M 0090 00 03 -- same --). The routine is now active.

Program Demonstrating use of the Interrupt to Play a Tune while BASIC runs.

NO.	DEC.	HEX DUMP	DISASSEMBLY		
1	768	E6 00	$0300	INC $00	; LOCATION $00 IS A COUNTER; EACH TIME IT IS
2	770	F0 03	$0302	BEQ $0307	; INCREMENTED TO #00, THE NOTE IS CHANGED.
3	772	4C 2E E6	$0304	JMP $E62E	
4	775	A9 FF	$0307	LDA #$FF	; RELOAD THE COUNTER (SMALL VALUE = SLOWER)
5	777	85 00	$0309	STA $00	; OUR DEMONSTRATION PICKS ITS NOTES FROM THE
6	779	A5 70	$030B	LDA $78	; ZERO PAGE, CYCLING THROUGH 256 VALUES.
7	781	EE 0C 03	$030D	INC $030C	; STORE ACCUMULATOR IN THE TIMER.
8	784	8D 48 E8	$0310	STA $E848	; AND CONTINUE NORMAL INTERRUPT.
9	787	4C 2E E6	$0313	JMP $E62E	

POKE 59467,16 starts SR shifting in free-running mode. This starts the 'tune'. POKE 59466,X enters a byte (and may be needed to make the tune audible!). It also controls the timbre of the notes produced, within the restrictions imposed by the square wave. A BASIC program can be run while the noise continues. (POKE 59467,0 to turn it off).

---

*These are fairly easy to convert to other ROM sets; see Chapter 15. Fourier was a French mathematician who proved that any periodic waveform could be generated by adding sine curves together (sometimes - as with square waves - infinitely many in number). 'Fundamentals' and 'overtones' are an aspect of this. The process is similar to that of Ptolemy, who in effect synthesised ellipses from many circular motions.

POKE 776,X varies the rate at which notes are changed; this location is used to count interrupts; the smaller the value, the longer is each note played, up to a maximum of 4 or 5 seconds. Obviously, a table of bytes can be played, so a recognisable tune will emerge, while BASIC runs. A table of 256 bytes (say) played at the rate of 2 notes per second (the equivalent of POKE 776,25 or so in our example) will run for 2 minutes before repeating. We can use our technique of 'switching', i.e. using a special POKE location to cause the interrupt to choose different tunes, from tables of perhaps 4 or 5. In fact, these could be chosen by the program; thus, a game program might be accompanied by music of the appropriate mood, like a piano accompaniment to a silent film.

This demonstration BASIC program plays 2 octaves:

```
92 REM ################################################################
93 REM #    DEMONSTRATION OF CHROMATIC SCALE OF 2+ OCTAVE RANGE        #
94 REM # EACH NOTE OBTAINABLE BY A SINGLE 'POKE' FROM BASIC OR MACHINE-CODE #
95 REM ################################################################
96 REM
97 REM
100 POKE 59467,16:      REM FREE RUN MODE
110 POKE 59464,0 :      REM SWITCH OFF IF CONTENTS EXIST
120 POKE 59466,22:      REM OTHER VALUES WILL PRODUCE THEIR OWN TIMBRES/ TONES
130 DIM N(30)    :      REM ARRAY WHICH WILL HOLD THE CONTENTS OF DATA
140 FOR X = 1 TO 1000: READ N(X): IF N(X) <> 999 THEN NEXT:    REM READ DATA IN.
150 N = X - 1:          REM N IS THE NUMBER OF ITEMS IN THE TABLE OF NOTES
191 REM
192 REM
193 REM ################################################################
194 REM #      PLAY ALL THE NOTES FROM THE DATA TABLE IN ASCENDING SEQUENCE    #
195 REM ################################################################
196 REM
200 FOR X = 1 TO N:              REM PLAY ALL NOTES
210 POKE 59464, N(X):            REM POKE TIMER WITH DATA
220 FOR J = 0 TO 200: NEXT J:    REM DELAY LOOP BETWEEN NOTES
230 NEXT X
500 POKE 59467,0: END:           REM SWITCH OFF
991 REM
992 REM
993 REM ################################################################
994 REM # STARTING VALUE OF 252.1 FOUND BY TRIAL TO GIVE CLOSE APPROXIMATIONS #
995 REM # TO TRUE CORRECT CHROMATIC RATIOS; BUT DOES NOT HAVE PERFECT PITCH. #
996 REM # START VALUE MULTIPLIED BY 2 ^ ONE-TWELFTH; THEN 2 CYCLES SUBTRACTED.#
997 REM ################################################################
998 REM
1000 DATA 250,236,223,210,198,187,176,166,157,148,139,132 :REM FIRST SET OF 12
1010 DATA 124,117,110,104,98,92,87,82,77,73,69,65          :REM SECOND SET OF 12
1020 DATA 61,999                                          :REM 999 SIGNALS END
```

How can we calculate the absolute frequency of a note? Let's say that timer 2 contains the value T (<256). The timer decrements once every microsecond, so one bit will shift every T microseconds.* So 8 bits are sent in 8T microseconds, and the full period of a wave is twice this, assuming a pattern like 0000 0001, not an internally repeating one such as 0101 0101; so the frequency is 1000000/16T cycles per sec. This is the same as 62500/T. So a frequency of 256 is obtained by poking 244 into T2. When generating square waves, the greatest precision can be got by using the longest possible value of T2; this means a correspondingly fast square-wave, 0101 0101. This has 4 times the frequency of 0000 1111. Timer 2's high byte (in E849 = 59465) may need to be used as well as its low byte.

Several square waves can be generated simultaneously, although the resolution is inevitably poor: this requires the reloading of the shift register after each timer countdown, with the next in the sequence of combined waveforms. The timing could be carried out by enabling T2's interrupt, but the technique is tricky.

---

*This seems a reasonable assumption. R Zaks ('6502 Applications Book') implies that the half-period is N+1.75, made up of an average from N+2 (top of pulse), and N+1.5 (bottom of pulse). His method gives slightly different values for the constants; it was used in the BASIC program above - see line 996. At the time of writing I haven't found a definitive answer to this simple question.

These short BASIC routines demonstrate typical easily-achieved sound effects:

(i) <u>Glissando</u>

```
1000 POKE 59466,15: POKE 59467,16: REM REGULAR SQUARE WAVE + SR ON
1010 FOR J = 255 TO 1 STEP -1    : REM LOW TO HIGH PITCH
1020 POKE 59464,J                : REM BRIEFLY PRODUCE TONE
1030 NEXT: POKE 59467,0          : REM SR OFF WHEN LOOP ENDS
```

(ii) <u>Beep</u>

```
1000 POKE 59466,15: POKE 59467,16: REM OR USE DIFFERENT (<>15) TIMBRE...
1010 POKE 59464,140              : REM ... OR TONE
1020 FOR J = 0 TO 20: NEXT       : REM SHORT DELAY LOOP
1030 POKE 59467,0                : REM SWITCH OFF AFTER BEEP
```

(iii) <u>Murmur</u>

```
1000 POKE 59466,45: F=59464
1010 FOR J = 1 TO 200
1020 X=RND(1)*16                 : REM NOTES VARY AROUND THIS
1030 FOR K = 0 TO 8*RND(1)
1040 POKE F,N(X+K)               : REM ASSUMES N() HOLDS TABLE OF NOTES
1050 NEXT K,J: POKE 59467,0
```

SUMMARY OF CB2 SOUND VIA LOCATIONS.

E848	59464	Timer 2 (low byte)	T2L	0=off; otherwise, small means high pitch
E849	59465		T2H	Only useful for slow timing
E84A	59466	Shift Register	SR	Contents determine timbre and octave
E84B	59467	Auxiliary Control	ACR	Bits x<u>xxxxxx</u> control SR (Free run 100)

Tones with 8-bit resolution Output port A of the VIA has bits PA0-PA7 connected to pins C,D,E,F,H,J,K, and L of the user port. (These are on the underside). The diagram shows a digital-to-analogue converter which allocates weights to each bit, so that the most significant bit L has twice the effect of pin K, which in turn has twice the effect of pin J, and so on. The resistor values chosen are 'preferred values'; their values are only approximately in the ratio 2 to 1. Greater precision requires the use of a 'ladder' circuit, corrected for the 1K resistor to ground. The diagram includes only 5 pins, ignoring the least significant 3 bits, whose effect, in such a simple circuit, is small. The resistor-capacitor arrangement provides some smoothing. Pin M is included, so CB2 sound is available as well; pin 6, with a resistor, can be added, so tape loading can be aurally checked. The output should be amplified by (for example) a plug inserted into a portable radio's DIN socket. A simple transistor circuit isn't sufficient.

DEMONSTRATION PROGRAM

```
       LDA #FF  ;CONFIGURE PORT A
       STA E843;   FOR OUTPUT
L1     LDA 2000,X
       STA E84F;SEND OUTPUT
       INX
       BNE L1
       LDA 9B   ;TEST STOP KEY
       CMP #EF  ;[0209 IN BASIC1]
       BNE L1
       RTS      ;RETURN TO BASIC
                [OR BRK IN M/CODE]
```
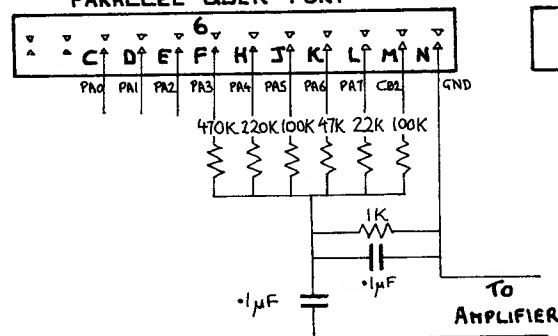

This simple demonstration routine first configures all 8 bits of port A for output, then repeatedly stores 256 bytes (1 page) of bytes into the output port. This repeating pattern constitutes the waveform. A routine to stop the output is provided. In this way 256 separate speaker positions model one wavelength of the sound. For high frequencies this is too great, and a smaller table, perhaps 32 bytes, must be used. Two examples follow; note that the starting value of $2000 (=8192) is arbitrary, and other values, such as the top of RAM, can be used. An oscilloscope will display the waveform.

```
I=0: FOR J = 8192 TO 8192+255: POKE J,I: I=I+1: NEXT: REM GIVES SAWTOOTH
I=0: FOR J = 8192 TO 8447: POKE J,128+125*SIN(I): I=I+2*[PI]/256: NEXT: REM SINE
```

CHAPTER 10: THE TRANSITION TO MACHINE-CODE

10.1 Introduction and some 8-bit concepts.

Machine-code programming can only be learnt by trial and error, by experimenting with sample programs to see what they do, and transferring the results of this learning to one's own programs. This chapter explains the connection between decimal and hexadecimal notation, and the meaning of 'bit', 'byte', and other related words. It also has short examples of machine-code programming; these are continued and expanded in the next chapter. But the bulk of the present chapter is concerned with monitors: not the VDUs, but software enabling the programmer to get to the 6502 chip. The novice in 6502 machine-code will find some of the detail hard to follow: the problem being that machine-code can be understood only with the help of a monitor, but a monitor cannot be understood without knowledge of machine-code. Some of the detail must be skipped on the first reading. Chapter 12 has an alphabetic guide to the 6502, to which reference may be made, but again, because of its comprehensiveness, much will be obscure to the comparative beginner.

The 6502 microprocessor performs all the processing of the PET/CBM. It is supplemented by chips to control the keyboard, screen, and other peripherals, and circuitry to perform such functions as the screen scanning and the control of the power supply. A crystal-controlled clock determines the speed of operation of the 6502, so it is possible to calculate the precise time taken by a program.* Each variety of microprocessor has its own version of 'machine-code' or 'machine-language'. This is a map or dictionary (in effect) giving a one-to-one translation of the contents of locations accessed by the chip to the chip's activity. Each separate machine-code instruction has little effect; only the combined effect of millions of instructions enables a computer to achieve anything. The 6502 is an 8-bit processor. It operates in units of 8 bits. A 'bit' (as many people know) is a 'binary digit'. Conventionally represented as 0 for off and 1 for on, it is the smallest unit of data. Note that a bit isn't *actually* a '0' or a '1'; it is a voltage, interpreted as 'off' in the range zero volts and up, and 'on' in the range five volts down, with the exact range depending on the chip. Most of the 6502's data is stored in RAM or in ROM. If a static charge or voltage spike causes a voltage to drop from (say) 4 volts to 2, the bit will no longer hold its correct value; the data will be 'corrupted'.

A *byte* is a set of 8 bits wired so that they correspond to a single address. 8 pins on the 6502 are used for data transfer, both into and out of the chip. The individual bits are usually represented as bits number 7 to 0 in descending order, with bit 7 the 'high' and bit 0 the 'low' bit. This is consistent with ordinary mathematical notation, using standard base 2 (binary) arithmetic. The value of a byte can be any integer from 0 to 255; there are 256 ($=2^8$) different possibilities. The table below, familiar to everyone exposed to 'modern' mathematics, shows the connection between bits and the overall byte value:

ONE BYTE:

BIT NUMBER:	7	6	5	4	3	2	1	0
POWER OF 2:	7	6	5	4	3	2	1	0
'WEIGHT':	128	64	32	16	8	4	2	1

So, for example, the decimal equivalent of 0000 0000 = 0,
0000 0001 = 1,
0000 0110 = 6,
1111 1110 = 254.

The division of a byte into two halves of 4 bits (known, sometimes, as *'nybbles'*, by a process of paronomasia) is another convention: it is impossible to remember 256 separate numeric symbols for a byte, so *hexadecimal*[2] notation is widely used instead. Each nybble is represented by 0 - 9, A,B,C,D,E, or F. 'A' in hexadecimal ('hex' for short) means 10 in decimal, 'B' means 11, ... ,'F' means 15. This is the representation used by the CBM's built-in monitor. This notation expresses a decimal number of 0 - 255 in two characters at most; and decimal numbers up to 65535 ($=2^{16}$) in four characters at most. The appendix has a complete table of hexadecimal-decimal 8-bit conversions, and

---

*This suggests that programs (calculations for example) might be accelerated by the use of a faster clock.

[2] 'Sedecimal' (all-Latin in origin) is sometimes recommended as a more satisfactory word, naturally without much success.

a conflated table of values multiplied by 256. These may be used to convert 16-bit hexadecimal numbers into decimal and vice-versa. The 6502 is equipped with a 16-bit address bus; pins 9-20 and 22-25 between them carry address data. This design allows $256^2 = 65536$ RAM/ROM addresses to be used directly, without a system of switching. 16-bit, 2-byte hexadecimal numbers are represented by an extension of the notation to four characters:

TWO BYTES (16 BITS):

BIT NUMBER:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
POWER OF 2:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
256* POWER:	7	6	5	4	3	2	1	0								
'WEIGHT':	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

So that, for example, the following conversion relationships hold:

DOUBLE BYTE		HEX	DECIMAL VALUE
0000 0000	0000 0000	$0000	0
0001 0000	0000 0000	$1000	4096 (= 16*256 = 16^3)
0000 0100	0000 0001	$0401	1025 (= 4*256 + 1)
1010 1011	1100 1101	$ABCD	43981 (= ((10*16 + 11)*16 + 12) *16 + 13)
1111 1111	1111 1000	$FFF8	65528 (= 65535 - 7)

I have used the convention of prefixing a hexadecimal number with '$'. This avoids ambiguity in the case of those numbers which happen to include no alphabetic characters. (An alternative convention, unusual with the 6502, is to write 'H' after the number. This is not always satisfactory: 'BEACH' can be a hexadecimal number or an assembler label).

At first, this notation seems odd - it appears strange that $CAFE or $BEEF can represent an ordinary number, and that $20 is 32, and $100 is 256. With practice the interconversion becomes fairly easy, at least with small values, which can be converted mentally - $A2 is clearly 10 sixteens plus 2, i.e. 162; $55 is 5 sixteens plus 5, 88. Chapter 4, section 4.1.1 has one-line BASIC interconversion routines which may be useful. In the absence of a computer or tables, conversions can be carried out with a calculator:

(i) <u>Hexadecimal to decimal.</u> A four-digit hex numeral (say FGHJ) has weights of 16^3, 16^2, 16, and 1 respectively to be multiplied by each respective digit's decimal value. (This is what is meant by 'Base 16'; it is exactly analogous to 10^3, 10^2, 10, and 1 weighting the digits of a decimal number). So the result is $F*16^3 + G*16^2 + H*16 + J$, where F,G,H and J are intended as algebraic representations of any value $0 - 15$. It is often easier to evaluate the result as a continual calculation, multiplying F by 16, adding G, multiplying the result by 16, adding H, multiplying by 16 again, and lastly adding J. In this way, the correct weights are automatically assigned.

(ii) <u>Decimal to hexadecimal.</u> The method is to first divide by $1000, which is 4096; this gives the first, most significant hex digit, of $0 - 15$. Note this digit, then subtract it from the currently-stored decimal value, and multiply by 16. This reveals the second most significant digit. Continue until all four have been found.

The combination of two bytes into an address is an important feature of the 6502 chip, and the formula for a two-byte value, which equals 256*the high byte + the low byte recurs in machine-code, POKEs and PEEKs, and SYS commands. It is perhaps a pity that the 6502 handles double-byte numbers assuming that *the low byte is stored first, followed by the high byte.* In other words, the order is opposite to what you would expect from a normal number. (Some other chips, for example the 6809, have double-byte addressing in 'natural' order). Because of this, pointers used by BASIC BASIC are almost always in this format, which can be used without modification by the chip.

Two other general points about the computer's handling of hexadecimal arithmetic can be made at this point; they are not enormously important, and may be skipped:

(iii) <u>Two's complement arithmetic.</u> This is a convention for the representation of negative numbers, which is implemented on the 6502. In its simplest form, with 8 bits only, bit 7 determines the sign of a number: '0' means positive, '1', bit 7 set, means negative. The rule to change the sign is to *flip the bits and add 1.* Thus, the bit pattern 0101 1001 ($59 = 89 decimal) is made negative by flipping the bits (to 1010 0110) and adding 1, to give 1010 0111. Normally this counts as $A7 = 167 decimal, and this example shows that a number and its two's complement add to 256 or $0100. The point is that addition of positive and negative numbers is consistent with normal use, so that, for instance, $59 and $A7 add to $100, which, ignoring the bit which over-

flows, is zero, which is the result required from the addition of two numbers of opposite sign but equal magnitude. Simply flipping the bits does not provide this, as the numbers add to $FF or 255. Note that the complement of zero ($00 = 0000\ 0000$), i.e. minus zero, does not exist. For experienced programmers, there is an example in section 9.5 which involves subtraction by taking the two's complement. As we shall see, branches use two's complement arithmetic to calculate the address to jump to; only one byte is allowed for this offset, which therefore has a maximum value of $0111\ 1111 = 127$ in the forward direction, and $1000\ 0000 = -128$ in the backward direction.

(iv) The meaning of 'K'. The prefix 'K' or 'Kilo' implies a unit of measurement one thousand times larger than some standard unit. In computer jargon, however, a 'kilobyte' is not a unit of 1000 bytes, but $2^{10} = 1024$ bytes, a figure which derives naturally from the organization of present-day computers, with on-off storage. 1024 bytes is *not* the same thing as $1000 bytes; the hexadecimal interpretation of '1000' is $16^3 = 4096$. Not surprisingly, this can lead to confusion. For example, the RAM needed to store a high-resolution graphics display may occupy (say) $9000 - $AFFF, which is two batches of length $1000. It is easy to think that the display occupies 2K, whereas in fact it uses 8K. This table, of decimal and hex equivalents to integer multiples of 1 K, may be helpful:

TABLE OF KILOBYTE VALUES

			16K	16384	$4000	32K	32768	$8000	48K	49152	$C000
1K	1024	$0400	17K	17408	$4400	33K	33792	$8400	49K	50176	$C400
2K	2048	$0800	18K	18432	$4800	34K	34816	$8800	50K	51200	$C800
3K	3072	$0C00	19K	19456	$4C00	35K	35840	$8C00	51K	52224	$CC00
4K	4096	$1000	20K	20480	$5000	36K	36864	$9000	52K	53248	$D000
5K	5120	$1400	21K	21504	$5400	37K	37888	$9400	53K	54272	$D400
6K	6144	$1800	22K	22528	$5800	38K	38912	$9800	54K	55296	$D800
7K	7168	$1C00	23K	23552	$5C00	39K	39936	$9C00	55K	56320	$DC00
8K	8192	$2000	24K	24576	$6000	40K	40960	$A000	56K	57344	$E000
9K	9216	$2400	25K	25600	$6400	41K	41984	$A400	57K	58368	$E400
10K	10240	$2800	26K	26624	$6800	42K	43008	$A800	58K	59392	$E800
11K	11264	$2C00	27K	27648	$6C00	43K	44032	$AC00	59K	60416	$EC00
12K	12288	$3000	28K	28672	$7000	44K	45056	$B000	60K	61440	$F000
13K	13312	$3400	29K	29696	$7400	45K	46080	$B400	61K	62464	$F400
14K	14336	$3800	30K	30720	$7800	46K	47104	$B800	62K	63488	$F800
15K	15360	$3C00	31K	31744	$7C00	47K	48128	$BC00	63K	64512	$FC00
									64K	65536	$10000

Note that 32K marks the half-way point for a 64K system. The PET/CBM screen starts here, and generally RAM is below this dividing-line, and ROM above, except in the case of RAM on boards accessed by the memory-expansion ports, and other special cases.

10.2   CBM machine-language monitors - TIM and MLM .

The earliest (BASIC 1) machines have no machine-code monitor in ROM: instead, an assembly listing of machine-code was provided in the manual (pp. 100ff). This occupies the space of a BASIC program, and in fact consists of 10 SYS(1039) followed by the monitor, saved as a single program by extending the end-of-BASIC pointers to include the monitor. (The redundant brackets enclosing 1039 have recurred elsewhere ever since). Since this machine lacks a monitor, entering the program is difficult - there is no way to directly key in the hex information provided! A series of pokes will do the trick, but there are a great many. The easiest way (apart from copying someone else's tape) is to use a loader program like the following, which inputs hex bytes, turns them into decimal, and pokes the result into the correct location. But the BASIC will itself have to be overwritten by poking the values of the bytes relevant to the monitor into place; and, finally, the end-of-program pointer must be altered to include the monitor.

```
10 INPUT "START ADDRESS";S
20 INPUT "BYTE";L$
30 L=0:FORJ=1TO2:L%=ASC(MID$(L$,J)):L=16*L+L%-48+(L%>64)*7:NEXT
40 POKE S,L: S=S+1: PRINT S;: GOTO 20
```

TIM ('tiny monitor') has a 'call entry' of $040F (=1039), and a 'Break entry' point of $0427 (=1063). The latter works only if ($021B), the pointer from BRK, is set to $0427. TIM has similar features to later monitors, but displays PC SR AC XR YR SP only.

BASIC 2 machines have a built-in monitor (called 'MLM', 'machine-language monitor'; rather more dignified than 'TIM'). This displays the interrupt request address (IRQ) in addition to the program counter, status register, A, X, and Y, and the stack pointer. It has a few slight improvements; for example, the decimal flag is cleared. From BASIC, a SYS call to any location with peek-value zero causes the monitor to be entered at the break entry point. SYS 1024 (using BASIC's initial 0) or SYS 4 (using a flag which is only set when a program is running) are the favourites. The call entry point is $FD11 (=64785), although this is not often used.

BASIC 4 has a monitor similar to BASIC 2. The differences are (i) it occupies different locations in ROM; see Chapter 15, starting at D472 in the BASIC 4 column, for comparative locations of the subroutines. (ii) The break entry is modified to abort output to printer; the idea is that a BRK always displays the registers on the screen, without printing strange information onto a printout or listing. Because of this, the call entry point is easiest with a printer: $D472 (=54386) is the relevant address, and OPEN 128,4: CMD 128;: SYS 54386 a typical series of commands to divert the output to a printer. The file-number, larger than 127, ensures a carriage-return character is accompanied by line-feed. If this feature is unwanted, use a lower file-number (e.g. 4) or switch 'auto-line-feed' off.

MLM commands The machine-language monitor contains a table of single-byte commands, which are checked against the actual input. These commands are : ; R M G X L and S, in that order. The monitor also puts a period or full-stop at the start of each line, the sole function of which is to verify that a line is to be considered input into the monitor. The syntax and operation of each command is as follows. (For actual entry addresses and other detailed information, see Chapter 15).

; Alter registers takes 7 parameters and stores them in a buffer from $0200 - $0208, where they remain until (i) they are altered again, or (ii) exit to BASIC ignores them, or (iii) the command G ('GO TO' or 'GO RUN') loads them all into their respective locations and executes machine-code accordingly. Note that *nothing happens until G is entered;* in this way, the altered values are controllable.

$0200	PC High
$0201	PC Low
$0202	Flags
$0203	Accumulator
$0204	X-Register
$0205	Y-Register
$0206	Stack ptr.
$0207	IRQ High
$0208	IRQ Low

MLM values are input according to their absolute position. The following line, for example, inputs the values which are underlined, ignoring the others:

. ;1234567890123456789012**3456789**[RETURN]
     PC   IRQ   SR AC XR YR SP

: Alter memory contents inputs a four-character starting address and eight bytes, which it stores in the eight memory locations from the starting address on. Like the previous command, the values depend on absolute positions.

  .:   0400 00 06 04 00 0A 8A 00 00 00 for example puts 8 bytes into RAM, where they make a BASIC program 10 RUN. Any other values in the line are simply ignored. This routine incorporates a read-back comparison, so that an attempt to write to ROM or non-existent RAM gives .?

R Display registers (no parameters) is always called on entering the monitor. After this, .R has the same effect. This command is normally a preliminary to changing the registers; for instance, suppose the interrupt vector is to be changed from BASIC 4's E455 to a routine at $027A in the first cassette buffer. First, .R displays the text PC IRQ SR AC XR YR SP with

.;   0005 E455 32 32 32 32 FA

or something similar. After moving the cursor up, E455 is overwritten with 027A. Nothing happens until G; .G 0004 causes a break entry, in effect performing SYS 4, and the IRQ is changed, as the screen will show.

M Display memory contents has syntax .M fghj fghj, where the two hex addresses are mandatory, and the second must be not less than the first. Sets of eight bytes are output, preceded by .: and their start address, like this:

.M   0070 0080

.:   0070 E6 77 D0 02 E6 78 AD 8A

.:   0078 0D C9 3A B0 0A C9 20 F0

.:   0080 EF 38 E9 30 38 E9 D0 60

G Go to, Go run has two valid syntactical structures: G alone executes code from the current program counter, PC; G fghj executes code starting at the hex address here represented algebraically as fghj. The effect is similar to a SYS call in BASIC, control being transferred to the new address. However .G 027A differs from SYS 634 in having the capacity to set values for the registers, stack pointer, and so on as a standard feature.

X Exit to BASIC (no parameters) returns to BASIC in direct mode. This is therefore the converse command to a SYS call into monitor.

S Save machine-code to tape or disk has this syntax:
```
.S  "NAME (LENGTH<17)",01,027A,0304
.S  "0:DISK NAME",08,027A,0304
```
for tape (cassette #1 in the example; could also be 02) and disk (drive 0 in the example) respectively. The commas *are* necessary, and help ensure correct input. *Note:* SAVE finishes when the final address is reached; consequently, the 'end address' must be at least one byte beyond the true end of the machine-code.

L Load machine-code from tape or disk has this syntax:
```
.L  "NAME",01
.L  "0:NAME",08
```
for cassette #1, and drive 0 of CBM disk with device #8, respectively.

Adding commands to MLM. BASIC 4's monitor can be represented in a simplified form by a flowchart such as that on the following page, which shows the major features, but omits the details of line-input and so on. If a command doesn't match any of those in the table, for example if .Q was entered, a jump takes place with one level of indirection, to the address stored in the two bytes $03FA and $03FB. The default value in these bytes, put there when the machine is turned on, points to the subroutine in the monitor which prints .? and waits for another input. However, the pointer can be changed to a RAM routine which mimics the action of the monitor, enabling new commands to be added. In this way, extended monitors of much greater versatility can be written for these machines. *Note:* TIM lacks this feature, and must be slightly modified to include it.

10.3 Extended machine-code monitors.

Before we look at the extra commands offered by extensions to MLM, let's briefly survey some of the programs currently available. The first to become widely available was SUPERMON, which is available in versions for BASICs 1,2, and 4. Several versions exist within each type. This program includes work by Bill Seiler, and includes a disassembler based on Steve Wozniak and A Baum's Apple program, with a single-step utility written by J Russo. The whole thing was 'combined, choreographed, and trimmed up' by Jim Butterfield, and written in the form of a relocating loader, so that the code is put into the top of RAM, wherever this may happen to be, and protected from overwriting by BASIC by lowering the top-of-memory pointers. The result is powerful and easy to use. Later versions have a machine-code routine to do the relocating; this is much faster than the earlier BASIC. These are public domain programs; the appendices include listings of the BASIC 2 and BASIC 4 versions, for readers who lack the programs, but not the patience to key them in.

Supermon has itself been modified and extended by other users. The next major monitor was EXTRAMON, by Bill Seiler, which has more features than Supermon, and has been revised as MICROMON and also modified into other forms by non-Commodore software people. The main differences between this program and Supermon are:

(i) Machine-code 'trace' allows breakpoints, so program execution stops when some address is reached a pre-set number of times. Single-step trace is often too slow to be practically useful.

(ii) A relocater enables code to be moved about in memory so that it will run correctly. The command is not as easy to use as this bare description suggests.

(iii) An ASCII dump is provided, so tables and messages can be identified and read more easily.

(iv) Comparatively small, cosmetic, changes include improved scrolling, so that (for example) disassembled output to printers is easier, and better screen editing.

VIC has a monitor, available as a plug-in module, which includes most of these feat-

```
                          ┌─────────────────────┐
                          │     BREAK ENTRY      │
                          ├─────────────────────┤
                          │ (I.E. 'BRK' FOUND)   │
                          │ Save A,X,Y, and SP   │
                          │ on the stack.        │
                          └─────────────────────┘
                                     │
                          ┌─────────────────────┐
                          │ Store "B"; CLC/      │
                          │ CLD/ abort files/    │
                          │ pull SP,Y,X, & A     │
 ┌──────────────────┐     ├─────────────────────┤
 │   CALL ENTRY     │     │ Pull PCL, PCH        │
 ├──────────────────┤     └─────────────────────┘
 │ (SYS call,or RESET├──→              │
 │ with diag. sense │     ┌─────────────────────┐
 │ low). Store "C"  │     │  PRINT C* or B*      │
 └──────────────────┘     ├─────────────────────┤
                          │ Load accumulator     │
 ┌──────────────────┐     │ with "R" command     │
 │ Reset flags/print│     └─────────────────────┘
 │ c.rtn. & '.'/input│              │
 │ line (Accumulator│─→              │
 │ holds command)   │     ┌─────────────────────┐
 └──────────────────┘     │  Compare commands    │
                          │  in table with       │
                          │  accumulator         │
                          └─────────────────────┘
                                     │
                               ╱ Instruction ╲
                              ╱   found?       ╲──── N
                               ╲              ╱
                                     │ Y
                          ┌─────────────────────┐   ┌─────────────────────┐
                          │ Jump to address      │   │ JUMP TO (USRCMD).    │
                          │ in table + 1         │   ├─────────────────────┤
                          └─────────────────────┘   │ Jump to address in   │
                                     │               │ ($03FA). Default     │
                                ╱ Case ╲             │ value prints ?       │
                                ╲ entry ╱            └─────────────────────┘
                                     │
   ┌─────┬──────┬───────┬──────┬─────┬──────┬──────┐
   ↓     ↓      ↓       ↓      ↓     ↓      ↓
   :     ;      R       M      G     X      L,S
  ALTM  ALTR  DSPLYR  DSPLYM  GO   EXIT   LOAD
                                          & SAVE
```

FLOWCHART OF BASIC 4 MACHINE-LANGUAGE MONITOR.

---

ures, plus a disassembler that works backwards.

Among other monitors, BASMON is distributed by IPUG in the U.K., and is essentially Supermon with additions, notably to allow printout to a wide range of printer types, and to accept a larger range of input formats. For example, a table of bytes can be entered directly into the mini-assembler. Like some other monitors the interrupt is redirected to test for the Stop key, so a program in an infinite loop can be aborted (usually) by software. (Section 8.9 gives the method). ULTRAMON is an enlarged version of Extramon; it is an American monitor, containing (according to a review by J Strasma) assorted unacknowledged bits of code from diverse sources, including Compute! magazine. Occasionally one meets HIMON in user-group collections; this is simply one or other of the main monitors ready relocated into the high end of RAM, and naturally this will operate successfully only if the recipient CBM has the same RAM storage fitted.

Extended monitor commands: (i) SUPERMON

Most extended monitors consist of machine-code followed by tables: in Supermon's case these include assembler/ disassembler characters ($,#,X,Y and so on), opcodes, and other data, followed by the commands (T F H D C , A and I) and their respective addresses, which are stored (for reasons connected with the operation of the RTS instruction) one byte smaller than the true value. Some versions have an 'N' command which is not used. Note that each letter is different from those in MLM; this is clearly necessary, since otherwise they would never be executed, but it means that rather odd circumlocutions have to be used to rationalise some commands. 'S' is used for 'SAVE', so single-step has to be something else - in fact, I, perhaps implying '1' step.

, Alter memory, then disassemble screenful of data
The screen-dump (right) shows a Supermon disassembly, consisting of a screenful of disassembled data which was generated by .D 784C. In fact it is the start of BASIC 2's Supermon. Up to three bytes on any line may be changed; when Return is pressed, the new values are put into RAM and the disassemble command re-entered, with the same start point; 748C in the example. The new screen is there-fore very similar to the previous one.

```
., 748C A0 44 7A    LDA $7A44
., 748F 85 34       STA $34
., 7491 AD 45 7A    LDA $7A45
., 7494 85 35       STA $35
., 7496 A0 42 7A    LDA $7A42
., 7499 80 FA 03    STA $03FA
., 749C A0 43 7A    LDA $7A43
., 749F 8D FB 03    STA $03FB
., 74A2 00          BRK
., 74A3 A2 08       LDX #$08
., 74A5 DD 24 7A    CMP $7A24,X
., 74A8 D0 0E       BNE $74B8
., 74AA 86 B4       STX $B4
., 74AC 8A          TXA
., 74AD 0A          ASL
., 74AE AA          TAX
., 74AF BD 2F 7A    LDA $7A2F,X
., 74B2 48          PHA
., 74B3 BD 2E 7A    LDA $7A2E,X
., 74B6 48          PHA
., 74B7 60          RTS
```

A Assemble is a 'tiny' or 'mini' assembler, which con-verts opcodes and addresses into the correct byte form (deducing the address-ing mode from the input format), but not permitting labels or directives or any of the other features to be found on true assemblers. (See Chapter 14 on these). The format is illustrated in the screen dump (right). Pressing Return alone (or entering any erroneous data) causes the error routine to be called, so a query is printed, followed by the monitor's '.' on the following line. While assembly contin-ues, new '.A's are printed at the start of each new line.

```
.A 0300 LDA #$00
.A 0302 LDX #$FF
.A 0304 STA $8000,X
.A 0307 DEX
.A 0308 BPL $0304
.A 030A INC $0306
.A 030D LDA $0306
.A 0310 CMP #$84
.A 0312 BNE $0300
.A 0314 RTS
.A 0315 ?
.
```

C Calculate branch offset uses a format like this:
.C ABCD ABFF , from which the positive or negative offset is calculated. This is one byte only, and is counted as negative if its high bit is set. If the range is too large, the command replies with a query. This was intended to help with branches in a forward direction, but was probably not widely used. Later Supermons (SUPERMON.REL) dropped it.

D Disassemble is a standard disassembler (with $ and # for hexadecimal and immediate mode, respectively), without labels. See the example above under ','.

F Fill memory has three parameters, demonstrated by this example:
F 3000 4000 AA
which has the effect of filling RAM from $3000 to $4000 with $AA. This is useful in clearing an area of RAM. $00 may be used; $EA (NOP or 'no operat-ion') is also popular.

H Hunt memory (for bytes or ASCII) reports all in-stances of a byte combination or a string of ASCII characters, between two add-resses. The two formats are:
.H ABCD CDEF AB[CD][EF] ...
.H ABCD CDEF 'HELLO

I Single-step through program (see example, right)

	PC	IRQ	SR	AC	XR	YR	SP			
.;	748C	E62E	32	04	5E	00	F8			
.I										
A0	8C	5E	00	F8	748F	85	34		STA	34
A0	8C	5E	00	F8	7491	AD	45	7A	LDA	7A45
20	74	5E	00	F8	7494	85	35		STA	35
20	74	5E	00	F8	7496	AD	42	7A	LDA	7A42
A0	A3	5E	00	F8	7499	80	FA	03	STA	03FA
A0	A3	5E	00	F8	749C	AD	43	7A	LDA	7A43
20	74	5E	00	F8	749F	8D	FB	03	STA	03FB
20	74	5E	00	F8	74A2	00			BRK	
A1	77	F8	00	F6	FDD0	A9	0D		LDA	#00
21	00	F8	00	F6	FDD2	4C	02	FF	JMP	FF02
21	00	F8	00	F6	FFD2	4C	32	F2	JMP	F232
21	00	F8	00	F6	F232	48			PHA	
21	00	F8	00	F5	F233	A5	B0		LDA	B0
21	03	F8	00	F5	F235	C9	03		CMP	#03
23	03	F8	00	F5	F237	D0	04		BNE	F23D
23	03	F8	00	F5	F239	68			PLA	

performs GO to the location indicated by the program counter, loading the values of the status register, A,X, and Y, the stack pointer, and the interrupt vector. From then on it displays the contents of the five registers, plus the location, disassembly and corresponding byte(s), at each step in the program. The illustration shows the effect of running the program in memory from 748C; the program is Supermon itself, as listed in the first diagram of the three. Chapter 14 explains the working of this routine. In brief, each instruction is interrupted during its execution; this means that the interrupt is serviced when the instruction is finished. At this juncture the data displayed is collected together. This is the reason for the absence of the very first command. Note, for example, how the Y-register remains unchanged; how BRK causes a jump to a new part of the program; how the status register changes as the contents of the accumulator change; and how a JMP retains all the flags – the status register is unchanged – but alters the program counter.

The speed at which single-stepping takes place is controllable from the keyboard. < causes just one instruction to execute; RVS steps at a constant slow pace; and the space-bar causes rapid stepping. Nothing happens if no key is pressed. Press the Stop key to return to the monitor.

P Printer disassembler as its name implies gives a continuous output to a printer; the syntax is .P 3000 3100 , or whatever other limiting addresses are to be disassembled between. This command is not available on all Supermons; early versions disassemble 22 lines only, as though the printer were the screen.

T Transfer memory moves a block of memory. As the example (right) implies, only three parameters are required, two to delimit the block of memory, and another to indicate the starting-point of the transferred block. The end-point of the new block is implicit in these three values.

```
TRANSFER MEMORY

.T 1000 1100 5000

    TRANSFER MEMORY IN THE RANGE 1000
HEX TO 1100 HEX AND START STORING IT AT
ADDRESS 5000 HEX.
```

Extended monitor commands: (ii) EXTRAMON

EXTRAMON's table of commands is A B D E F H I N Q T U W ' and ,. Some of these are closely similar to the corresponding Supermon commands, namely Assemble, Disassemble, Fill memory, Hunt, and Transfer memory. Extramon's W is Walk code, identical to Supermon's single-step. The new commands are as follows:

B Breakpoint set & Q Quick trace let a program be single-stepped without any results appearing on the screen, so that a graphics program can be watched as it develops at any of the speeds allowed by single-stepping. These are:

```
< FOR SINGLE STEP;
RVS FOR SLOW STEP;
SPACE FOR FAST STEPPING.
```

The process aborts at a breakpoint, when a specified location is entered a pre-set number of times. Each command has alternative syntaxes:
.B 2345 breaks when location $2345 is entered for the first time; and
.B 2345 A0 breaks only when it is entered the 160th time.
.Q traces with the data currently displayable by .R, whereas
.Q 2000 alters the program counter to $2000 (so .R isn't necessary).

E Exit & U Undo are complementary routines which respectively set up and undo an emergency exit routine from an infinite loop. This is valuable when a program appears to be lost in an infinite loop. (The rationale is explained in section 8.9. It involves redirection of the interrupt vector through a test routine, with a jump to the start of the monitor if the test succeeds. This method fails if the interrupt is reset – for example by cassette tape activity – or with 'X2' type crashes, which are not susceptible to this cure). In order to permit normal use of the Stop key, a combined keypress is needed to trigger the return to monitor. 40-column machines rely on both = and Stop, which are checked by examination of $E812. Two keys pressed at once register the logical AND of each separate key: $6F in this case.

The syntax is simply .E and .U in each case.

'Integrate memory' (not to be confused with Supermon's single-step) provides a hex dump of 8 bytes, plus the equivalent in ASCII, so that tables, messages, and so forth are readable. It is analogous to the .M command, except that (i) ASCII is present after the hex, (ii) the colon of .M is replaced by a new symbol, '. Thus I and ' between them perform a similar function to, and may be used in place of, M and : . There is however a slight syntax difference:

.I ABCD continues to the end of memory (FFFF) unless stopped, whereas
.M ABCD elicits the ? error indication.
.I ABCD BCDE behaves like .M ABCD BCDE and ceases at the second address.

```
.I F000

.' F000 54 4F 4F 20 4D 41 4E 59TOO MANY
.' F008 20 46 49 4C 45 D3 46 49 FILESFI
```

N **'New locater'** is a true relocater, not just a memory-move routine like .T . It has six parameters, three of which are identical to those of .T , and which specify the start and end addresses of the chunk of code to be moved and the new starting-point. The end-point is of course implicit in these. Two types of code may be moved: continuous machine-code, or 'word tables', i.e. individual bytes of data not intended to be 'run' as a program. If we consider machine-code first, our chunk of code may not be an entire program; and if it is a subroutine or subprogram, it may be called by code outside itself. So, in addition to an indication that the code is of program type, the remaining parameters define the range to be examined for calls to the relocated code. 'Word tables' can also be referred to externally, but do not themselves require any internal changes of the sort required by absolute addressing. (See Chapter 13 on relocation, for further details). The way to use this command is (i) Put zero bytes into the receiving area of RAM (.N will not move BRK instructions); (ii) Move the tables next, and (iii) Move the code. The examples given in the instruction program are

```
.N 7000 77FF 1000 0400 8000

.N 7000 77FF 1000 0400 8000 W
```

in which the whole of normal RAM from $0400 to $8000 is examined for references to $7000 - $7FFF, and, if any are found, are converted into the range $1000 - $1FFF. Even if many entirely disparate machine-code routines coexist in RAM, this will probably be successful, since it is not likely that any of them will reference each others' tables or subroutines. This routine will not relocate zero-page programs; it will relocate ROM routines, which can be useful, as they are modifiable in RAM, but tables may their references rewritten manually, to make the resulting relocated code compact.

Modifying monitors. An advantage of programs stored in RAM is their accessibility to the programmer. Given some experience in machine-code, it is possible to introduce modifications to carry out functions otherwise unattainable, but at the same time to preserve the input and formatting features which make monitors easier to use than ad hoc pieces of coding. On the other hand, this process is tricky if a source listing of the routine doesn't exist, because machine-code is typically written in a compact form, 'fitting together like polished mahogany' as Churchill wrote. (Of Latin sentences!). Let's consider a concrete example: the 'Hunt' function enables us to search any part of RAM for any sequence of bytes which will fit into a single line;

.H B000 FFFF AD 48 E8

searches BASIC 4 ROM for three bytes which dissamble as LDA E848. What must we do to allow the use of a 'wild card', e.g. 00 in this example:

.H B000 FFFF 85 FB 00 00 85 FC

assuming our only software tool is an extended monitor, and not, say, an assembler with a label-generating disassembler? The object is to permit 'Hunt' in which the bytes in the positions corresponding to 00 may take any value, so STA FB any 2 bytes STA FC will be sought by the particular input line just quoted. As it happens, this is an easy modification to make. To illustrate the method, I'll use BASIC 2 Supermon in

a 32K machine. Other monitors and memory-sizes will therefore not give identical dis-assemblies, but the method should be clear enough.

First we have to find the routines which process H. By examining the tables (at the end of the program) we can find $48 (ASCll for H) within the table of other ASCll characters which make up the additional commands. We then use the relative position of H in its table to deduce the entry address of H, by looking through the programs for a table of addresses which seem to correspond to entry-points in the monitor. lf. as is usual, the address is entered by two PHA commands followed by RTS, we must add 1 to this address. The resulting possible entry-point can be checked by inspecting the code for CMP #$27 (checking for ', as in .H ABCD BCDE 'HELLO). At this point the code separates; if we follow the branch - since we are not concerned with the ASCll string test - we find a block of code from which exit occurs on CMP #$0D. The function of this loop is to fetch the bytes from the screen and store them in a buffer. Finally, we reach the code where comparisons are made. (See the disassembly, right). Location B4 holds the number of bytes being matched; 7BBC and 7BBE compare the contents of memory with the contents of the buffer, and, if these are equal for all X values, the address is printed, to show that a match has been found.

To introduce our modification, all we need do is insert a test for the presence of our 'wild card' byte, and, if one is found, treat it as a genuine match. The second batch of code (see right) is one version of this. Line 7BBB and 7BBD (4 bytes in all) compare the buffer contents with #00 and branch past the memory comparison if #00 has been found. (7BBC may be altered to any other value, should a #00 wild card be unsuitable; namely when #00 bytes are themselves sought). Because of the four-byte patch, some branches have to be slightly changed; moreover, the hunt function for ASCII strings cannot be made to coexist with our new function, without a great deal of rewriting. In practice, the new version would be stored separately under a different name from the original.

All the modifications are marked on the second piece of code, including the chunk of code which was relocated 4 bytes back in RAM. Apart from rewriting the comparisons, the remaining changes only dealt with recalculating branch destinations.

Rather obviously, this type of adjustment can't be made without a fair amount of machine-code experience. In the same way, comparative beginners will not find it a simple matter to decipher the workings of these monitors. Supermon, for example, starts with a series of ten subroutines, which have the following functions:

(i) Reset top-of-BASIC and USRCMD;
(ii) Search for extra Supermon instruction;
(iii) Decrement contents of (FD) or (FB);
(iv) Get next character from input, ignoring spaces;
(v) lnput hex address into (FB), ignoring spaces;
(vi) Skip a character. Get hex address into (FB).
(vii) Print X spaces;
(viii) Increment address in (FB);
(ix) Exchange contents of FB and FC with contents of 020B and 020C respectively;
(x) Test for equality of (020B) and (FB). If equal, Z flag is set; if less, carry flag is cleared.

Section of unmodified code:

```
7BB3  86 B4         STX $B4
7BB5  20 D0 FD      JSR $FDD0
7BB8  A2 00         LDX #$00
7BBA  A0 00         LDY #$00
7BBC  B1 FB         LDA ($FB),Y
7BBE  DD 10 02      CMP $0210,X
7BC1  D0 0C         BNE $7BCF
7BC3  C8            INY
7BC4  E8            INX
7BC5  E4 B4         CPX $B4
7BC7  D0 F3         BNE $7BBC
7BC9  20 6A E7      JSR $E76A
7BCC  20 CD FD      JSR $FDCD
7BCF  20 D5 FD      JSR $FDD5
7BD2  A6 DE         LDX $DE
7BD4  D0 92         BNE $7B68
7BD6  20 D9 7A      JSR $7AD9
7BD9  B0 DD         BCS $7BB8
```

Code after modification:

```
7B7F  C9 27         CMP #$27
7B81  D0 (10)       BNE $7B93
7B83  20 EB E7      JSR $E7EB
7B86  9D 10 02      STA $0210,X
7B89  E8            INX
7B8A  20 CF FF      JSR $FFCF
7B8D  C9 0D         CMP #$0D
7B8F  F0 22         BEQ $7BB3
7B91  E0 20         CPX #$20
7B93  8E 00 01      STX $0100
7B96  20 BE E7      JSR $E7BE
7B99  90 (CA)       BCC $7B65
7B9B  9D 10 02      STA $0210,X
7B9E  E8            INX
7B9F  20 CF FF      JSR $FFCF
7BA2  C9 0D         CMP #$0D
7BA4  F0 09         BEQ $7BAF
7BA6  20 B6 E7      JSR $E7B6
7BA9  90 (BA)       BCC $7B65
7BAB  E0 20         CPX #$20
7BAD  D0 EC         BNE $7B9B
7BAF  86 B4         STX $B4
7BB1  20 D0 FD      JSR $FDD0
7BB4  A2 00         LDX #$00
7BB6  A0 00         LDY #$00
7BB8  BD 10 02      LDA $0210,X
7BBB  C9 00         CMP #$00
7BBD  F0 04         BEQ $7BC3
7BBF  D1 FB         CMP ($FB),Y
7BC1  D0 0C         BNE $7BCF
7BC3  C8            INY
7BC4  E8            INX
7BC5  E4 B4         CPX $B4
7BC7  D0 (EF)       BNE $7BB8
7BC9  20 6A E7      JSR $E76A
7BCC  20 CD FD      JSR $FDCD
7BCF  20 D5 FD      JSR $FDD5
7BD2  A6 DE         LDX $DE
7BD4  D0 92         BNE $7B68
7BD6  20 D9 7A      JSR $7AD9
7BD9  B0 (D9)       BCS $7BB4
7BDB  4C 56 FD      JMP $FD56
7BDE  20 94 7A      JSR $7A94
7BE1  8D 0D 02      STA $020D
```

10.4 Monitors in BASIC.

Although good machine-code routines are unquestionably superior to their BASIC counterparts, there are often advantages in BASIC monitors to offset their slow speed and relatively large memory requirements. (i) They can be transferred between PETs without compatibility problems, except possibly with regard to top-of-memory pointers or printer formatting. This may not be true of machine-code programs, leaving the would-be user with the options of modifying a present program, or going back to BASIC, probably temporarily. (ii) The control of BASIC is familiar: the ability to stop the program and LIST it can be valuable. (iii) BASIC is easily changed to allow for smallish variations and differences. For example, a new printer, with different control characters, can easily be accommodated in BASIC. Decimal numbers can be used in place of hexadecimal, or alongside hexadecimal, if required. Different opcode conventions can easily be implemented. Changes of this sort are far more difficult in machine-code.

Beside these reasons, writing a disassembler is a useful exercise in understanding a chip. The next page has a PET/CBM 6502 version which runs on all models. A simplified flowchart of its logic (see below) shows that it operates by waiting for input of a starting address, then disassembling from that point by peeking the address and converting it into the equivalent opcode form, and looping back to repeat the process with subsequent addresses. As an example, suppose 027A holds this:

```
.:027A  20 E4 FF  F0 FA 60 00 00
```

On entry of starting address = 027A, the program peeks 027A, finding 32 (in decimal). This figure corresponds to "JSR" with address mode 10. This is processed by the subroutine at line 700, which prints the following two bytes, in hex, in the reverse order. Meanwhile, this instruction has 3 bytes (NB=3). So JSR FFE4 is printed, and the address to be peeked is updated, now being 027D, and the process continues. If line 0 includes a poke to lower the top of memory, (see HIMEM in Chapter 5), RAM below $8000 can partly be used for machine-code. Line 10 provides a 'warm start', re-entering the program without clearing the variables. The data storage method used here causes no garbage collection delays with any version of BASIC.

TABLE OF VARIABLES.

CA=current address (usu. of opcode)
L & L$=decimal and hexadecimal numb-
    ers for interconversion
M=addressing mode, coded 0-12
M%()=opcodes' address modes
NB=number of bytes in current
    instruction (always 1-3)
OP=decimal value of opcode
OP$()=table of opcodes by decimal
    value (e.g. OP$(0)="BRK")
P=peek value of current address
  (usually corresponds to opcode)

TABLE OF SUBROUTINES.

100 Print 4-byte hex number given L
200 Print 2-byte hex number given L
300 Convert 4-byte hex number into L
350 Convert 2-byte hex number into L
400 Print hex number from next two
    bytes <u>in reverse order</u>
500 Print next byte in hex
600 - 710 Print address or data after
    the opcode, punctuated in the
    standard way. See the REMs for
    details of modes M, 0-12
2000 Initialises...
2500 is warm start...
3000 Disassembles until spacebar is
    pressed (see line 3070)

```
        ( START )
            │
    ┌───────────────┐
    │  INITIALISE   │
    ├───────────────┤
    │ Set up tables │
    │ of opcodes &  │
    │ their         │
    │ addressing    │
    │ modes         │
    └───────────────┘
            │
    ╱───────────────────╲
   │ INPUT STARTING AD-  │
   │ DRESS FOR DISASSEMBLY│
    ╲───────────────────╱
            │
       ┌────┴────┐
       ▼
    ╱ ADDRESS HOLDS ╲
   │  VALID OPCODE?  │──── N ──┐
    ╲───────────────╱          │
            │ Y                 │
    ┌───────────────┐   ┌──────────────┐
    │ PRINT ONE LINE│   │ NOT AN OPCODE│
    │ OF DISASSEMBLY│   ├──────────────┤
    ├───────────────┤   │  Print ???   │
    │Location,opcode│   └──────────────┘
    │,& poss.data/  │          │
    │address        │          │
    └───────────────┘◄─────────┘
            │
    ┌───────────────┐
    │ UPDATE ADDRESS│
    ├───────────────┤
    │ Address =     │
    │ Addr.+1, 2,or 3│
    └───────────────┘
            │
            └──────► (loop back)
```
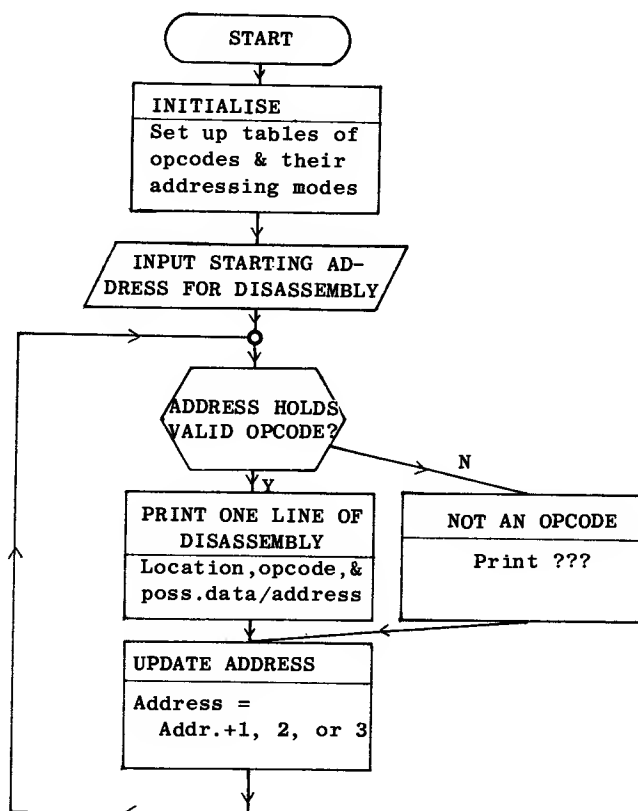
```
0 CLR: GOTO 2000: REM FITS 7K WITH SOME REMS REMOVED
10 GOTO 2500
100 L=L/4096:FORJ=1TO4:L%=L:PRINTCHR$(48+L%-(L%>9)*7);:L=16*(L-L%):NEXT:RETURN
200 L=L/16:FORJ=1TO2:L%=L:PRINTCHR$(48+L%-(L%>9)*7);:L=16*(L-L%):NEXT:RETURN
300 L=0:FORJ=1TO4:L%=ASC(MID$(L$,J)):L=16*L+L%-48+(L%>64)*7:NEXT:RETURN
350 L=0:FORJ=1TO2:L%=ASC(MID$(L$,J)):L=16*L+L%-48+(L%>64)*7:NEXT:RETURN
400 FORK=2TO1STEP-1:L=PEEK(CA+K):GOSUB200:NEXT:RETURN
500 L=PEEK(CA+1):GOSUB 200:RETURN
600 PRINT "(";: GOSUB 400: PRINT ")": RETURN: REM INDIRECT JUMP
610 GOSUB 500: PRINT ",Y": RETURN: REM ZERO PAGE, INDEXED BY Y REGISTER
620 PRINT "(";: GOSUB 500: PRINT ",X)": RETURN: REM INDEXED INDIRECT (ZERO PAGE,X)
630 PRINT "(";: GOSUB 500: PRINT "),Y": RETURN: REM INDIRECT INDEXED (ZERO PAGE),Y
640 L=PEEK(CA+1): IF L>127 THEN L=L-256
642 L=CA+2+L: GOSUB 100: PRINT: RETURN: REM RELATIVE BRANCH
650 GOSUB 400: PRINT ",Y": RETURN: REM ABSOLUTE, INDEXED BY Y REGISTER
660 PRINT "#";: GOSUB 500: PRINT: RETURN: REM IMMEDIATE
670 GOSUB 400: PRINT ",X": RETURN: REM ABSOLUTE, INDEXED BY X REGISTER
680 GOSUB 500: PRINT ",X": RETURN: REM ZERO PAGE, INDEXED BY X REGISTER
690 GOSUB 500: PRINT: RETURN: REM ZERO PAGE
700 GOSUB 400: PRINT: RETURN: REM ABSOLUTE
710 PRINT: RETURN: REM IMPLIED AND ACCUMULATOR
2000 DIM OP$(255), M%(255): SP$="                "
2010 FOR J = 0 TO 150: REM TOTAL OF 151 DIFFERENT OPCODE/MODE COMBINATIONS
2020 READ OP, OP$(OP), M%(OP)
2030 NEXT J
2500 INPUT "ASSEMBLE OR DISASSEMBLE";L$
2510 IF L$="A" GOTO 4000
3000 INPUT "DISASSEMBLE FROM"; L$
3005 GOSUB 300: CA=L
3008 INPUT "DEVICE#";N
3009 OPEN N,N: CMD N,;
3010 L=CA: PRINT L LEFT$(SP$,7-LEN(STR$(L)));: GOSUB 100
3015 P=PEEK(CA): M=M%(P)
3020 IF OP$(P) <> "" THEN 3025
3022 L=P:PRINT "   ";:GOSUB200:PRINT"          ???":NB=1:GOTO3065
3025 NB=2: IF M=0 OR M=5 OR M=7 OR M=10 THEN NB=3
3030 IF M=11 THEN NB=1
3035 PRINT "   ";
3040 FOR K=0 TO NB-1
3045 L=PEEK(CA+K): GOSUB 200: PRINT " ";
3050 NEXT K
3055 FOR J = NB TO 3: PRINT "   ";: NEXT: PRINT OP$(P) " ";
3060 ON M+1 GOSUB 600,610,620,630,640,650,660,670,680,690,700,710
3065 CA=CA+NB
3066 CLOSEN
3070 GET L$: IF L$ = " " THEN 2500
3075 GOTO 3009
5000 DATA 0,BRK,11,1,ORA,2,5,ORA,9,6,ASL,9,8,PHP,11,9,ORA,6,10,ASL,11
5010 DATA 13,ORA,10,14,ASL,10,16,BPL,4,17,ORA,3,21,ORA,8,22,ASL,8,24,CLC,11
5020 DATA 25,ORA,5,29,ORA,7,30,ASL,7,32,JSR,10,33,AND,2,36,BIT,9,37,AND,9
5030 DATA 38,ROL,9,40,PLP,11,41,AND,6,42,ROL,11,44,BIT,10,45,AND,10
5040 DATA 46,ROL,10,48,BMI,4,49,AND,3,53,AND,8,54,ROL,8,56,SEC,11,57,AND,5
5050 DATA 61,AND,7,62,ROL,7,64,RTI,11,65,EOR,2,69,EOR,9,70,LSR,9,72,PHA,11
5060 DATA 73,EOR,6,74,LSR,11,76,JMP,10,77,EOR,10,78,LSR,10,80,BVC,4
5070 DATA 81,EOR,3,85,EOR,8,86,LSR,8,88,CLI,11,89,EOR,5,93,EOR,7
5080 DATA 94,LSR,7,96,RTS,11,97,ADC,2,101,ADC,9,102,ROR,9,104,PLA,11
5090 DATA 105,ADC,6,106,ROR,11,108,JMP,0,109,ADC,10,110,ROR,10
5100 DATA 112,BVS,4,113,ADC,3,117,ADC,8,118,ROR,8,120,SEI,11,121,ADC,5
5110 DATA 125,ADC,7,126,ROR,7,129,STA,2,132,STY,9,133,STA,9,134,STX,9
5120 DATA 136,DEY,11,138,TXA,11,140,STY,10,141,STA,10,142,STX,10,144,BCC,4
5130 DATA 145,STA,3,148,STY,8,149,STA,8,150,STX,1,152,TYA,11,153,STA,5
5140 DATA 154,TXS,11,157,STA,7,160,LDY,6,161,LDA,2
5150 DATA 162,LDX,6,164,LDY,9,165,LDA,9,166,LDX,9,168,TAY,11
5160 DATA 169,LDA,6,170,TAX,11,172,LDY,10,173,LDA,10,174,LDX,10
5170 DATA 176,BCS,4,177,LDA,3,180,LDY,8,181,LDA,8,182,LDX,3,184,CLV,11
5180 DATA 185,LDA,5,186,TSX,11,188,LDY,7,189,LDA,7,190,LDX,5,192,CPY,6
5190 DATA 193,CMP,2,196,CPY,9,197,CMP,9,198,DEC,9,200,INY,11,201,CMP,6
5200 DATA 202,DEX,11,204,CPY,10,205,CMP,10,206,DEC,10,208,BNE,4,209,CMP,3
5210 DATA 213,CMP,8,214,DEC,8,216,CLD,11,217,CMP,5,221,CMP,7,222,DEC,7
5220 DATA 224,CPX,6,225,SBC,2,228,CPX,9,229,SBC,9,230,INC,9,232,INX,11
5230 DATA 233,SBC,6,234,NOP,11,236,CPX,10,237,SBC,10,238,INC,10,240,BEQ,4
5240 DATA 241,SBC,3,245,SBC,8,246,INC,8,248,SED,11
5250 DATA 249,SBC,5,253,SBC,7,254,INC,7
```

Tiny assemblers in BASIC are a little harder to write; each line must be validated, and the addressing-mode deduced from the input. The additional batches of code (see below) may be added to the disassembler, and called from line 2500. They do not provide full validation, but are designed for ease of programming. The rationale of subroutine 800, which determines the mode, is illustrated in this table of addressing modes, numbered as in the program, arranged as examples in columnar form, so that the lengths of each complete instruction and the positions of the punctuation symbols can be seen at a glance.

Lines 3035 – 4045 check for the existence of the opcode and for the correctness of its addressing mode, rejecting "PQR" and "PHA 1234", for example. The subroutines starting at 900 extract the address from the string which was input; and lines 4060 ff. poke the 'assembled' values into RAM. Note that line 4015 puts a " before the input; this enables commas to be accepted without ?extra ignored. The pokes apply to BASIC>1.

	1234567890	MODE
ABS.IND.	JMP (ABCD)	0
0-P,Y	LDA AB,Y	1
(IND,X)	LDA (AB,X)	2
(IND),Y	LDA (AB),Y	3
BRANCH	BEQ ABCD	4
ABS,Y	LDA ABCD,Y	5
IMMEDIATE	LDA #AB	6
ABS,X	LDA ABCD,X	7
0-P,X	LDA AB,X	8
0-PAGE	LDA AB	9
IMPLIED/A	BRK	10

This version does not include '$' symbols before hexadecimal numbers; there is little problem in introducing them, however. In some cases (e,g, line 3010) the TAB function has been replaced by a longer expression, as not all printers process TAB. Line 3009 may need to open a file-number > 127.

```
800 L=LEN(AS$)
805 IF L=3 THEN M=11: RETURN:     REM IMPLIED
810 IF L=6 THEN M= 9: RETURN:     REM ZERO PAGE
815 IF L=7 THEN M= 6: RETURN:     REM IMMEDIATE
820 L$=MID$(AS$,8,1)
825 IF L$="X" THEN M=8:RETURN :     REM ZERO PAGE,X
830 IF L$="Y" THEN M=1:RETURN :     REM ZERO PAGE,Y
835 IF L$="," THEN M=2:RETURN :     REM INDIRECT,X
840 IF L$=")" THEN M=3: RETURN:     REM INDIRECT,Y
845 L$=RIGHT$(AS$,1)
850 IF L$="X" AND L=10 THEN M=7:RETURN :     REM ABSOLUTE,X
855 IF L$="Y" AND L=10 THEN M=5:RETURN :     REM ABSOLUTE,Y
860 IF L$=")" AND L=10 THEN M=0:RETURN :     REM ABSOLUTE INDIRECT
865 IF LEFT$(AS$,1)="B" AND MID$(AS$,2,1)<>"I" THEN M=4: RETURN: REM BRANCH
870 IF L=8 THEN M=10: RETURN:     REM ABSOLUTE
875 PRINT "MODE ?":M=12: RETURN: REM CATCH ALL OTHER INCORRECT ENTRIES
900 P=6:L=4:GOSUB 960:RETURN
905 P=5:L=2:GOTO 960
910 P=6:L=2:GOTO 960
915 P=6:L=2:GOTO 960
920 P=5:L=4:GOSUB 960: GOSUB 300
921 L=L-CA-2: IF L>127 OR L<-128 THEN PRINT "BRANCH?":M=12:RETURN
922 IF L<0 THEN L=L+256
923 RETURN
925 P=5:L=4:GOTO 960
930 P=6:L=2:GOTO 960
935 P=5:L=4:GOTO 960
940 P=5:L=2:GOTO 960
945 P=5:L=2:GOTO 960
950 P=5:L=4:GOTO 960
960 L$=MID$(AS$,P,L): RETURN
4000 INPUT "ASSEMBLE FROM"; L$
4005 GOSUB 300: CA=L
4010 L=CA: PRINT L TAB(7);: GOSUB 100
4015 POKE 527,34: POKE 525,1: INPUT "  ";ASSEMBLER$
4020 IF AS$="END" GOTO 2500
4025 CO$ = LEFT$(AS$,3)
4030 GOSUB 800: IF M=12 THEN 4010
4035 FOR J=0 TO 255
4040 IF CO$<>OP$(J) THEN NEXT: PRINT "OPCODE?": GOTO 4010
4045 IF M<>M%(J) THEN J=J+1: GOTO4040
4050 NB=2: IF M=0 OR M=5 OR M=7 OR M=10 THEN NB=3
4055 IF M=11 THEN NB=1
4060 POKE CA,J: REM POKE OPCODE INTO MEMORY
4065 IF NB=1 THEN 4900
4070 IF M=4 THEN GOSUB 920: IF M=12 THEN 4010
4075 IF M=4 THEN POKE CA+1,L: GOTO 4900
4080 ON M+1 GOSUB 900,905,910,915,920,925,930,935,940,945,950
4085 IF NB=2 THEN GOSUB 350: POKE CA+1,L: REM ONE ADDRESS BYTE ONLY; TWO:-
4090 IF NB=3 THEN GOSUB 300: POKE CA+1,L-INT(L/256)*256: POKE CA+2,L/256
4900 CA=CA+NB:GOTO4010
```

10.5   Introduction to 6502 coding: elementary examples.

I shall assume in this section and subsequent chapters that the reader has a reasonable grasp of hex arithmetic, and has either a BASIC or machine-code monitor available. Equipped in this way, s/he can experiment with the 6502 and become confident in its use. This chip is not particularly easy to program. One of the designers of Commodore's 'Micro-mainframe' has said, among other things, 'If you can program the 6502 you're a genius' and 'After the 6502, everything from then on is easy'. Without going as far as this, it remains the case that machine-language cannot be mastered overnight.

We'll look at some of the simplest instructions and addressing-modes in this section, since progression from these to the more subtle instructions is a natural route which most or all programmers (I suppose) take. Each example can be entered either from the machine-language monitor in the CBM (SYS 4 is usually the easiest method of access), or via a monitor; I have used the convention of prefixing hex numbers with '$', which however should be omitted if the BASIC routine is the previous section is used. There are, of course, many other examples throughout the book: SYS and USR in Chapter 5, and some graphics routines in Chapter 9, ought to be fairly accessible even to quite inexperienced programmers.

Example 1: poking a single character to the screen.

Starting at 027A, enter the following 6 bytes, either with .M, or with a monitor's assembler. The two forms are exactly equivalent to each other, and are simply different ways of writing the same information. The opcodes are more readable - with experience - than the individual bytes, but either form can be deduced from the other. Thus disassembly of the bytes entered by .M will yield the result shown; and inspection of memory after entering the instructions from an assembler will show the same pattern of six bytes as though entered using MLM.

```
.M  027A A9 00 8D 00 80 60 xx xx        $027A  LDA #$00
                                        $027C  STA $8000
(xx may be any value).                  $027F  RTS
```

What does this short routine do? $027A = 634, so SYS 634 causes the code to execute. RTS, 'ReTurn from Subroutine', has the effect of returning to BASIC, so we may execute SYS 826 as often as we like from direct- or program-mode BASIC. Its effect is to print an '@' symbol in the extreme top-left corner of the screen, unless the screen scrolls and the character is lost. This top-left screen location, as we already know, is location $8000 in RAM. This should give a clue to the meaning of ' STA $8000 '. In fact, we can read the code like this: Load the accumulator with #0 (i.e. value zero), store the accumulator in $8000, and return to BASIC. The accumulator, abbreviated to A, and shown by MLM as 'AC' when the registers are displayed, is an 8-bit location within the chip itself, which can therefore be loaded with any value from 0 - FF. Our example has the same effect as poking $8000 with zero.

Can we do more with this? If we POKE 635,1 then SYS 634, the letter 'a' or 'A', depending on the upper- or lower-case mode, appears on the screen; and disassembly shows that our short routine now reads:

```
$027A  LDA #$01
$027C  STA $8000
$027F  RTS
```

because 027B was changed (from BASIC) into 1. This direct-mode statement:

> FOR J=0 TO 255: POKE 635,J: SYS 634: NEXT

runs through the entire gamut of characters: very rapidly changing, they are all displayed one after the other in the top-left corner of the CBM screen. The machine-code has been executed 256 times, each time in a slightly different form, being left with its first instruction changed to $027A LDA #$FF,   because $FF, 255 in decimal, was the last value put into $027B.

When this is fairly clear to you, look at the .M form of the six bytes again. Note that the address $8000 is held *in reverse order,* with $00 preceding $80. This is a feature of all 3-byte commands in the 6502 and many other chips. (But not the 6809). If we modify 027D's contents, SYS 634 will load the accumulator, then store it, not in $8000, but in a location from $8001 - $80FF, i.e. within the first few lines of the screen. Try this:

> FOR J=0 TO 255: POKE 635,J: POKE 637,J: SYS 634: NEXT

which calls the routine 256 times again, but this time prints each character separately on the screen, in ascending order. The final form of the routine, after 255 has been

poked in, is:

```
$027A   LDA #$FF
$027C   STA $80FF
$027F   RTS
```

You should now be able to print any character into any location on the screen, after a certain amount of calculation. Section 9.1's table of PET/CBM screen memory characters lists the values corresponding to each poked character; and section 9.1 has a table of the hex values of the start of each screen-line.

Example 2: Indexed addressing.

Enter the 7-byte routine

.M 027A A9 00 AA 9D 00 80 60 xx

(xx may be any value).

```
$027A   LDA #$00
$027C   TAX
$027D   STA $8000,X
$0280   RTS
```

This introduces two ideas: the idea of the X-register, and its use as an index. TAX stands for 'Transfer Accumulator to X-register'; X is an 8-bit register, similar to the accumulator, into which the contents of A are loaded when TAX is executed. $8000,X is a special notation, meaning the address $8000+X's current contents. That is, whatever value is in X is added to $8000, and the resulting address used in the command. Since X is 8 bits long, the range of addresses spanned is $8000-$80FF in our example. What happens when SYS 634 runs this code? We can read it like this: Load A with the value zero; transfer A to X, so that X now also holds the value zero; store A in the address $8000 indexed by X, which is therefore $8000; and return to BASIC. The effect is to put '@' in the top-left of the screen again.

POKE 635,1: SYS 634 runs this modified version:

```
$027A   LDA #$01
$027C   TAX
$027D   STA $8000,X
$0280   RTS
```

which puts 'a' or 'A' in the top-left-but-one location on the screen. This happens because STA $8000,X when X holds #1 is understood by the 6502 to refer to $8001.

Example 3: Incrementing and branching to generate loops.

Type in the next routine, which introduces a few more introductions. Once again, the PET's monitor and the 'assembler' and disassembler are dealing with identical data; the appearance may be different, but the essence is the same:

.M 027A A2 00 8A 9D 00 81 E8 D0
.M 0282 F9 60 xx xx xx xx xx xx

(xx may be any value).

```
 $027A   LDX #$00    ;LOAD REGISTER X WITH #00
┌>$027C   TXA         ;TRANSFER X TO ACCUMULATOR
│ $027D   STA $80A0,X ;STORE ACC'R IN $80A0,X
│ $0280   INX         ;INCREMENT X-REGISTER BY 1
└─$0281   BNE $027C   ;BRANCH IF RESULT IS NOT 0
  $0283   RTS         ;RETURN (TO BASIC)
```

Now enter SYS 634 from BASIC. The effect is to print all 256 screen values in 256 adjacent locations (i.e. reading across, then down), starting at the fourth line of the screen, or the second with an 80-column screen. How does this work? The instructions have been annotated to help make the process clear. (These comments won't be accepted by many tiny assemblers, so don't try to enter them with the program).

First, X, like A, can be loaded with any 8-bit value; #0 in our example. TXA transfers X to A. At this stage, therefore, both hold #0, or, in terms of bits, 0000 0000. $80A0 is the start of a line on the screen; when X holds #0, the indexed address 80A0,X is therefore calculated to be $80A0. So #0 is poked into $80A0. The next instruction, INX, adds 1 to the contents of the X-register. If the value is #FF, it is incremented to #0. So long as it is not zero, BNE ('Branch if Not Equal to zero) will cause the program to jump to the address specified; in the example, therefore, the code from 027C to 0281 is executed 256 times, the value of X at the start of the loop being incremented from #0 to #FF. After this, the branch fails and RTS returns to BASIC. Note that the branch command, in spite of disassembling to three bytes, nonetheless occupies only two bytes of machine-code. All branches have *relative addressing* in the 6502. This is a fairly simple concept. When the branch is instruction has been read by the chip, the program counter points just after it, to the next instruction – RTS here. The byte following the branch is added to the program counter, and a jump made to the new address, if the branch's test succeeded. In the example, counting back from RTS to TXA gives -7 bytes. This is 256 - 7 = 249 in 2's complement form, or F9.

Example 4: Subroutines and comparisons.

The previous machine-code example is a subroutine, which we called from BASIC. It is also callable from machine-code; the routine which follows calls it 256 times, each time incrementing the address which, when indexed, determines the placing of each character on the screen. Type in the code, retaining that of the previous example, which it uses:

```
.M  0284 20 7A 02 EE 7E 02 AD 7E
.M  028C 02 C9 A0 D0 F3 60 xx xx
```

(xx may be any value).

```
$0284  JSR $027A  ;CALL SUBROUTINE AT $027A
$0287  INC $027E  ;INCREMENT CONTENTS OF $027E
$028A  LDA $027E  ;LOAD ACC'R WITH NEW CONTENTS
$028D  CMP #$A0   ;EQUAL TO #A0 YET?
$028F  BNE $0284  ;IF NOT, REPEAT LOOP
$0291  RTS        ;BUT IF SO, RETURN TO BASIC
```

From BASIC, SYS 644 runs this. ($0284 = 644 in decimal). It takes about .8 of a second to return to BASIC; meanwhile the entire set of characters is printed on the screen 256 times, the starting-points varying from $80A0-$80FF, then $8000-$80A0. INC 027E has the same effect as poking J from BASIC when J is incremented from within the FOR ... NEXT loop. CMP ('CoMPare accumulator'), in our example, compares #A0 with the contents of the accumulator, which holds the incremented value in $027E. The branch back occurs until the accumulator's contents equal #A0, after a complete cycle of 256 increments. Note that it is easier to test for equality with zero (as in the previous example); a comparison with #0 is not usually needed. Note that JSR, which is analogous to GOSUB in BASIC, returns when RTS is encountered. JSR actually means 'Jump Saving Return address', not 'jump to subroutine' as might be thought. The branch instruction is a little longer here, jumping 13 bytes back; this is F3 in 2's-complement hexadecimal.

Example 5: Decrementing and counting.

If we call the previous routine 256 times (taking almost 4 minutes) the pattern of characters repeats. We can use the third and final register to count; this is the Y-register.

```
.M  0292 A0 00 20 84 02 88 D0 FA
.M  029A 60 xx xx xx xx xx xx xx
```

(xx may be any value).

```
$0292  LDY #$00   ;LOAD COUNTER
$0294  JSR $0284  ;CALL PREVIOUS SUBROUTINE
$0297  DEY        ;DECREMENT COUNTER
$0298  BNE $0294  ;BRANCH IF COUNTER NON-ZERO
$029A  RTS        ;BACK TO BASIC
```

SYS 658 sets this going. Note that the Stop key will have no effect, since this works in BASIC only by specially being tested before the execution of each statement. Y is very similar to X, although there are some differences in indexed addressing modes, which are asymmetrically distributed between X and Y. Decrementing, when used to count, is very similar to incrementing, but is often superior from the programming point of view, enabling a few bytes to be saved. Like an increment, this command passes directly between #0 and #FF. In the example, therefore, the value of Y within the loop is #0, #FF, #FE, #FD, ..., #0.

Example 6: Simple program with BASIC driver to look at CBM's memory.

The machine-code subroutine, which we shall call from a BASIC program, moves 256 bytes of memory from some portion of the CBM to the screen. (Note that the originals are not altered in any way by the process of being read). The BASIC program loops until Stopped; any keypress causes the 256 bytes following those currently on the screen to be displayed - except the comma, which moves back. Any other key may be used instead of the comma - see line 10030. Put the keyboard into lower-case mode to make strings, BASIC keywords in ROM, etc., readable.

```
.M  027A A2 00 BD 00 C0 9D 00 80
.M  0282 E8 D0 F7 60 xx xx xx xx
```

(xx may be any value).

```
$027A  LDX #$00
$027C  LDA $C000,X ;LOAD ACC'R FROM INDEXED
$027F  STA $8000,X ;ADDRESS & SAVE IT
$0282  INX
$0283  BNE $027C
$0285  RTS
```

A simple BASIC program is this:

```
10000 L=192 : REM THIS CORRESPONDS TO $C0 OF $C000; IT COULD BE INPUT AS A HEX NUMBER
10010 POKE 638,L: SYS 634      :REM DISPLAY 256 BYTES
10020 GET X$: IF X$="" GOTO 10020  :REM WAIT FOR KEYPRESS
10030 IF X$="," THEN L=L-2      :REM IF SPECIAL CHARACTER, REDUCE ADDRESS BY 2
10040 L=L+1: GOTO 10010         :REM INCREMENT ADDRESS; DISPLAY BYTES ETC.
```

CHAPTER 11: PROGRAMMING THE 6502 MICROPROCESSOR

11.1 Hardware features of the 6502.

This section deals with the following topics:

11.1.1 Addressing modes   11.1.2 The status register; NVBDIZC flags
11.1.3 The program counter, zero-page, and stack
11.1.4 Hardware vectors in the 6502: NMI, RESET, and IRQ
11.1.5 Instructions and opcodes

Note that Chapter 12 has a comprehensive guide to the 6502, prefaced with a table which indicates the meanings of the standard mnemonics. The appendices include a comprehensive set of tables of reference on the 6502.

11.1.1 Addressing modes. The 6502 has 12 or 13 addressing modes, depending on how they are counted. Most of them are quite easy to understand; a few are difficult. Let's first consider how addressing modes are built into the chip. We've seen, in the elementary examples of the previous chapter how an instruction may be followed by one or two bytes, or stand on its own. This is inescapable with this chip: no command extends, in total, over more than three bytes. Now suppose an instruction is encountered while a program runs, and assume it to be a three-byte instruction. It might appear like this: xx 00 80 , referring to the address $8000. Without knowledge of the precise instruction, however, it is impossible to state what addressing-mode is in use; as the previous chapter showed, xx=AD loads the accumulator with the contents of $8000; xx=BD loads it from $8000,X. So the *instruction* has, implicit within it, an addressing mode; and in fact this determines whether the total instruction is 1,2, or 3 bytes long, and the position at which the next instruction is deemed to begin. Note that all addressing modes but one deal in memory locations; typically, the contents of some location may be added to the contents of another, and compared with the contents of a third. Only 'immediate' mode addressing loads an explicit value. This rather abstract property of processors takes some time to grasp. Now we can examine each mode in turn. For convenience, we can divide instructions into those of length 1,2, and 3 bytes:

1-byte instructions have no reference to either address or data, and therefore operate only on hardware features within the chip itself. In a sense, the phrase 'addressing mode' doesn't apply at all, but for consistency these are described as possessing 'implied addressing'. Some of the flags, and some stack operations, can be processed by these commands, as we'll see in the next sections. The accumulator can also be shifted or rotated bit by bit with a single-byte instruction; this is sometimes distinguished as 'Accumulator addressing'.

2-byte instructions consist of an instruction followed by a single byte. If this byte is treated as data, the instruction uses 'Immediate mode'. This is usually indicated by a hash symbol (#) before the data; we had examples in the elementary programs of the last chapter, for example LDA #$00 and LDX #$00. Apart from loading one of the three registers with a value, this addressing mode is used in arithmetic operations, logical operations, and comparisons.

All other 2-byte instructions refer to addresses, not data. There are six different types. We have already used branches in the previous chapter. Their addressing is usually called 'Relative', because off its use of an offset, which, in the 6502, confines the maximum range reachable by a branch to a backward distance of 128 and a forward distance of 127 bytes.

The remaining five 2-byte modes all use *zero-page* addressing. The zero-page is not a feature of the chip itself; it is the section of RAM (or ROM) which is wired to addresses $0000 - $00FF. However, the chip has the facility of enabling the most significant byte, of zero, to be ignored, so that, for example, LDA $34 can be written in place of LDA $0034 . This saves a byte, which in turn shortens programs and increases their speed.*For this reason, the first 256 bytes are usually in great demand in 6502

---

*The appendices include a quick-reference chart of 6502 addressing-modes' timing which condenses the information on timing provided by the manufacturers of the chip. The MOS manuals on the chip have examples to show how the timing is determined by the separate sub-instructions carried out at each clock-cycle. For our purposes it is probably sufficient to note that long, complex instructions are slower than short simple ones.

programs, so that machine-code routines which coexist with BASIC must be careful to take into account BASIC's use of these locations. The five types of addressing are illustrated by these examples:

(i) Zero-page. This is the simplest type: LDA $55 (A5 55) loads the accumulator with the contents of address $55. $55 may hold any value from #0 to #FF. Note the difference between this and the immediate mode instruction LDA #$55 (A9 55) which loads the value #55 into the accumulator, and has no connection with location $55.

(ii) Zero-page indexed by X. LDA $A0,X (B5 A0) loads the accumulator from $A0 plus the contents of the X-register at the time the instruction is carried out. Note that the total of $A0+X is itself treated as a zero-page address; if there is overflow, it is ignored. If X holds #60, A0+60 is treated as 0, not $0100, and the contents of address 0 are loaded into A.

(iii) Zero-page indexed by Y. This is exactly analogous to the latter mode, but the chip is designed so that only two commands can use this mode, viz. LDX and STX .

(iv) Indexed indirect. An example of this type is: LDA ($00,X) (A1 00). The brackets are a convention, which indicates that A is loaded from an indirect address. That is, two bytes point to the address from which the data is taken. Let's assume for the moment that X contains #0, to simplify matters. In effect, we now have LDA ($00) , since the indexing effect of X is zero. Suppose the start of the zero-page is like this:
     .M 0000 01 80 84 02 xx xx xx xx.
Now, LDA ($00) loads A from the address it finds in ($00), which is $8001. So the instruction, in this instance, has the same effect as LDA $8001. In fact, pure indirect addressing like this is not available on the 6502; 'indexed indirect' addressing, as the name implies, allows indexing of the indirect address. Thus, if X were loaded with #2, then LDA ($00,X) has the effect of loading A from the indirect address of $00+2, or ($02). In effect, therefore, with the figures above, LDA $0284 is executed. The command is useful (a) when X=#0, as pure indirect addressing of the zero-page; (b) when a table of pointers exists in the zero-page. The BASIC pointers to the start of BASIC, its end, and its variables, provide an example.

This command is again asymmetrical with respect to the X and Y registers; see STY in Chapter 12 for some comment on this.

(v) Indirect indexed. An example of this type is: LDA ($2A),Y (B1 2A). As with the latter mode, the brackets indicate indirect addressing; if Y holds #0, the effect is identical to that obtained when X holds #0 and LDA ($2A,X) is executed. Apart from this special case, however, this mode is post-indexed by Y; that is, the indirect address is calculated, *then* Y is added, and the resulting address is the object of the processing. To show how this works, consider the data shown above, of four possible bytes at the very start of RAM. Now, LDA ($00),Y loads from $8001 + Y, so the 256 bytes from $8001 to $8100 can all be accessed, depending on Y's value. Chapter 9 has some graphics examples which use this mode. See for example the routine to plot vertical bars, histogram-fashion, in section 9.3 .

3-byte instructions in the 6502 always consist of an instruction followed by a 2-byte address. (Since the accumulator, for example, is an 8-bit register, 'LDA #$1234' makes no sense). There are four interpretations of the address:

(i) Absolute. This mode is a simple reference to a 2-byte address, as in:
LDA $1234   or   LDA $8000 or   LDA $0012.

(ii) Absolute, indexed by X. The contents of X are added to the address to give the actual referenced address. Thus, if X holds #$50, LDA $8000,X loads the accumulator from $8050. As with zero-page indexing, the maximum value cannot exceed the legitimate range, so LDA $FFF0,X when X holds #$11 loads the accumulator from $0001, not from the non-existent $10001.

(iii) Absolute, indexed by Y. This is exactly analogous to the previous mode.

(iv) Absolute indirect. The 6502 has one instruction only with this mode, namely JMP. An indirect jump transfers the program's flow of control to a new address; this is found from the contents of the address pointed to, by the indirect command. An example is perhaps in order here: JMP ($0000) with the zero-page data we've used before has the same effect as JMP $8001; and JMP ($0001) jumps to $8480, and so on. This command is useful when a table of addresses exists in a block, like the three vectors at the top of RAM, without JMP commands between the addresses. The RESET vector at (FFFC) can be called by JMP ($FFFC) irrespective of BASIC ROM. Tables with JMP, the kernel for example, do not need this.

11.1.2 The status register and N,V,B,D,I,Z and C flags

The **status register** or 'Processor status register', recorded as 'SR' by the PET/CBM monitor, is an 8-bit register within the chip containing seven status bits or 'flags'. The eighth bit, bit 5 in fact, is not used, and is fixed at 1. The flags are intimately related to the chip's operation, at least three being in a position in the register which is directly related to their function. A chart in the appendices enables a status register to be separated into its individual bits; from it, an SR of A4 (say) can be immediately recognized as having its N and I flags on, and its other flags off. Before we examine each flag's purpose, it is important to make clear the idea that the flags do not change unless a command explicitly alters them; the decimal (D) bit for example typically remains off throughout the entire operation of all the programs which a CBM runs. The appendices have a double-page table of opcodes, which includes a list of flags, those which are altered by a command being marked. All the blank spaces in this part of the table mark flags which are left unchanged. LDA alters both the N and Z flags, but no others, for example. Where a flag is marked, a '1' or '0' implies that the command explicitly sets this value in the flag. 'CLD' for example 'CLears the Decimal' flag; the flag is set to 0, irrespective of its value before. The flags marked as 'N','V','Z' and so on may be set in either direction, depending on the result of the processing. LDA sets the 'Z' or 'Zero' flag true when the accumulator loads the value of zero, and so on.

The **'N', 'Negative' flag (bit 7 of SR)** usually holds bit 7 of the result of an operation, or of an intermediate result, and can be pictured as a direct copy of bit 7 into the status register. **LDA #$D3** loads #D3 into A, and is a command which affects N. Since D3 = 1101 0011 in bit terms, with bit 7 high, the N flag is turned on by the instruction. The word 'negative' is based on the 2's complement idea: where this is in use, N = 1 shows a negative, and N = 0 a positive, number. In other cases the flag may be used in a conventionalised sense: hardware may be wired to bit 7, and BMI and BPL (Branch on MInus and Branch on PLus) used to detect whether bit 7 is high or low. See BMI and BPL in Chapter 12. These branches depend on the state of N; when on, BMI is taken; when off, BPL. So 'BPL' really means 'branch if bit 7 is low', or 'branch if zero or positive'.

The **'V', 'internal oVerflow' flag (bit 6 of SR)** is probably the least-used 6502 flag, because of the infrequent use of 2's complement arithmetic outside the chip's own branch istructions. See the entry under BVC in Chapter 12. V is altered by only five instructions, including addition (ADC) and subtraction (SBC).

The **'B', 'Break' flag (bit 4 of SR)** is usually set only on BRK and when the SR is examined after having been pushed on the stack. Its purpose is to enable a BRK instruction to be distinguished from an interrupt, since both jump to the same address. Section 11.1.4 explains the hardware vectors on the 6502.

The **'D', 'Decimal calculation mode' flag (bit 3 of SR)** changes the mode of operation of the chip from hexadecimal arithmetic to 'BCD' or 'binary-coded decimal'. See SED in Chapter 12. When bit 3 of the status register is on, the effect on addition is to add 6 to the low nybble if its result exceeds 9, and to add 6 to the high nybble if that result exceeded 9. When this bit is set - see SYS in Chapter 5 for an example - the normal arithmetic operations of the PET become confused. For this reason the built-in monitor takes the precaution to clear the decimal flag.

The **'I', 'Interrupt disable' flag (bit 2 of SR)** prevents the interrupt request line from causing the 6502 to service an interrupt, when it is on. When off, any interrupt which uses the IRQ line will cause an interrupt, as explained in 11.1.4. In this way, the program can be made to ignore interrupts of this sort until it is ready to deal with them. See SEI and CLI in Chapter 12. The CBM uses a regular interrupt to read the keyboard. If this is redirected to a new program, it may be necessary to set the disable flag to ensure that the interrupt is not itself interrupted.

The **'Z', 'Zero result' flag (bit 1 of SR)** is set by most of the instructions which set N. Instead of registering the result in a single bit, Z in effect logically ORs together all the bits of a result; if this process gives a value of zero, the Z bit is set, to show a zero result; otherwise, when Z is off, the result was non-zero. The notes to BEQ and BNE in Chapter 12 expand on this.

The **'C', 'Carry' flag (bit 0 of SR)** is primarily of use in additions or subtractions, where its function is similar to the carry which is used (or was used, before cheap

calculators) to denote overflow from a column of figures to a more significant column. BCC in Chapter 12 has notes on this flag, and an example involving addition; CLC, SEC, and BCS are other commands involving this flag.

11.1.3 The program-counter, zero-page, and stack.

The program-counter is a pair of 8-bit registers, usually represented as PCL and PCH, connected to appear like a 16-bit register. This counter keeps a record of the current RAM or ROM location of the command being executed. Because of its 8-bit structure, updating in order to point to consecutive items of data takes 2 clock cycles, which is why the fastest instructions take 2 cycles with this chip. The registers can only be accessed indirectly; BRK or an interrupt causes the value to be saved, as does JSR, where the 'Save Return address' refers to the program counter. RTS and RTI accordingly both load their saved data into the program counter, so these commands can be used to load values directly into PC. CBM BASIC uses 'RTS' to jump to the addresses at which BASIC keywords are run; and the MLM uses 'RTI' to load the status register and the program counter when .G is run. The same effect could be got - perhaps more easily - with a JMP instruction, whose function is solely to reload PC with some new value.

The zero-page, as we've seen in 11.1.1 on addressing, is the section of memory from $00 - $FF. The *stack* is a hardware feature of the chip. It uses page 1, i.e. $100 - $1FF of RAM. (Note that memory is divided into 256-byte *pages*. Some machine-code instructions take an extra clock cycle to compute branches and indexed addresses if the result happens to cross the boundary of a page). The stack is difficult to understand, for several reasons. In the first place, the area of RAM from $0100 - $01FF which holds the stack also doubles as normal RAM; it is not reserved for the stack alone. Secondly, bytes 'pushed' onto the stack are added at the *bottom* of the present stack. Thirdly, the *stack pointer*, in order to be consistent, behaves in an apparently inconsistent way, operating differently when pulling than pushing. The stack pointer is another 8-bit register, which in the 6502 is always preceded by $01, and which keeps track of the current stack of data. (The leading $01 forces the pointer into the range $0100 - $01FF). Two complementary pairs of instructions exist on the 6502: PHA and PLA, and PHP and PLP. Any of these instructions followed by the other member of the pair must leave the accumulator or processor status flags unchanged. Because of this, the sequence store data/ update pointer is used with a 'push', and the sequence update pointer/ load data with a 'pull'. Chapter 12 has examples and comment under PHA, PHP, PLA, and PLP. Four other commands operate on the stack automatically: these are JSR and its complement RTS, BRK, and RTI. The stack pointer is accessible by transfer with the X-register only: TSX and TXS respectively transfer the current stack pointer (omitting $01-) to X, and vice-versa.

11.1.4 Hardware vectors in the 6502: NMI, RESET, and IRQ.

The 6502 has, like many microprocessors, a clutch of specially reserved addresses at the top of ROM. On activation of the non-maskable interrupt line, the reset line, or the interrupt request line while the interrupt-disable flag (I) is off, the chip automatically sets the program counter to the address in (FFFA), (FFFC), and (FFFE) respectively. The designer of the system has to ensure that suitable processing routines exist at the destination addresses. For example, BASIC 4 has

.M FFFA 49 FD 16 FD 42 E4 xx xx,

so the effect of causing a non-maskable interrupt can be investigated by disassembling from $FD49; the reset sequence (triggered at switch-on) follows $FD16; and ordinary maskable interrupts are processed from $E442. These three vectors are the 6502's total complement of special vectors; some chips have many more. The BRK ('BReaK') instruction in fact shares the IRQ vector, so a routine has to be used to work out whether a BRK or interrupt caused the execution of the routine: this is easy to find in the PET/CBM by disassembling the machine-language from (FFFE). If the entry on the stack has its BRK flag (B) set, an instruction like SYS 4 is assumed, and the monitor is entered; otherwise, the regular keyboard and screen servicing routine is entered. (Note: the PET, with BASIC 1, is different - see Chapter 15). An interrupt has a similar effect to BRK, in that it pushes the program counter and status register onto the stack. (But not A,X, and Y). In this way, RTI (ReTurn from Interrupt) can continue execution of the interrupted routine when the interrupt has been serviced. For this reason, BRK and RTI respectively push and pull the same data on the stack, and so carry out their operations in the opposite sense from each other.

The NMI vector (which is usable in BASIC>1) is sometimes used to supply a reset switch to the CBM, so that infinite loops in machine-code can be aborted. See section 8.9. The RESET vector is self-explanatory; JMP (FFFC) calls it, and can be

used to erase a program after it has been used, for instance. IRQ is used in the PET and CBM to scan the keyboard, the frequency depending on the VDU screen. Setting the interrupt disable flag (with SEI) turns this off indefinitely. In process-control systems, most interrupts use IRQ, and can be temporarily ignored by disabling the servicing routines; the NMI is reserved for emergency use as a rule. The pins on the 6502 which correspond to the vectors are pins 6, 40, and 4 respectively.

11.1.5 6502 instructions and opcodes.

An *opcode* ('operation code') is a mnemonic, intended to make machine-language relatively easy to read. 6502 opcodes are all three letters long (unlike e.g. Z80 opcodes) which makes for tidy assembler and disassembler listings. An alphabetical list of opcodes in the next chapter explains the workings of each instruction. It is prefaced by a table to summarise the mnemonics' meanings. An opcode bears the same relation to machine-language that a BASIC keyword does to its tokenised form. Just as a BASIC keyword is stored in one byte, but LISTed by a special routine which expands it into a recognisable word, so a machine-code instruction occupies one byte only, and is converted into a three-letter opcode for the sake of readability. Although the opcodes are standard, there is nothing to stop anyone using their own, for example by modifying the BASIC program in the previous chapter. This may in fact be helpful as a learning aid, although it would be unorthodox.

There are 56 opcodes, some with one addressing mode, some with as many as eight. We can group them by function as follows:

i **Add/ Subtract** ADC ('Add with carry') and SBC ('Subtract borrowing carry') are the 6502's arithmetic functions. Both addition and subtraction are carried out on all 8 bits, using the carry flag (C) for overflow. 2's complement arithmetic is *not* used,*but flags are present which enable it to be implemented. Binary-coded decimal (BCD) arithmetic is available if it's wanted.

ii **Branches** The 6502 has eight branches, all conditional on the status of a flag, and all having a single-byte 2's complement offset. The instructions are: BCC & BCS, BNE & BEQ, BPL & BMI, BVC & BVS, and the branch is taken if the C, Z, N, and V flag is off/on respectively.

iii **'Break'** BRK causes an unconditional jump to (FFFE), having first saved both bytes of the program counter and the status register on the stack.

iv **Comparisons** CPX, CPY, and CMP enable X, Y, and A to be compared with data or with memory contents. The data or memory is subtracted from X, Y, or A and flags are set, without storing the result. N, Z, and C are set, so a comparison may be followed by any branch (except BVC or BVS) to test the comparison.

v **Data transfers** Data can be loaded from RAM or ROM by LDA, LDX, or LDY; it can be stored in RAM by STA, STX, or STY. These few commands are extended in power by being equipped with a large number of addressing modes.

vi **Decrements/ increments** alter X, Y, or memory by subtracting/ adding 1 bit, setting N and Z. The instructions are DEX, DEY, DEC and INX, INY, INC.

vii **Flag clear/ set** enable some flags to be altered at will: CLC, CLD, CLI, and CLV clear flags C, D, I, and V; SEC, SED, and SEI set flags C, D, and I.

viii **Jumps** JMP acts like GOTO in BASIC. JSR acts like GOSUB, with RTS the equivalent of RETURN. JSR saves 2+current address on the stack.

ix **Logical operations** AND, EOR ('Exclusive or') and ORA ('Inclusive or') perform binary logical operations on the Accumulator and data or memory, retaining the result in A, and setting N and Z. BIT sets 3 flags.

x **No operation** NOP does nothing

xi **Return** RTS returns to the instruction following JSR by jumping to the address currently on the stack + 1. RTI jumps to the address on the stack and also loads the status register from the stack.

xii **Rotate/ shift** ROL and ROR act on the Accumulator and the C flag (a 9-bit rotation). ASL ('Arithmetic shift left') and LSR ('Logical shift right') also involve A and C, but do not rotate C, so that bit 0 with ASL and bit 7 with LSR are always set to zero. Flags N, Z, & C are set.

xiii **Stack operations** are PHA, PHP, PLA, and PLP. These are explicit operations on the stack, but BRK, JSR, RTS, and RTI also use the stack. TSX and TXS allow the stack pointer to be found/ set respectively.

xiv **Transfers between registers** Six instructions allow transfers between any two neighbours of Y,A,X, and S. The opcodes are TYA & TAY, TAX & TXA, TXS & TSX.

---

*For example, CLC/ LDA #50/ ADC #50 leaves A holding #A0, which is obviously not a negative number. However, the results are consistent with 2's complement arithmetic; in this case, the N flag signals overflow into the sign bit.

11.2 Software methods using the 6502.

This section deals with the following topics:

11.2.1 Incrementing 2 bytes. The best method is

```
INC LOBYTE
BNE +2 or +3; depending on position in RAM of hibyte
INC HIBYTE
```

where the more significant byte is incremented only if the low byte has just been incremented from #FF to #0. Note that a branch after INC HIBYTE can be put in to test for the transition from FFFF to 0000, if this is important.

11.2.2 Decrementing 2 bytes. There is no unique test for a decrement from #0 to #FF so this is less simple than incrementing two bytes:

```
LDA LOBYTE
BNE +2 or +3
DEC HIBYTE
DEC LOBYTE
```

DEC HIBYTE may be replaced by LDA HIBYTE/ BEQ EXIT/ DEC HIBYTE if it is required to go to EXIT when the two bytes hold #0000.

11.2.3 Adding 2-byte pairs. The carry bit (C) is used to carry from the low to the high byte; if the carry bit is set on exit from the routine, overflow has taken place from the high byte; i.e. the result is too large for 16 bits.

```
CLC              e.g. CLC
LDA LO1               LDA $2A
ADC LO2               ADC $01
STA LO2               STA $01
LDA HI1               LDA $2B
ADC HI2               ADC $02
STA HI2               STA $02
```

Note that an addition leaves the result in the accumulator; it must therefore be stored in some way if it is to be kept. In the example, the 16 bits made up of HI1 and LO1 are added to the 16 bits of HI2 and LO2; the result is stored in HI2 and LO2, but could equally well be stored in any other locations. The example with numerals adds the contents of ($2A) to those of ($01), leaving the result in ($01). For instance, if $2A holds #34 and $2B holds #AB, ($2A) contains the 16-bit value #AB34. And if $01 holds #FF and $02 holds #11, ($01) contains #11FF. The machine-code will leave ($2A) unaltered, but change ($01) to #BD33.

11.2.4 Subtracting 2-byte pairs. The carry bit is usually set before subtraction. If it is cleared on exit, the result is negative; the first address held data of smaller value than the second. The general reasoning is similar to that in the previos instruction.

```
SEC              e.g. SEC
LDA LO1               LDA $2A
SBC LO2               ADC $01
STA LO2               STA $01
LDA HI1               LDA $2B
SBC HI2               SBC $02
STA HI2               STA $02
```

11.2.5 Multiplying single bytes. Multiplication and division are not functions present on the 6502. The following example uses two zero-page bytes only; the answer is left in the same two bytes, erasing the original values. The actual zero-page locations are those for the NMI vector, which is unlikely to be used: if it is, switch to other locations. It is possible to test a routine of this sort exhaustively, which is quite unusual. The routine is relocatable; the actual values given (i.e. in the middle of cassette buffer #1) can of course be changed, but as the routine stands this BASIC program can demonstrate the machine-code:

```
10 INPUT X,Y: POKE 136,X: POKE 137,Y: SYS 768: PRINT PEEK(136)+256*PEEK(137)
```

```
$0300  18       CLC
$0301  A9 00    LDA #$00
$0303  A2 08    LDX #$08
$0305  6A       ROR A
$0306  66 88    ROR $88
$0308  90 03    BCC $030D
$030A  18       CLC
$030B  65 89    ADC $89
$030D  CA       DEX
$030E  10 F5    BPL $0305
$0310  85 89    STA $89
$0312  60       RTS
```

This routine can be expected to take about 165 clock cycles *on average* – depending on the number of internal branches. So about 6000 multiplications of this type can be performed in one second. The method of operation relies on the ROR instruction, which is used both to detect bits in $88 and to store both bytes of the result.

11.2.6 Division of 2 bytes by a single byte. The routine that follows has the converse effect to the latter routine. A 16-bit number is divided by an 8-bit number; the answer is assumed to be in the range 0-255, and this is normally taken care of by the way the routine is used. The division is therefore a slight cheat: the multiply routine allows any pair of single-byte values (i.e 0 - 255) to be multiplied, giving results from 0 - 65025 (=FF01), but division, although permitting any of these calculations to be performed in reverse, won't work correctly with (say) 65025 divided by 1.

Three zero-page bytes, ($88) for the numerator and $8A for the denominator, are assumed here, partly to make the routine easily workable from BASIC, like this:

```
10 INPUT X,Y: POKE 136,X-INT(X/256)*256: POKE 137,X/256: POKE 138,Y
20 SYS 768: PRINT "RESULT =" PEEK(136) " AND REMAINDER =" PEEK(137)
```

```
$0300  18       CLC
$0301  A2 08    LDX #$08
$0303  A5 89    LDA $89
$0305  26 88    ROL $88
$0307  2A       ROL A
$0308  B0 04    BCS $030E
$030A  C5 8A    CMP $8A
$030C  90 03    BCC $0311
$030E  E5 8A    SBC $8A
$0310  38       SEC
$0311  CA       DEX
$0312  D0 F1    BNE $0305
$0314  26 88    ROL $88
$0316  85 89    STA $89
$0318  60       RTS
```

The numerator is held in the analogous locations to the multiply routine, so the two may be used together. On average, a division takes about 185 microseconds, about 5400 per second. An exhaustive test of this routine is much slower than obtains with a multiplication routine, since all the remainders need to be checked. Note that, at the start, the high byte of the numerator must be less than the denominator, because an answer less than 256 is taken for granted. For each of eight loops, the denominator is doubled by a pair of leftward rotations. and the numerator subtracted, if this is possible; if not, C is clear. The low byte is replaced by the result; A is the remainder.

11.2.7 Comparing 2-byte pairs. The trick in this case is to avoid the comparison instructions, using SBC instead, which (unlike CMP &c) retains the result of subtraction without only setting flags. This enables us to write a routine to test for equality as well as for the less-than-or-equal-to condition which the carry bit tests. For example:

```
        SEC
        LDA LO1    ;LOW BYTE OF THE FIRST VALUE'S LOCATION
        SBC LO2    ;LOW BYTE OF THE SECOND VALUE'S LOCATION
        STA TEMP   ;TEMPORARY STORE FOR RESULT (MAY BE ZERO)
        LDA HI1    ;HIGH BYTE OF THE FIRST VALUE'S LOCATION
        SBC HI2    ;HIGH BYTE OF THE SECOND VALUE'S LOCATION
        ORA TEMP   ;RESULT ZERO IF BOTH ACC'R AND TEMP WERE ZERO
```

If the contents at the first address equalled the contents of the second, the Zero flag (Z) is set. If the contents at the first were *less* than those at the second, the Carry flag is clear; and if the contents at the first address exceeded those at the second, C is set. So BEQ, BCC, and BCS respectively test for =,<, and >.

11.2.8 Negation by 2's complement. The rule for 2's complement is 'flip the bits and add 1'. As we've seen, this ensures that a number and its conventionalised negative add to exactly zero, with overflow. 8-bit numbers which add to 256 (=$0100) are 2's complements; so are 16-bit bumbers adding to 65536 (=$10000). AN 8-bit number is easily 2's complemented:

```
        LDA NUMBER
        EOR #$FF    ;THIS REVERSES EACH BIT
        CLC
        ADC #$01    ;OR SEC/ ADC #$00
```

If this process is performed twice on a number, the original number is obtained; there is no need to subtract 1 and then flip bits. The 16-bit equivalent involves EOR #$FF with both bytes, and then a 2-byte addition of #1.

11.2.9 Other 2-byte operations. Since there are two registers in addition to the accumulator, it is often possible to write compact code using these to store the two bytes of a 16-bit address. For example, suppose ($2C) contains a pointer which we wish to decrement and store in ($0300). We can combine these objectives like this:

```
        LDX $2A
        LDY $2B
        DEX
        BNE LABEL
        DEY
LABEL   STX $0300
        STY $0301
```

11.2.10 Loops. Loops generally use either the X-register or Y-register as a counter, and often as an offset too. Three example programs below print the word 'HELLO' at the top-left of the screen. In each case, X is a counter, but is also used as an offset to select one letter from the 5 bytes of 'HELLO' which are stored in RAM. In the first example, X is incremented (with INX), and needs CPX #05 to test for the end of the text. The second and third examples use DEX, which is more elegant, since DEX sets the Z flag (on transition from #1 to #0) and also the N flag (on transition from #0 to #FF). BPL allows X to take the value #0; BNE does not, and the differences between the routines (e.g. X starting at #4 or #5, the use of 7FFF,X or 8000,X) are the result of this. Possibly BPL gives code which is easier to follow.

$027A	A2	00		LDX #$00	$027A	A2	05		LDX #$05	$027A	A2	04	LDX #$04
$027C	BD	88	02	LDA $0288,X	$027C	BD	85	02	LDA $0285,X	$027C	BD	86	02	LDA $0286,X
$027F	9D	00	80	STA $8000,X	$027F	9D	FF	7F	STA $7FFF,X	$027F	9D	00	80	STA $8000,X
$0282	E8			INX	$0282	CA			DEX	$0282	CA		DEX
$0283	E0	05		CPX #$05	$0283	D0	F7		BNE $027C	$0283	10	F7	BPL $027C
$0285	D0	F5		BNE $027C	$0285	60			RTS	$0285	60		RTS
$0287	60			RTS	$0286	48	45	4C	.BYTE 'HELLO'	$0286	48	45	4C	.BYTE 'HELLO'
$0288	48	45	4C	.BYTE 'HELLO'									

Section 11.2.14 has other methods for accessing tables of data. Nested loops can be written for the 6502; this is a delay loop, which uses all three registers (A,X, and Y) as counters. When run from BASIC, it shows how setting the interrupt disable flag turns off the CBM's internal BASIC clock. With the addresses given here,

$027A	78	SEI
$027B	18	CLC
$027C	A9 00	LDA #$00
$027E	A2 00	LDX #$00
$0280	A0 00	LDY #$00
$0282	E8	INX
$0283	D0 FD	BNE $0282
$0285	C8	INY
$0286	D0 FA	BNE $0282
$0288	69 01	ADC #$01
$028A	D0 F6	BNE $0282
$028C	60	RTS

PRINT TI: SYS 634: PRINT TI

gives equal or virtually equal values for TI, in spite of the delay, which is over a minute with the values as they appear in the program. (Loading A with other values alters the length of the delay in direct proportion). Note that the innermost loop runs 256 times for each execution of the next loop, which in turn runs 256 times as often as the outermost loop. To estimate the duration of such a loop, the outer loops can often be ignored (as they contribute less than ½% to the overall time). Reference to the timing charts shows that INX takes 2 clock cycles, and BNE with a successful branch, not across a page boundary, takes 3 cycles. So the total delay is about $256^3 * 5 /1000000$ seconds, or 1 minute 24 seconds.

11.2.11 Saving and restoring the zero-page. This is sometimes a useful trick, either to enable a long machine-code program to run in tandem with BASIC, or when using certain ROM routines which would otherwise change BASIC pointers. The TRACE routines in Chapter 5, for example, do this. The point about machine-code programs is that they are likely to be faster if the zero-page is used; in any case, a program may be written already, using these locations, and the effort of rewriting to fit CBM zero-page usage may not be worthwhile. The routines, one to save in RAM, the other to restore, are simple enough; the only difficulty is to ensure the inviolability of the RAM area in which the zero-page is stored. Usually this will be in the top of RAM, below the screen.

```
           SAVE ZERO-PAGE                              RESTORE ZERO-PAGE

           LDX  #$00                                   LDX  #$00
    LABEL  LDA  $00,X                           LABEL  LDA  STORE,X
           STA  STORE,X                                STA  $00,X
           INX                                         INX
           BNE  LABEL                                  BNE  LABEL
```

11.2.12 Memory-move with several pages. The maximum range spanned by an 8-bit register, used for post-indexing, is 256 bytes, so moving (say) 1K bytes normally means moving four batches of 256 bytes, with either a counter to run from 4 to 1 or 0 to 3 or whatever, or a test on the actual addresses. Section 11.2.16 has examples of ways to do this, and includes timing comparisons.

11.2.13 Using shift and rotate commands. Shift instructions (ASL, LSR) are useful whenever a byte is to be processed bitwise. After shifting, the Carry flag holds the latest bit to be shifted, so BCC or BCS processes the routine appropriate to a high or low bit respectively. In Chapter 14, a VIA program displays the contents of all 16 registers in this chip, in bit form, using this method. Routines to convert parallel data to serial (e.g. RS232 output and input routines) typically load the byte into A, then shift it 8 times, sending individual bits serially down the connecting wire, with a few other bits for parity, etc. Rotations involve 9 bits, including C, and so may be used to inspect bits while eventually returning a location to its original condition, or, more subtly, entering some new, dependent value; the routines in 11.2.5 and 11.2.6 for multiplication and division do this. Rotations involving A are much faster than those operating on memory locations, so it is good practice to work on A where possible, for example in 16-bit calculations. On the next page is a typical calculation sub-

routine. It shows how a process of
successive shifts and rotates left
(which in effect multiply by 2) can
be used in computations. Three
parameters are involved:

$00 holds an X-coordinate (0 - 39),
$01 holds a Y-coordinate (0 - 24),
and $02 holds #$20.

```
LDA $01    ;A holds Y-coordinate
ASL A      ;A holds 2*Y-coord. (0 - 48)
ASL A      ;A holds 4*Y-coord. (0 - 96). C=0
ADC $01    ;A holds 5*Y-coord. (0 - 120).
ASL A      ;A holds 10*Y-coord. (0 - 240)
ASL A      ;A = 20*Y-coord (0-480); overflow C
ROL $02    ;#4000 + overflow
ASL A      ;A = 40*Y-coord (0-960); overflow C
ROL $02    ;#8000 + overflow
STA $01    ;($01) holds #8000 + 40*Y-coordinate
```

The object is to set up a pointer
to the Xth column of the Yth row
of a 40-column screen. This can be
done with a table (and in fact it is
faster with that method). However,
the method demonstrated here cal-
culates #8000 + 40*Y, putting the
result in ($01), so an instruction
like   LDY $00/ LDA ($01),Y
references the correct position on the screen. This is used in SET (see Chapter 5).

11.2.14 Jump tables, data tables, and address tables. *Jump tables* contain JMP in-
structions and addresses. The point is to provide a table of values which are either
(i) easy to remember, or (ii) easy to program (the actual addresses to be used only
being filled in later), or (iii) constant, even with different ROMs (say). BASIC 4 has
such a table at E000 in the 80-column version. All BASICs have the 'kernel', so that
JSR FFE4 always 'gets' a character, for example, even though FFE4 has different add-
resses following JMP. *Data tables* store ASCII data, arithmetic values, messages, and
so on. CBM BASIC for example has keywords stored in a table, and these are separ-
ated from each other by setting the high bit of the final letter of each keyword. The
high bit is masked (by AND #7F) before printing, but the N flag determines the end
of a keyword. A similar method is to store strings terminated by a zero byte; this
wastes one byte per string, but makes it easier to have strings of different lengths.
A routine to print such a string looks
something like this, assuming no string
is longer than 255 characters (so X on
its own spans the whole length of it).
Address tables are similar in purpose
to jump tables, but contain addresses
only, normally in 2-byte form with the
order reversed, without JMPs inter-
spersed. The beginning of CBM BASIC

```
      LDX #$00
LABEL LDA START,X ;START=START OF DATA TABLE
      BEQ EXIT
      JSR PRINT   ;USE STANDARD ROUTINE(S)
      INX         ;ASSUMES X IS PRESERVED
      BNE LABEL   ;BRANCH ALWAYS TAKEN
EXIT              ;CONTINUE PROCESSING
```

has several (very long) tables of this sort. Address tables in practice are tricky to
access. CBM BASIC has examples in which BASIC keywords' addresses are jumped to
by pushing two bytes on the stack and performing the routine to get the next BASIC
character, which ends RTS, and thus both loads the accumulator with the appropriate
value and jumps to any required address. (However, the addresses are actually stored
as address-1, because RTS always increments the return address). A similar process,
using RTI, is used by the machine-language monitor; this allows the status register to
be set, in addition to jumping to a specified address. These manoeuvres are necessary
because this part of BASIC is in ROM. From RAM, tables can be accessed by the
indirect jump instruction. This example
assumes that the addresses are tabled
starting at $B000, and that the X-reg-
ister contains the offset from the start

```
027A STX 027E  ;X ASSUMED TO BE OFFSET
027D JMP (B0xx);xx ALTERED BY X
```

of $B000 of the desired address. This method is far easier than using the stack, but
has the drawback that the indirect jump command has a bug, which causes it to work
wrongly if a page-boundary is crossed. See the appendices ('Further aspects of the
6502' on this). Also, the table mayn't begin at precisely $B000 or $C100 or whatever,
so X may need to be increased by some value.

11.2.15 Random numbers. There are several ways to find pseudo-random numbers in
machine-code. In the first place, the BASIC routine may be used; this has its own
special storage location, and derives each number from the previous value using this,
unless the argument is zero (i.e. RND(0)). The result is in the range 0 - 1; it is

forced into this range by putting the exponent to #80. So a random integer in the range 0 - 256 can be generated by calling the routine, replacing its exponent by #88, and calling the routine to convert this to an integer. This, though thorough, is long; two alternatives follow:

(i) <u>Using the VIA counters.</u> This method, like RND(0), is strictly dependent on time for the value it yields, so if it used within any regularly repeating code will show regularities. Nevertheless it is easy and quick to program:

```
LDA $E844 ; TIMER 1 LOW
EOR $E845 ; TIMER 1 HIGH
EOR $E848 ; TIMER 2 LOW
EOR $E849 ; TIMER 2 HIGH
```

This leaves A with an 8-bit value, which for many purposes is 'random'.

(ii) <u>Pseudo-random number calculations.</u> We'll consider 1-byte and 2-byte random numbers only; the principles are the same for larger byte numbers. The usual formula for pseudo-random numbers is a recurrence relation:

$$x_{i+1} = a * x_i + c \quad (\text{mod } m).$$

If we set m=256 or 65536, the modulo condition, which means take the remainder after division by m, is automatically set by ignoring any overflow. A theorem of Gauss's says that the maximum period of repetition is obtained when a=4n + 1 and c is an odd number. That is, if we base our calculations on a single byte, a formula in which 'a' is a multiple of four, plus one, and c is odd, will generate values which repeat with a cycle of length 256. The simplest case is

$$x_{i+1} = 5 * x_i + 1 \quad (\text{mod } 256)$$

which is easily programmed:

```
LDA STORE  ;LOAD LAST RANDOM NUMBER
ASL A      ;DOUBLE IT (IGNORING OVERFLOW)
ASL A      ;QUADRUPLE (IGNORING OVERFLOW)
SEC        ;THIS WILL ADD 1
ADC STORE  ;FIVE TIMES NUMBER + 1
STA STORE  ;SAVE NEW RANDOM NUMBER
```

This gives 0,1,6,31,156,13,66,75,120,89,190,183,148,229,122,99,240,177,118,79,140, ... which, because of the small values (5 & 1) may give an obvious sequence of ascending values when the seed becomes low (e.g. 2,11,56) but otherwise is probably satisfactory.* The process is similar with 16-bit numbers. We may use this easily-programmed relation: $x_{i+1} = 257*x_i + 43981$ (mod 65536). 43981 is #ABCD; other values are of course usable. If we represent the 16 bits by bytes HI and LO, this program generates a sequence of pseudo-random numbers recurring every 65536 repetitions:

CLC/ LDA LO/ ADC HI/ STA HI/ CLC/ LDA #CD/ ADC LO/ STA LO/ LDA #AB/ ADC HI/ STA HI

11.2.16 Addressing modes. To show how different addressing modes may be used to solve a programming problem, consider the question of memory-moving 1024 bytes from 6000 - 63FF into the screen area of a 40-column CBM, 8000 - 83FF. Four separate 'pages' of 256 bytes have to be moved. Section 9.5 has a solution to this problem, in which X counts from 4 to 0, while a loop involving indirect indexed addressing uses Y (changing from #0 through to #0 again) takes care of each page. It is possible to improve on that routine; it can be made both smaller and about 13% faster. Using its address locations, we could use routine (a):

```
     (a)                          (b)
L  0297  LDA (88),Y        L  0297  LDA 63FF,Y
   0299  STA (8A),Y           029A  STA 83FF,Y
   029B  INY                  029D  INY
   029C  BNE L                029E  BNE L
   029E  DEC 88               02A0  DEC 0299  ;DECREMENTS HIGH BYTE OF EACH
   02A0  DEC 8A               02A3  DEC 029C  ;ADDRESS IN LDA AND STA ABS,Y
   02A2  BPL L                02A6  BPL L
```

which uses no counter, but relies instead on the fact that #80, decremented to #7F, leaves the N-flag clear. Routine (b) is similar in its operation but modifies absolute addresses). It is faster, since the commands within the loop take less time. But it is

---

*Note: it is a peculiarity of this system of pseudo-random number generation that the values it produces are always alternately odd and even. In addition, particular series may have internal repeats or subseries of many kinds.

less easily relocated, since the program modifies itself as it runs, and it is essential to modify exactly the right addresses. Program (a) has the advantage of using the same zero-page locations to store its temporary data wherever the routine is put in memory. For the same reason, (b) cannot be used in ROM.

Self-modifying code is occasionally used by the CBM: the example (right) is part of the GETCHR routine, which BASIC uses when scanning through a program (See Chapter 14 on this topic). This avoids the need to reserve an index for use with indirect addressing; it also means the routine has to be present in RAM, not ROM.

```
0070   INC $77
0072   BNE $0076
0074   INC $78
0076   LDA xxxx   ;CURRENT ADDR.
0079              ;CONTINUE
```

11.2.17 Testing for the range of a byte of data.

A mysterious construction sometimes met with is illustrated (right) in a form which should make its operation fairly clear. Note that the first SEC is not necessary, since the carry flag is known to be clear at that point, so SEC/ SBC #30 could be replaced by SBC #2F . However, it draws attention to the fact that #30 and #D0 are 2's complements. The point is that, on exit, the contents of A are unchanged, but the carry flag is clear for values #30 - #39, and set for all other values. This range is of course the ASCII equivalent of numerals 0 - 9. As a variation on the theme, the second piece of code is an ASCII-to-hex routine; this replaces #30 - #39 by #00 - #09, and #41 - #46 (ASCII for A - F) by #0A - #0F. Note the temporary storage of the status register, later recovered to test for the status of C after the comparison.

```
       LDA xxxx
       CMP #3A   ;BRANCH IF
       BCS L     ; >= #3A
       SEC       ;#00 - #39
       SBC #30   ; →D0,D1,,,0,9
       SEC
       SBC #D0   ;0-9 CLEAR C
L                ;CONTINUE
```

```
       LDA xxxx
       CMP #3A
       PHP
       AND #0F   ;REMOVE HIBITS
       PLP
       BCC L
       ADC #08   ;ADDS 9,AS C=1
L                ;CONTINUE
```

11.2.18 Using subroutines.

JSR and RTS are complementary instructions, designed to be easy to use and trouble-free; usually they perform perfectly well, the exceptions being caused by failure to appreciate the workings of the stack. See PHA, and 11.2.14, for details on loading the stack with a new address for RTS, and for RTI; and see PLA on 'popping' return addresses from the stack, in the next chapter. Note that JSR xxxx/ RTS can always be replaced by JMP xxxx, saving two bytes from being pushed onto the stack. Conversely, however, JMP can be replaced by JSR / RTS only if it includes RTS at some stage.

Neither JSR nor RTS alters A,X, or Y, or any of the flags. This means that flags, set within a subroutine, can be tested on return. This is a valuable feature, which the specimen program (right) demonstrates. The subroutine's function is to examine the RAM area from $0100 - $010B for the presence of the character in A on entry. For example, to test for #45 in the buffer (ASCII for E),

```
7F5E   LDX #00
7F60   CMP 0100,X
7F63   BEQ 7F6B
7F65   INX
7F66   CPX #0C
7F68   BNE 7F60
7F6A   CLC
7F6B   RTS
```

```
       LDA #45
       JSR 7F5E
       BCS FOUND
```

can be used. BCS appears because, on return, the carry flag is clear if the character wasn't found; it is set if the character *was* found, and moreover X holds its position in relation to $0100. The INX construction has been used because the subroutine is intended to search the buffer from left to right; in fact, this buffer holds numerals as they are converted into strings for output to the screen or the printer. PRINT USING (Chapter 5) uses this subroutine.

Although subroutines can save a great deal of memory space and make programs more readable, they have one (admittedly small) drawback in the 6502, which is that they use only absolute addressing; relocating them is therefore tedious. Chapter 14 has remarks and methods to help get round this difficulty.

CHAPTER 12: ALPHABETIC REFERENCE TO 6502 OPCODES

This chapter lists each opcode with full details on its use. Short demonstration programs are provided for most opcodes.

The following conventions have been adopted:

:=   is read 'becomes'. For example, A:=X means that the value in A becomes that in X.

x,0 and 1   show the effect on the status flags of an opcode. x means that the flag is set, but its value may be 0 or 1. 0 and 1 represent flags which an opcode always sets to 0 and 1 respectively. All other flags are left unchanged.

$ and %   prefix hexadecimal and binary numbers; where these are omitted, a number is decimal.

A,X, and Y   are the accumulator and the two index registers X and Y.

M   means memory; this may be ROM in the case of load instructions. Note that # (immediate mode) loads from memory immediately following the opcode. All other addressing modes load from elsewhere in memory.

PSR   or SR is the processor status register.

SP   is the stack pointer.

PC   is the program counter; this is composed of two 8-bit registers, PCL and PCH.

The table below is intended as a reminder or summary of opcode mnemonics on the 6502, for readers who are not yet familiar with the opcodes or who have forgotten what they mean.

GUIDE TO 6502 OPCODE MNEMONICS

A	accumulator/ arithmetic shift	L	left/ logical shift
AD	add	LD	load accumulator, X, or Y
AND	logical AND	MI	minus
B	borrow the carry bit/ branch	NE	not equal to zero
BIT	bitwise instruction	NOP	no operation
BRK	break	OR	logical OR
C	carry bit/ flag is clear	P	processor status register
CL	clear flag	PH	push onto stack
CMP	compare accumulator	PL	pull from stack
CP	compare X or Y register	R	right
D	decimal flag	RO	rotate byte
DE	decrement X or Y register	RT	return
DEC	decrement memory	S	flag set/ shift/ stack pointer/ subroutine/ subtract
E	exclusive OR		
EQ	equal to zero	SE	set flag
I	interrupt flag	ST	store accumulator, X, or Y
IN	increment X or Y register	T	transfer between registers
INC	increment memory location	V	overflow flag
JMP	jump to new address	X	X register
JSR	jump to subroutine, saving return	Y	Y register

ADC

Add memory plus carry to the accumulator. $A := A + M + C$

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$61 (97 %0110 0001)	ADC (zero page,X)	oo	6
$65 (101 %0110 0101)	ADC zero page	oo	3
$69 (105 %0110 1001)	ADC # immediate	oo	2
$6D (109 %0110 1101)	ADC absolute	ooo	4
$71 (113 %0111 0001)	ADC (zero page),Y	oo	5*
$75 (117 %0111 0101)	ADC zero page,X	oo	4
$79 (121 %0111 1001)	ADC absolute,Y	ooo	4*
$7D (125 %0111 1101)	ADC absolute,X	ooo	4*

*Add 1 if page boundary crossed

Flags:

N	V	-	B	D	I	Z	C
x	x					x	x

Operation: Adds together the current contents of the accumulator, the byte referenced by the opcode, and the carry bit. If the result is too large for a single byte, C is set to 1. The internal overflow flag, V, is set if there is overflow from bit 6 into the high bit, bit 7. If A holds zero (i.e. each bit = 0) the Z flag is set to 1; otherwise it is 0.*If bit 7 in A is 1, the N flag is also set 1, to denote a 'negative' value in A.

Uses: [1] Single, double and multiple byte additions. The carry bit automatically provides for overflow from one byte to the next. For example:

```
CLC           ; ENSURES CARRY BIT IS 0
LDA $4A       ; WE WISH TO ADD #$0A (10 DECIMAL) TO THE CONTENTS
ADC #$0A      ; OF ($4A), I.E. THE DOUBLE-BYTE ADDRESS WHERE $4A
STA $4A       ; IS THE LOW BYTE AND $4B THE HIGH BYTE.
LDA $4B
ADC #$00      ; ADDS THE CARRY BIT WHERE APPLICABLE
STA $4B       ; RESULT MUST BE STORED, ELSE IT WILL REMAIN ONLY IN A.
```

[2] Increasing or decreasing the accumulator; there is no 'INC A' opcode.

```
BCS AWAY      ; EXAMPLE ONLY. (WE KNOW CARRY BIT IS CLEAR NOW).
ADC #$01      ; INCREMENTS A; #0 BECOMES #1, #1 #2,..., #FF BECOMES #0.

CLC           ; ANOTHER EXAMPLE. CARRY BIT NOW 0.
ADC #$FF      ; THIS SUBTRACTS 1 FROM A AND SETS CARRY FLAG UNLESS A=0
```

[3] In binary-coded decimal mode, obtained by setting D to 1, each nybble represents 0-9 and addition is corrected for this basis. This mode is unused in CBM equipment at present. On switching on and on entry to the monitor, D is always cleared, so ADC is in hexadecimal mode unless D is specifically set. This example adds 123 (decimal) to the contents of locations 0 and 1, which are assumed to contain, in ascending order, 4 binary coded digits. Thus locations 0 and 1 contain, in BCD, 0-9999.

```
SED           ; SET THE DECIMAL FLAG
CLC           ; CLEAR CARRY FLAG
LDA $01       ; WE'VE ASSUMED THE BCD DATA IS STORED IN NORMAL ORDER,
ADC #$23      ;   WITH LOW BYTES FOLLOWING HIGHER ONES, NOT 6502 ORDER
STA $01       ; ADD 23 DECIMAL
LDA $00
ADC #$01      ; ADD 01 DECIMAL PLUS POSSIBLY CARRY BIT EQUIVALENT TO 100
STA $00
CLD           ; CLEAR THE DECIMAL BIT, UNLESS MORE DECIMAL MATH NEEDED
```

Notes: [1]*In decimal mode, the zero flag doesn't operate normally with ADC because of the automatic correction (adding 6) which the 6502 carries out. Testing for a zero result requires (for example) TAX/ BEQ ... or CMP #00/ BEQ ... which is an extra step not required in hexadecimal arithmetic.

[2] The V flag is important if the 2's complement convention is in use, in which case it tests whether the high bit means negative, or implies that the addition has overflowed to bit 7. In BCD, 2's complement is unusable.

AND

Logical AND of memory with the accumulator.   A := A AND M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$21 (33 %0010 0001)	AND (zero page,X)	oo	6
$25 (37 %0010 0101)	AND zero page	oo	3
$29 (41 %0010 1001)	AND # immediate	oo	2
$2D (45 %0010 1101)	AND absolute	ooo	4
$31 (49 %0011 0001)	AND (zero page),Y	oo	5
$35 (53 %0011 0101)	AND zero page,X	oo	4
$39 (57 %0011 1001)	AND absolute,Y	ooo	4*
$3D (61 %0011 1101)	AND absolute,X	ooo	4*

*Add 1 if page boundary crossed

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: Performs AND of the 8 bits currently in the accumulator and the 8 bits referenced by the opcode. See BASIC's AND command for a description of the truth table of AND. Essentially, when both bits are 1, the result is 1, but if either or both bits are zero, the result is 0. (So the first bit AND the second must be 1). The resulting byte is stored in A. If A now holds 0, i.e. all its bits are zero, the Z flag is set to 1; and if the high bit is set, i.e. bit 7 is 1, the 'negative' flag N is set to 1. Otherwise the flag is 0.

Uses: [1] 'Masking' off unwanted bits (cp. ORA) typically to test for the existence of a few high bits, or to test that some bits are zero:

```
LDA $E081,X ; LOADS ACCUMULATOR FROM A TABLE OF CODED VALUES..
AND #$3F    ; .. TURNS OFF BITS 6 AND 7, LEAVING ALPHABETIC ASCII.

LDA $E840   ; LOADS ACCUMULATOR FROM VIA'S IEEE REGISTER + CASSETTE
AND #$EF    ;  MOTOR CONTROL. THEN TURNS OFF BIT 4 WITH %1110 1111.
STA $E840   ; STORES RESULTING VALUE BACK IN REGISTER
```

[2]   AND #$FF RESETS FLAGS AS THOUGH LDA HAD JUST OCCURRED; AND #$00 HAS THE SAME EFFECT AS LDA #$00

ASL

Shift memory or accumulator left one bit. |C|◄|7 6 5 4 3 2 1 0|◄|0|

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$06 (6 %0000 0110)	ASL zero page	oo	5
$0A (10 %0000 1010)	ASL accumulator	o	2
$0E (14 %0000 1110)	ASL absolute	ooo	6
$16 (22 %0001 0110)	ASL zero page,X	oo	6
$1E (30 %0001 1110)	ASL absolute,X	ooo	7

Flags:

N	V	-	B	D	I	Z	C
x						x	x

Operation: Moves the contents of memory or the accumulator left by one bit position, moving 0 into the low bit, and the high bit into the carry flag, erasing its current value. The carry bit therefore is set to 0 or 1 depending on bit 7 previously being 0 or 1. Z and N are set according to the result; thus Z can be true (1) only if the location or A held #$00 or #$80 before ASL. The N bit can only be set true if bit 6 was previously 1.

Uses: [1] Doubles a byte (though not in decimal mode). If signed arithmetic is not being used the result can safely reach values not exceeding 254, after which the carry must be taken into account, often with ROL. This example uses A from 0 to 127 to load *two* bytes from a table of address pointers: ASL A/ TAY/ LDA ADDHI,Y/ PHA/ LDA ADDLO,Y/ PHA and store them on the stack. Another example: LDA $20/ ASL A/ ADC $20 multiplies the contents of $20 by three, *provided that* it originally held #85 decimal at most. In this case, the carry bit is automatically cleared by the shift.

[2] Tests a bit by moving it into Z or N, perhaps with a flag in BASIC. Note that 4 ASLs move the low nybble into the high nybble.

BCC

Branch if the carry bit is 0.   PC:=PC + offset

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$90 (144 %1001 0000)	BCC relative	oo	2*

*Add 1 if branch occurs; add 1 more if the branch crosses a page

Flags: N V - B D I Z C

Operation: If C holds 0, the byte following the opcode is added to the program counter, which is set to the following command. If C holds 1, the program counter is unaffected. The effect is to cause a jump to the offset address when C is clear.

Uses: [1] If the carry bit is known to be clear, this command becomes effectively a 'branch always' instruction. So the flag may be set in a purely signalling sense, with no significance other than to show that one of two conditions applies. For example, BASIC's commands to END and STOP a program are almost identical, except that one command prints a message with a line-number, and the other doesn't. So:

```
C763 BCC C768 ; CARRY BIT IS SET FOR STOP, CLEAR FOR END...
C765 JMP C37E ; PRINT "BREAK IN LINENUMBER" MESSAGE
C768 JMP C389 ; PRINT "READY"
```

[2] Usually the test is concerned with the result of a previous operation which may or may not set the carry flag; this compare routine for instance:

```
JSR GETCHAR ; LOAD THE ACCUMULATOR WITH A VALUE (DEVICE NO.,TRACK,SEC-
CMP #$0A     ;   TOR, OR WHATEVER).  COMPARE WITH #0A; THIS SETS C.
BCC TO 0-9   ; BRANCH TO PROCESS THESE LOW VALUES;
             ; CONTINUE HERE WITH HIGH VALUES, 10-255 DECIMAL.
```

ADC and SBC, the add and subtract operations, are obvious candidates here, but occur less often with BCC than might be expected, because the value of the carry bit can often be taken care of by the mathematical routine without the need for branching. This example might be used in 2-byte addition where an overflow warning is needed:

```
CLC          ; CLEAR FOR ADD
LDA LOADD    ; ACCUMULATOR HOLDS CONTENTS OF THE LOWER BYTE..
ADC #LO      ; .. ADDS THE LOWER BYTE VALUE
STA LOADD    ; AND STORES IT; C MAY BE 0 OR 1.
LDA HIADD    ; ACCUMULATOR HOLDS MORE SIGNIFICANT ADDRESS BYTE..
ADC #HI      ; .. ADDS THE HIGHER BYTE VALUE
STA HIADD    ; AND STORES IT. AGAIN, C MAY BE 0 OR 1:
BCC CONT     ; IF IT'S CLEAR, THE ADDITION HASN'T OVERFLOWED.
JMP OVERFL   ; IF IT'S 1, PROCESS ACCORDINGLY; E.G. ERROR INDICATION
```

BCS

Branch if the carry bit is 1.   PC:=PC +offset if C=1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$B0 (176 %1011 0000)	BCS relative	oo	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page

Flags: N V - B D I Z C

Operation: Identical to BCC, except that the branch is taken if C=1 and not C=0.

Uses: [1] The uses are identical to those of BCC; the choice between BCC and BCS at a branch point depends on convenience only. For example, suppose a hardware port is to be read until bit 1 is set to 0; this routine:

```
LOOP LDA PORT    ; LOAD FROM HERE UNTIL xxxx xx0x
     LSR A
     LSR A
     BCS LOOP
```

is more natural than one involving BCC, and easier to read.

BEQ

Branch if zero flag is 1.   PC := PC + offset if Z = 1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$F0 (240 %1111 0000)	BEQ relative	oo	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page

Flags: | N V - B D I Z C |
|---|

Operation: If Z=1, the byte following the opcode is added, in 2's complement arithmetic, to the program counter, which currently points to the next opcode. The effect is to cause a jump, forward or backward, up to a maximum of about ±128, if the zero flag is set. IF Z=0 the branch is ignored.

Uses: [1] The zero flag cannot be set directly (there is no 'SEZ'). But since it is very often set in the course of a program, the use of BEQ as an unconditional branch is common. This sort of thing may be used to make short routines relocatable, where the branch command isn't quite wide-ranging enough to permit all the branching that is needed within the routine to take place without an intermediate hop. This, of course, is not really recommended with large programs.

```
LDA #$F5; SOME VALUE OR OTHER
BEQ BACK; THESE TWO BRANCHES
BEQ FWRD  RELY ON Z=1
```

[2] A common use is to end a loop, either when a counter is decremented to zero, or because a zero byte is deliberately used as a terminator:
```
LOOP LDA TABLE,X ; LOAD A WITH THE NEXT CHARACTER
     BEQ EXIT    ; EXIT LOOP WHEN ZERO BYTE FOUND
     ... CONTINUE, E.G. STA OUTPUT,X/ INX/ BNE LOOP
```

[3] BEQ is popular after comparisons because it's easy to use:
```
JSR GETCHR/ CMP #$2C/ BEQ COMMA looks for a comma in BASIC.
```

Notes: [1] When a result is *zero,* the zero flag Z is made 'true'- i.e. 1. This point can confuse people. 'BEQ' is usually read 'branch if equal to zero' but when comparisons are being made it could be read 'branch if equal'.

BIT

Test memory bits.   Z flag set on A AND M; N flag=M7; V flag=M6

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$24 (36 %0010 0100)	BIT zero page	oo	3
$2C (44 %0010 1100)	BIT absolute	ooo	4

Flags: | N V - B D I Z C |
|---|
| M7 M6   x |

Operation: BIT affects *only three flags* leaving registers and data unchanged. Z is set on A AND M: if no bit of the memory location and of A has a 1 in each, then A AND M is zero and Z=1. Also, bits 6 and 7 are copied from memory to V & N.

Uses: [1] The 3-byte, absolute address BIT is the only instruction regularly used
```
LDA #$0D   A9 0D   ; LDA #0D
BIT $20A9  2C A9 20 ; LDA #20
BIT $1DA9  2C A9 1D ; LDA #1D
```
for unconventional disassembly from a non-standard entry point. The example loads A with Return, space, or [RIGHT] depending on the entry point into the routine.

[2] BIT with BMI/BPL or BVC/BVS tests bits 7 and 6. This is often used
```
    BIT $07
    BMI ERR
    RTS
ERR JMP ERROR
```
with PIA/VIA locations- see the IEEE examples. The example here tests location $07, with an error indication if it holds a negative. $07 is in fact used to check for type mismatches. #FF denotes a string, #00 a numeric variable.

[3] This example shows the AND feature in use. CHRFLG holds #0 if no char-
```
LDA VALUE
BIT CHRFLG
BEQ NOTOUT
```
acter is to be output, and #FF otherwise. Assuming the accumulator holds a non-zero value, BIT tests whether to branch past the output routine, while retaining A's value.

BMI

Branch if the N flag is 1. PC:=PC + offset if N=1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$30 (48 %0011 0000)	BMI relative	oo	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page

Flags: | N V - B D I Z C |
       | |

Operation: The program counter is incremented to point at the next opcode, and if N holds 1 the byte following the branch opcode is added to the program counter in 2's complement form. The effect is to force a jump to the new address. The maximum range of a branch is therefore about ±128. When N holds 0 the branch command is ignored.

Uses: [1] Testing the 'negative' bit of a location; for example:
```
LOOP BIT $E840; BIT 7 IS CONNECTED TO DATA VALID SIGNAL..
     BMI LOOP ; .. THIS LOOP WAITS UNTIL DAV IS 0.
```

Like other flags, N may be used in a purely conventional sense. As an example, consider BASIC's keyword tokens: these all have values, in decimal, of 128 or more, which keeps keywords logically separate from other BASIC, and also permits instructions like this schematic branch:
```
LDA NEXT  ; LOAD NEXT CHR INTO ACCUMULATOR
BMI TOKEN ; BRANCH TO PROCESS A KEYWORD
          ; OTHERWISE, PROCESS DATA AND EXPRESSIONS
```

Notes: [1] It's important to realise that the 'minus' in BMI refers only to the use of bit 7 to denote a negative number in 2's complement arithmetic. It may be easier to think of this operation as 'branch if the high bit is set'. BPL is exactly the opposite of BMI. Where one branches, the other does not.

BNE

Branch if Z is 0. PC:=PC + offset if Z = 0

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$D0 (208 %1101 0000)	BNE relative	oo	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page

Flags: | N V - B D I Z C |
       | |

Operation: BNE operates exactly like BEQ, except that the condition is opposite. If Z=0 the offset contained in the byte after BNE is added to the program counter, so the branch takes place. If Z=1 the branch is ignored.

Uses: [1] BNE may be used in unconditional branches in circumstances like those which apply to BEQ- see note [1] on BEQ.

[2] BNE is very often used in a loop in which a counter is being decremented. This is probably the easiest type of loop to write, although the starting address of the loop's data needs to be fixed with care, as offset 0 isn't executed by a loop like this. The example prints ten characters from a table, their offsets being 10,9,8, ...,2,1.
```
    LDX #$0A
LOOP LDA TABLE,X
    JSR OUTPUT
    DEX
    BNE LOOP
```

[3] BNE, like BEQ, is popular after comparisons:
```
B4C0 LDA $C1
B4C2 CMP #$42; IS IT B?      Comparisons like this can continue over
B4C4 BNE $B4C9               many bytes of machine-code.
B4C6 JMP $B876
B4C9 CMP #$48; IS IT H?
```

Notes: [1] When a result (say, of LDA) is non-zero, the zero flag Z is made false, i.e. set to 0. This can be confusing. 'BNE' is usually read 'branch if not equal to zero'. The result of a comparison is zero if both bytes are identical, because one is subtracted from the other. Hence the use of BNE and BEQ.

BPL

Branch if the N flag is 0.   PC:=PC + offset if N = 0

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$10 (16 %0001 0000)	BPL relative	oo	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page

Flags: | N V - B D I Z C |
|---|
| |

Operation: BPL operates exactly like BMl, except that the condition is opposite. The branch is taken to the new address given by program counter plus offset *if* N=0. This means that if the result was positive *or zero* the branch is taken.

Uses: [1] In testing the negative bit of a memory location. This code, for example waits until the accumulator holds a byte with bit 7 on. Such a location must be hardware cotrolled, not just RAM.

```
LOOP LDA TESTLOCN
     BPL LOOP
```

[2] Testing for the end of a loop where a counter is being decremented, and the counter's value 0 is needed. This simple loop prints 10 bytes to screen:

```
     LDX #$09   ; X REGISTER WILL COUNT 9,8,7, ... ,1,0
LOOP LDA BASE,X ; 'BASE' IS THE STARTING ADDRESS OF THE 10 BYTES
     STA 8000,X ; CBM SCREEN STARTS AT $8000. (OTHER MACHINES DIFFER).
     DEX        ; DECREMENT X
     BPL LOOP   ; BRANCH WHEN POSITIVE OR ZERO
```

BRK

Force break.   S:=PCH, SP:=SP-1, S:=PCL, SP:=SP-1, S:=PSR, SP:=SP-1, PC:=(FFFE)

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$00 (0 %0000 0000)	BRK implied	o	7

Flags: | N V - B D I Z C |
|---|
| 1   x |

Operation: BRK is a forced interrupt, which saves its current position and status and jumps to a standard address. Note that (i) The program counter saved points to the BRK byte plus two (like a branch), and (ii) The processor status register on the stack has flag B set to 1. In CBM machines, the new address is shared by the IRQ service routine and generally with the 6502 maskable interrupts always jump to ($FFFE); the break flag is a sort of designer's patch so that BRK can be recognized as different from other interrupts.

Uses: [1] BRK *can* be used to patch programs (as mentioned in Zaks' 6502 book), but this requires (i) a change in the interrupt's vector, i.e. ($92) in BASIC>1 or ($0219) in BASIC 1 and (ii) processing required by the program. Also a return, using the stack program counter and processor status, is needed. This is sufficiently complex not to be done often.

[2] With BASIC vectors left as on power-up, BRK causes BASIC to jump to the monitor, or to location $0 in BASIC 1. This is why SYS 1024 or SYS 4 may be used to enter the monitor. 1024 normally contains a zero byte at the start of BASIC; 4 is zero whenever quotes mode is not set, e.g. when in direct mode. On entering the monitor the data on the stack is pulled off and displayed. The program counter is (for some reason) decremented; since PC+2 is stored, the monitor address points at the byte after BRK.

BVC

Branch if the internal overflow flag (V) is 0.   PC:=PC + offset if V=0

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$50 (80 %0101 0000)	BVC relative	oo	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page

Flags: | N V - B D I Z C |

Operation: If V is clear, the byte following the opcode is added, as a two's complement number, to the program counter, set to point at the following instruction. The effect is to jump to a new address, which must be within a range of about ±128 bytes. If V=1 the next instruction is processed and the branch ignored.

Uses: [1] As a 'branch always' instruction:
```
          CLV
          BVC LOAD
```

[2] With signed arithmetic, to detect overflow from bit 6 into bit 7, giving a spurious negative bit. This is rather rarely used since the sign of a number can be held apart from the number, perhaps as #0 or #FF, so that ordinary arithmetic can be used without the extra complication of the V bit. See the note for an explanation of V's derivation.

```
          LDA ADD1      This routine adds two numbers, in 2's complement form;
          CLC           their range therefore is -128 to 127. Clearing V is only
          (CLV)         used in examples like [1]; unlike the carry bit C, it is
          ADC ADD2      never added in to results, so clearing is not needed.
          BVC OK        CLC is necessary; it may add in 1 to the result.
          JMP OVERFL
```

[3] BIT copies a location's 6th bit into the processor status register, so BVC or BVS can be used to test bit 6. For example, this routine:

```
          F103 BIT $E840      waits until the hardware sets bit 6 of location $E840
          F106 BVC $F103      equal to 1.
```

Notes: [1] The Meaning of V. When using signed arithmetic, two numbers of opposite sign cannot overflow. The most extreme values, e.g. -128+0, are always within the acceptable range. However, if the signs are the same, overflow is possible. #$6A + #$5B overflows: the result is not in the range -128 to 127. We have: %0110 1010 + %0101 1011 = %1100 0101. When the two leftmost bits are 0, each original number is positive; if in addition the result has bit 7 equal to 1, an overflow must have occurred. Similarly, two negative numbers with overflow behave like this: consider -100 and -89, in decimal. When added, these overflow, since -189 is out of the acceptable range for signed bytes. Now, +100 = #$64 and +89 = #$59. The negatives are therefore -100 = #$9C and -89 = #$A7. We have: %1001 1100 + %1010 0111 = %0100 0011. The overflow shows itself in causing two negative numbers to apparently add to a positive. V is thus calculated by complementing the EOR of 2 negativebits, and ANDing the result with the EOR of the result's negative bit and ont the original number's negative bits. This is not so complicated as it might seem.

BVS

Branch if the internal overflow flag (V) is 1.   PC:=PC + offset if V= 1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$70 (112 %0111 0000)	BVS relative	oo	2*

*Add 1 if branch occurs; add 1 more if branch crosses a page

Flags: | N V - B D I Z C |

Operation: This branch is identical to BVC except that the test logic to decide whether the branch is taken is opposite. See notes on BVC.

CLC

Clear the carry flag.   C:= 0

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$18 (24 %0001 1000)	CLC implied	o	2

Flags:

N	V	-	B	D	I	Z	C
							0

Operation: The carry flag is set 0. All other flags are unchanged.

Uses: The carry bit is an automatically included feature in add and subtract commands (ADC and SBC), so that accurate calculations require the flag to be in a known state. CLC is the usual preliminary to additions:

```
        CLC
        LDA #$02      After CLC, this routine adds 2 and 2 and prints the
        ADC #$02      resulting byte, which is 4. In multiple-byte additions
        JSR PRBYTE    C is cleared at the start but subsequently used to
carry through the overflows, if they exist.
```

CLD

Clear the decimal flag.   D:= 0

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$D8 (216 %1101 1000)	CLD implied	o	2

Flags:

N	V	-	B	D	I	Z	C
				0			

Operation: The decimal flag is set 0; all other flags are unchanged.

Uses: Resets the mode for ADC and SBC so that hexadecimal (binary) arithmetic is performed, not binary-coded decimal. Typically, SED precedes some decimal calculation, with CLD following when this is finished.

Notes: Commodore BASIC uses no decimal mode calculations; on switching the machine on, CLD is executed and the flag is permanently left off. Entry to the monitor clears D, should it happen to have been set. There have been reports that future BASICs may contain BCD arithmetic.

CLI

Clear the interrupt disable flag.   I:= 0

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$58 (88 %0101 1000)	CLI implied	o	2

Flags:

N	V	-	B	D	I	Z	C
					0		

Operation: The interrupt disable flag is set to 0. From now on, interrupts will be processed by the system, using the IRQ vector in ($FFFE).

Notes: [1] Interrupts through the NMI line ('non-maskable interrupts') take place irrespective of the I flag.

[2] Commodore use the interrupt to process the keyboard and clocks. Typically, CLI is used after SEI plus changes to interrupt vectors. Often, CLI isn't needed when used with BASIC, as a number of BASIC routines themselves use CLI. See SEI for an example including CLI .

CLV

Clear the internal overflow flag.   V:= 0

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$B8 (184 %1011 1000)	CLV implied	o	2

Flags:

N	V	-	B	D	I	Z	C
	0						

CMP

Compare memory with the contents of the accumulator.   PSR set by A - M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$C1 (193 %1100 0001)	CMP (zero page,X)	2	6
$C5 (197 %1100 0101)	CMP zero page	2	3
$C9 (201 %1100 1001)	CMP # immediate	2	2
$CD (205 %1100 1101)	CMP absolute	3	4
$D1 (209 %1101 0001)	CMP (zero page),Y	2	5*
$D5 (213 %1101 0101)	CMP zero page,X	2	4
$D9 (217 %1101 1001)	CMP absolute,Y	3	4*
$DD (221 %1101 1101)	CMP absolute,X	3	4*

*Add 1 if page is crossed

Flags:

N	V	-	B	D	I	Z	C
x						x	x

Operation: CMP affects *three flags only,* leaving registers and data intact. The accumulator is *not* changed. The byte at the address specified by the opcode is subtracted from A, and the three flags N, Z, and C set depending on the result. Thus, if the accumulator holds the same value as the memory location, the result is zero, and BEQ causes the appropriate action to be taken. Before performing the subtraction, the carry bit is set by the chip. Within the chip, what happens is that the accumulator's contents are added to the 2's complement of the data. and the result of this determines how the flags are set.

Uses: [1] With the zero flag, Z. This is the easiest flag to use with CMP*:-

```
FF22 JSR FFCF;  INPUT A CHARACTER
FF25 CMP #$20;  IS IT A SPACE?
FF27 BEQ FF22;  YES. INPUT AGAIN
FF29 CMP #$0D;  IS IT C.RETURN?
FF2B BEQ FF47;  YES. BRANCH ...
FF2D CMP #$22;  ..NO. MAYBE "?
```

This is part of a routine to parse BASIC lines from the keyboard; the characters it looks for are typical of such routines.

[2] With the carry flag, C. This is quite straightforward. If the memory contents are less than A, or equal to A, the carry flag is set. 'Less than' means in the absolute sense, not the 2's complement sense. Thus, 100 is less than 190, although in 2's complement notation, 190, being negative, would count as the smaller number of the two.

```
...
LDA 085D
CMP #$3A
BCS 0087
CMP #$20
...
```

This is part of CHRGET, where A is loaded with one byte of the BASIC program in memory. It's then processed; see Chapter 14 for details. This extract compares with the ASCII for a colon, which is #3A, and branches for any less value, or if equal, to RTS.

```
LDY #$00
LDA (PTR),Y
CMP #$20
BCC B1
CMP #$40
BCC B2
...
```

This example shows how a range of values may be tested for and processed. Starting with the lowest ranges, comparisons are carried out until the correct range is found; each comparison is followed by a branch to B1, B2 etc. where processing is carried out for 0-#1F, #$20- #$39, and so on.

[3] With the negative flag, N. This is the most obscure flag to use with CMP. The reason is that 2's complement numbers are assumed, and if you are working with these CMP operates as expected, subtracting the memory from the accumulator, and therefore giving a negative answer whenever the memory exceeds the accumulator. If both numbers are positive, or both negative, the N flag is set as though absolute subtraction were being used, and in these circumstances BMI/BPL can be used. But if the two data items have different signs, the comparison process is complicated by the fact that the V bit may register internal overflow. See Chapter 11 for more detail.

---

*We have it on the authority of Gerry Weinberg that poor quality machine-code invariably has a branch of this type; you have been warned!

CPX

Compare memory with the contents of the X register.   PSR set by X - M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$E0 (224 %1110 0000)	CPX # immediate	oo	2
$E4 (228 %1110 0100)	CPX zero page	oo	3
$EC (236 %1110 1100)	CPX absolute	ooo	4

Flags:

N	V	-	B	D	I	Z	C
x						x	x

Operation: CPX affects *three flags only,* leaving the registers and data intact. The byte referenced by the opcode is subtracted from X, and the flags N,Z, and C set depending on the result. Within the chip, X is added to the 2's complement of the data, and the result of this determines how the flags are set.

Uses: [1] With the zero flag, Z. This flag tests equality.

```
        LDX #$00
LOOP LDA $0270,X
        STA $026F,X
        INX
        CPX $9E
        BNE LOOP
```

The loop in this example is part of the keyboard buffer processing, showing how the contents of the buffer are shifted one character at a time. $9E is a zero-page location, updated whenever a new character is keyed in, which holds the current number of characters in the buffer: the comparison provides a test to end the loop.

[2] With the carry flag, C. This flag tests for X>=M and X<M:-

```
        LDX $00
        CPX #$50
        BCS EXIT; IF X>79
        ...
```

The test routine is part of a graphics plot program; location $00 holds the horizontal coordinate, which is to be in the range 0-79 to fit the screen. The comparison causes exit, without plotting, when X holds 80-255.

[3] With the negative flag, N. When X and the data have the same sign, i.e. both are 0-127 or 128-255, BMI has the same effect as BCC, and vice versa. When the signs are opposite, the process is complicated by the possibility of overflow into bit 7. For example, 78 compared with 225 sets N=0, but 127 compared with 255 sets N=1. (Because 225=-31 as a 2's complement number; thus 78+31=109 with N=0, but 127+31=158 with N=1)

CPY

Compare memory with the contents of the Y register.   PSR set by Y - M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$C0 (192 %1100 0000)	CPY # immediate	oo	2
$C4 (196 %1100 0100)	CPY zero page	oo	3
$CC (204 %1100 1100)	CPY absolute	ooo	4

Flags:

N	V	-	B	D	I	Z	C
x						x	x

Operation: CPY affects *three flags only,* leaving the registers and data intact. The byte referenced by the opcode is subtracted from Y, and the flags N, Z, and C set depending on the result. Apart from the use of Y in place of X, with the resulting asymmetry in the implementation of addressing, this opcode is identical in its effects to CPX.

Notes: The major difference in addressing between X and Y is the fact that post-indexing of indirect addresses is available only with Y. So this type of

```
        LDY #$00
LOOP LDA (PTR),Y
        JSR OUTPUT
        INY
        CPY LENGTH
        BNE LOOP
```

construction, in which a set of consecutive bytes, perhaps a string in RAM or an error message, is processed up to some known length, tends to use the Y register.

DEC

Decrement memory contents. M:=M-1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$C6 (198 %1100 0110)	DEC zero page	oo	5
$CE (206 %1100 1110)	DEC absolute	ooo	6
$D6 (214 %1101 0110)	DEC zero page,X	oo	6
$DE (222 %1101 1110)	DEC absolute,X	ooo	7

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The byte referenced by the addressing mode is decremented by 1, setting the N flag and the Z flag. If the byte contained anything from #81 to #0, after DEC the N flag will be set; however, Z will be 0 except for the one case where its value was #1 before the decrement. Probably #FF is added within the chip itself, setting N and Z on the result. Note that the carry bit is unchanged irrespective of the outcome of DEC.

Uses: [1]
```
     LDA $93
     BNE +2
     DEC $94
     DEC $93
```
This short routine shows an efficient method to decrement a zero page pointer or any other double-byte value. It uses the fact that the high byte must be decremented only if the low byte is exactly zero. Compare INC.

[2] Counters other than the X register and Y register can easily be implemented with this command (or INC). Such counters must be in RAM; there is no 'DEA' instruction. This simple delay loop which decrements locations $00 and $01 shows the type of thing:-

```
          AND #$00; FOR A CHANGE
          STA $00 ; SET THESE BOTH
          STA $01 ; TO ZERO
     LOOP DEC $00
          BNE LOOP; 255 LOOPS...
          DEC $01
          BNE LOOP; ... BY 255
```
A zero page decrement takes 5 clock cycles to carry out; a successful branch takes 3. (We'll assume a page isn't crossed, as in fact it is statistically unlikely to be if this code is put into RAM at random). The inside loop therefore takes 8*255 cycles to complete, and the whole loop is a little more than 8*255*255 cycles. We can divide this by a million to get the actual time in seconds, which is about half a second.

DEX

Decrement the contents of the X register. X:=X-1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$CA (202 %1100 1010)	DEX implied	o	2

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The contents of the X register are decremented by 1, setting the N flag if the result has bit 7 set, and setting the Z flag if the result is 0. As with DEC, the carry bit is unaltered.

Uses: DEX is almost exclusively used to count X in a loop. Its maximum range, of 255 bytes, is often insufficient, so several loops may be necessary.

```
E12F LDX #$1C
E131 LDA E0F8,X
E134 STA 6F,X
E136 DEX
E137 BNE E131
```
This routine moves 28 bytes from ROM to RAM, including the CHRGET routine, in BASIC 2.

```
          LDX #$00
     LOOP LDA #$20
          STA $8000,X
          STA $8100,X
          STA $8200,X
          STA $8300,X
          DEX
          BNE LOOP
```
This is a screen-clearing routine, which puts 1000 bytes of #$20 (space) into RAM. With an 80-column machine only the top half of the screen blanks out, because 80 columns by 25 rows gives 2000 locations.

DEY

Decrement the contents of the Y register. Y:=Y-1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$88 (136 %1000 1000)	DEY implied	o	2

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The contents of the Y register are decremented by 1, setting the N
flag if the result has bit 7 set (i.e. is greater than 127), and setting the Z
flag if the result is #0. As with DEC, the carry bit is unaltered.

Uses: DEY, like DEX, is almost exclusively used to count within loops. There are
more opcodes which have indexing by X than by Y, so X is more popular for
this purpose. This example is less of a loop than those I've chosen for X:-

```
        LDY #$02
        LDA (PTR),Y; LOAD 2ND BYTE   This inclusively ORs together three adjacent
        DEY                          bytes; so that if the result is #0, each of
        ORA (PTR),Y; ORA 1ST BYTE    the three must have been zeros. Note the
        DEY                          addressing mode, which is indirect indexed,
        ORA (PTR),Y; ORA 0TH BYTE    the indirect mode which is post-indexed by
        BNE CONT   ; END IF ZERO     the Y register.
```

EOR

Accumulator's contents are exclusively ORed bitwise with the contents of memory.
A := A EOR M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$41 (65 %0100 0001)	EOR (zero page,X)	oo	6
$45 (69 %0100 0101)	EOR zero page	oo	3
$49 (73 %0100 1001)	EOR # immediate	oo	2
$4D (77 %0100 1101)	EOR absolute	ooo	4
$51 (81 %0101 0001)	EOR (zero page),Y	oo	5*
$55 (85 %0101 0101)	EOR zero page,X	oo	4
$59 (89 %0101 1001)	EOR absolute,Y	ooo	4*
$5D (93 %0101 1101)	EOR absolute,X	ooo	4*

*Add 1 if page is crossed

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: An exclusive OR (compare ORA for a description of an inclusive OR)
is a logical operation in which bits are compared, and EOR is considered to
be 'true' if A or B - but not both, or neither - is true. For example,
let's evaluate #AB EOR #5F. Now #AB is %1010 1011, i.e. in decimal, ten
followed by eleven. #5F is %0101 1111. So the EOR of these two is
%1111 0100, or #F4. We arrive at this result by a process of bit comparisons,
where bit 7 is 0 EOR 1=1, and so on.

Uses: [1] CBM graphics use bit 7 to signal reverse video. In BASIC, this screen
POKE can be used to reverse any character(s): POKE P, (PEEK(P) OR 128)
AND NOT (PEEK(P) AND 128). This exclusively-ORs the high bit with the peek
value, reversing the video byte. In machine code, the same reverse effect

```
        LDA LOCN        is more elegantly and quickly achieved with this
        EOR #$80        EOR, which is not directly available in BASIC.
        STA LOCN        Chapter 9 has more on this topic.
```

[2] EOR is exceptional among CBM and 6502 logical functions in that no
'information' in the technical sense is lost. If you repeatedly AND, you will
finish with #0; if you ORA, you'll end with #FF. For this reason, hashtotals
and data encryption algorithms often use EOR. To code data, each byte is
EORed with a byte generated by some secret, repeatable process. When the
result is EORed again with the identical sequence, all the original bytes are
restored.

INC

Increment memory contents.   M:=M+1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$E6 (230 %1110 0110)	INC zero page	oo	5
$EE (238 %1110 1110)	INC absolute	ooo	6
$F6 (246 %1111 0110)	INC zero page,X	oo	6
$FE (254 %1111 1110)	INC absolute,X	ooo	7

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The byte referenced by the addressing mode is incremented by 1, setting the N flag and the Z flag. The N flag will be 1 if the contents' high bit is 1, and otherwise 0; and Z will be 1 if the contents now equal zero exactly. The carry bit is unchanged.

Uses: [1]   INC $93   This short routine shows an efficient method to increment a
           BNE CONT   zero page pointer or any other double-byte value. The high
           INC $94   byte must be incremented only when the low byte changes
       CONT ...   from #FF to #00. Compare DEC.

[2] Exactly as note [2] in DEC, INC may be used to implement counters in RAM where the X and Y registers are insufficient. Suppose we use the IRQ interrupt servicing to (say) flash a cursor or repeat a key. Something like:

IRQ   INC $00
      BEQ +3   where IRQCONT is the interrupt's usual routine enables
      JMP IRQCONT   some periodic routine to be performed. Here, the zero
      LDA #20   page location $0 is used to count from #20 up to #FF and
      STA $00   #00, so the processing occurs every 255-32=223 jiffies -
      ...   about every 3.7 seconds.

Notes: [1] The accumulator can't be incremented using this; CLC/ ADC #01 or SEC/ ADC #00 must be used, or TAX/ INX/ TXA or some other variation.

[2] Remember that INC doesn't load; if the incremented contents are to be used in A, or in a register, then INC $C6 say must be followed by LDA $C6 or LDX $C6 or LDY $C6.

INX

Increment the contents of the X register.   X:=X + 1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$E8 (232 %1110 1000)	INX implied	o	2

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The contents of the X register are incremented by 1, setting the N flag if the result has bit 7 set, and the Z flag if the result is zero. These flags may both be 0, or one of them may be 1; it is impossible for both to be set 1 by this command. The carry bit is unchanged.

Uses: INX is common as a loop variable. It is also often used to set miscellaneous values which happen to be near each other, like this:

      LDX #$00
      STX $033A
      STX $033C
      INX
      STX $10

Stack pointer processing tends to be connected with the use of the X register, because TXS and TSX are the only ways of accessing SP. Most of CBM's stack handling and memory checking for BASIC uses the X register.

INY

Increment the contents of the Y register. Y := Y + 1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$C8 (200 %1100 1000)	INY implied	o	2

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The contents of the Y register are incremented by 1, setting N=1 if the result has bit 7=1, and vice versa, and setting Z=1 if the result is zero, and vice versa. A zero result is obtained by incrementing #$FF. Note that the carry bit is unchanged.

Uses: Like DEX, DEY, and INX this command is often used to control loops; and like them it is often followed by a comparison (CPY) to check whether its exit value has been reached. See CPY for a typical example.

JMP

Jump to a new location anywhere in memory. PC := M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$4C (76 %0100 1100)	JMP absolute	ooo	3
$6C (108 %0110 1100)	JMP (absolute)	ooo	3

Flags:

N	V	-	B	D	I	Z	C

Operation: JMP is the 6502 equivalent of a GOTO, transferring control to some non-sequential part of the program. An absolute JMP, opcode $4C, causes the following byte to be transferred to the low byte of the program counter and the next-but-one to the high byte of the program counter, resulting in a jump. The indirect absolute jump is more elaborate, and takes longer: PCL and PCH are loaded from the address following JMP and from the next address respectively. This is the only absolute indirect command available on the 6502.

Uses: JMP, unlike JSR, keeps no record of its present position, and transfers control unconditionally to its new destination. The resulting code is not

```
        LDA PTRLO
        LDY PTRHI
        JMP ADD
```
relocatable and in the case of ROM routines not usually portable between CBMs, so branching may be preferable where possible- i.e. in short programs.

```
        CMP #$2C   ;',
        BEQ +3
        JMP ERROR
        ...
```
These extracts from programs demonstrate how JMP is used. The first loads two pointers to a work area for calculations; a routine to add accumulator #1 to this data is jumped to. The second is part of a parsing subroutine which checks for a comma in a BASIC line; if the comma has been omitted, an error message is printed to inform the user of this fact.

Notes: [1] Indirect addressing. This is a 3-byte command which therefore looks like this: JMP ($0072) or JMP ($7FF0). A concrete example is the IRQ vector in the CBMs. When a hardware interrupt occurs, an indirect jump to ($0090) takes place. (($0129)in BASIC 1). A look at this region of RAM with the monitor reveals something like this:

```
        .:  0090 2E E6 17 FD 89 C3 00 FF     NOTE: THIS IS BASIC 2
```
So JMP ($0090) is equivalent to JMP $E62E. And JMP ($0092) jumps to $FD17. Pairs of bytes can be used in this way to form an indirect jump table. Note that this instruction has a bug: JMP ($02FF) takes its new address from $02FF and *$0200,* not $0300.

[2] JSR, RTS, and JMP. As the depth of subroutine nesting grows with increasing program complexity, inevitably some subroutines which themselves call other subroutines develop code like this:

```
        LOAD VALUES
        JSR PROCESS
        JSR CHECK
        RTS
```
And the point is that a subroutine call followed by a return is exactly identical to a jump, except that the stack use is less and the timing is shorter. Replacing JSR CHECK/ RTS by JMP CHECK is a common trick.

JSR

Jump to a new memory location saving the return address.
S := PC+2 H, SP := SP-1, S := PC+2 L, SP := SP-1, PC := M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$20 (32 %0010 0000)	JSR absolute	ooo	6

Flags: | N V - B D I Z C |

Operation: JSR is the 6502 equivalent of a GOSUB, transferring control to another
part of the program until an RTS is met, which has an effect like RETURN.
Like BRK, this instruction saves PC+2 on the stack, which is the last byte
of the JSR command, RTS therefore has to increment the stored value in
order to execute a correct return. Note that no flags are changed by
JSR. RTS also leaves flags unaltered.

Uses: [1] JSR is a very valuable command and is used a great deal in complex
programs: see for example the ROM BASIC interpreter. It has drawbacks
which are similar to those of JMP. In particular, JSR is not relocatable
except as regards fixed addresses such as those in ROM; and even these
don't usually carry over between ROMs. The exception is the so-called
'kernel' commands. JSR $FFE4 is a GET command and is relocatable in the
usual sense between any CBM ROM. Note that the 6809 has 'BSR', branch
on subroutine, with both short and long offsets permitted, which overcomes
the relocatability difficulties*.

```
LOOP JSR FFE4
     BNE LOOP

     LDA CHAR
     JSR FFD2
0300 BNE 0305
0302 JSR 0308
0305 JMP IRQCONT
0308 PROCESS
...  ...
     RTS
```

The first two examples here show how ROM routines
in the kernel may be used. It is not necessary to
know how they operate; all that's needed is know-
ledge of their principal features, which in these
examples are that data is transferred by the accum-
ulator, and that the flags are set by FFE4 as though
LDA had been used. So the first example loops until
a non-null byte has been fetched; the second loads
a byte from memory and outputs it to cassette or
printer or whatever. The third example, part of a
routine inserted into the IRQ servicing routine,

```
030D JSR EOF9;  INCREMENT GETCHR ADDRESS
0310 JSR CC9F;  EVALUATE EXPRN; PUT IN ACC#1
0313 JSR D72C;  ADD .5 TO ACC#1 TO ROUND
0316 JSR D6D2;  CONVERT ACC#1 TO INTEGER IN ($11)
0319 JSR C52C;  SEARCH FOR LINENUMBER IN BASIC
031C BCS 0321;  CARRY SET IF FOUND
031E JMP C7EB;  ?UNDEF'D STATEMENT ERROR
...  ...
```

has checked for some condition - possibly a particular keypress - and, if
the condition was true, calls a subroutine before returning to the interrupt
servicing as usual. This example illustrates the point made before about the
problem of relocation. Suppose the routine were shifted to a new part of
RAM. It would reappear as: 0200 BNE 0205/ 0202 JSR 0308/ 0205 JMP IRQCONT/
0208 PROCESS ... RTS. What is wanted is JSR 0208. See Chapter 14 on this
subject.

The fourth example is part of a computed GOTO routine for BASIC, which
uses ROM routines (in fact, BASIC 2 routines). BASIC subroutines provide
debugged code, but need rewriting to cope with each new issue of ROM.

Notes: See RTS for the PLA/ PLA construction which 'pops' one subroutine return
address from the stack. RTS also explains the special construction in which
an address (minus 1!) is pushed onto the stack, generating a jump when
RTS occurs. Finally, see JMP for a note on the way in which JSR.../RTS
may be replaced by JMP... .

---

*Very often there is no need to worry about this aspect of the 6502, of course.

LDA

Load the accumulator with a byte from memory.   A:=M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$A1 (161 %1010 0001)	LDA (zero page,X)	oo	6
$A5 (165 %1010 0101)	LDA zero page	oo	3
$A9 (169 %1010 1001)	LDA # immediate	oo	2
$AD (173 %1010 1101)	LDA absolute	ooo	4
$B1 (177 %1011 0001)	LDA (zero page),Y	oo	5*
$B5 (181 %1011 0101)	LDA zero page,X	oo	4
$B9 (185 %1011 1001)	LDA absolute,Y	ooo	4*
$BD (189 %1011 1101)	LDA absolute,X	ooo	4*

*Add 1 if page boundary crossed

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: Loads the accumulator and sets the zero flag Z to 1 if the accumulator now holds zero (i.e. all bits = 0). Bit 7 is copied into the N ('negative') flag. No other flags are altered.

Uses: [1] General transfer of data from one part of memory to another needs a temporary intermediate store of data, which A (or X or Y) can be. As an

```
LDX #00        example, this program transfers 256 consecutive
LDA 7000,X     bytes of data from $7000 ff to $8000 ff. The
STA 8000,X     accumulator is alternately loaded with data and
DEX            written to memory.*
BNE -9
```

[2] Some binary operations use the accumulator: ADC, SBC, and CMP all require A to be loaded before adding/ subtracting/ comparing. The addition or whatever can't be made directly between two RAM locations. (Even if it could the opcode would make a 5-byte instruction).

```
LDA 97   ; WHICH KEY?
CMP #FF  ; PERHAPS NONE?
BNE KEY  ; BRANCH IF KEY
```

[3]   LDA $E843   When the CBM is switched on, the code it executes contains this instruction. It initialises a register by reading from it. The value is not important; the fact of reading out is.

LDX

Load the X register with a byte from memory.   X:=M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$A2 (162 %1010 0001)	LDX # immediate	oo	2
$A6 (166 %1010 0101)	LDX zero page	oo	3
$AE (174 %1010 1110)	LDX absolute	ooo	4
$B6 (182 %1011 0101)	LDX zero page,Y	oo	4
$BE (190 %1011 1110)	LDX absolute,Y	ooo	4*

*Add 1 if page boundary crossed

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: Loads X from memory and sets Z=1 if X now holds zero. Bit 7 from the memory is also copied into N. No other flags are altered.

Uses: [1] Transfer of data and holding temporary values (e.g. for comparisons). These closely resemble LDA (q.v.)

[2] X has two characteristics which distinguish it from A: it is in direct communication with the stack pointer and it can be used as an offset with indexed addressing. (There are other differences too!) So constructions like these are common: LDX #$FF/ TXS   and   LDX #$00/   ... / DEX/ BNE ...

*Some chips (e.g. Z80) have documentation in which 'load' means 'load <u>into</u> memory'.

LDY

Load the Y register with a byte from memory. Y:=M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$A0 (160 %1010 0000)	LDY # immediate	oo	2
$A4 (164 %1010 0100)	LDY zero page	oo	3
$AC (172 %1010 1100)	LDY absolute	ooo	4
$B4 (180 %1011 0100)	LDY zero page,X	oo	4
$BC (188 %1011 1100)	LDY absolute,X	ooo	4*

*Add 1 if page boundary crossed

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: Loads Y from memory and sets Z=1 if Y now holds zero. Bit 7 from memory is copied into N. No other flags are altered.

Uses: [1] Transfer of data and storage of temporary values: Cp. LDX, LDY.

[2] Since Y can be used as an index, and can be incremented/ decremented easily, it is often used in loops. However, X generally has more combinations of addressing modes in which it is used as an index; often therefore X is reserved for indexing, while A and Y between them process other parameters. When indirect addressing is used this preference between X and Y is reversed, since (usually) LDA (addr,X) is less useful than LDA (addr),Y.

```
        LDY #00    ; X HOLDS LENGTH
LOOP DEX           ; DECREMENT IT
     BEQ EXIT      ; EXIT WHEN 0
     LDA (PTR),Y; LOAD ACCUMULATOR
     JSR PRINT     ; PRINT SINGLE CHR
     CMP #0D       ; EXIT IF
     BEQ EXIT      ; 'RETURN'
     BNE LOOP      ; CONTINUE LOOP
```

This (admittedly rather unexciting) example shows how A,X, and Y have distinct roles; the ROM routine to print the character is assumed to return the original X and Y values (as in fact it does).

LSR

Shift memory or accumulator right one bit. [0]→7 6 5 4 3 2 1 0→[C]

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$46 (70 %0100 0110)	LSR zero page	oo	5
$4A (74 %0100 1010)	LSR accumulator	o	2
$4E (78 %0100 1110)	LSR absolute	ooo	6
$56 (86 %0101 0110)	LSR zero page,X	oo	6
$5E (94 %0101 1110)	LSR absolute,X	ooo	7

Flags:

N	V	-	B	D	I	Z	C
0						x	x

Operation: Moves the contents of memory or the accumulator right by one bit position, putting 0 into bit 7 and the negative flag, and moving the rightmost bit, bit 0, into the carry flag. Z is set to 1 if the result is zero, and cleared if not. Z can therefore only become 1 if the location before LSR held either #0 or #1.

Uses: [1] This instruction is similar to ASL (and could just as well be called 'arithmetic shift right'). A byte is halved by this instruction (unless in decimal mode, with D set), its remainder moving into the carry flag. See for example the machine-code corresponding to the BASIC 'SET' command, which halves the coordinates of a point in 'high resolution' graphics to fit the screen. A rotate command can save the carry bit; see ROL.

[2] Miscellaneous uses include: (i) LSR/ LSR/ LSR/ LSR move a high nybble into a low nybble; (ii) LSR/ BCC tests bit 0, and branches if it was not set to 1; (iii) LSR turns off bit 7; sometimes this is an easy way to convert a negative number into its positive equivalent, when the sign is stored as a separate byte. The BASIC ABS function for instance does this.

NOP

No operation.

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$EA (234 %1110 1010)	NOP implied	o	2

Flags:

N	V	-	B	D	I	Z	C

Operation: Does nothing. (Well, not quite nothing - it increments the program counter and continues with the next opcode).

Uses: [1] Filling disused portions of program; this is useful with hand 'assembly' and other methods where recalculation of branch addresses and so on can't be done easily. Some CBM ROM has this feature: for example, BASIC 2's PEEK has many NOPs left over from deleting the PEEK protection from BASIC 1.

[2] Conversely, when writing machine-code which hasn't been thoroughly thought out beforehand (I am assured that this does very rarely happen) a large block of NOPs, or occasional sprinkling of them, can make the task of editing the code and inserting corrections easier. There is some time lost in this process; NOP can be used as part of a timing loop.

ORA

Logical inclusive OR of memory with the accumulator.   A:=A inclusive OR M

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$01 (1 %0000 0001)	ORA (zero page,X)	oo	6
$05 (5 %0000 0101)	ORA zero page	oo	3
$09 (9 %0000 1001)	ORA # immediate	oo	2
$0D (13 %0000 1101)	ORA absolute	ooo	4
$11 (17 %0001 0001)	ORA (zero page),Y	oo	5
$15 (21 %0001 0101)	ORA zero page,X	oo	4
$19 (25 %0001 1001)	ORA absolute,Y	ooo	4*
$1D (29 %0001 1101)	ORA absolute,X	ooo	4*

    *Add 1 if page boundary crossed

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: Performs the inclusive OR of the 8 bits currently in the accumulator with the 8 bits referenced by the opcode. The result is stored in A. If either bit is 1, the resulting bit is set to 1, so that for example: %0011 0101 ORA %0000 1111 is %0011 1111. The negative flag N, and the zero flag Z, are set or cleared depending on the result.

Uses: [1] 'Flagging in' a bit or bits. This is the opposite process to 'masking out' bits, as described under AND. These two examples are typical extracts

     LDA (PTR),Y   from larger routines which use this function: the
     ORA #$80   first loads a character from the screen, then sets
                  the high bit, reversing it - unless the character
     ORA $96   was already in reverse, in which case it is left
     STA $96   unchanged. (Cp. EOR). The second is the method
by which an error code of #1,#2,#4 or whatever, held in A, is flagged into the status byte ST. ST is stored in $96. Note the necessity for STA $96; without it, only A holds the correct value of ST.

[2] Other miscellaneous uses include the testing of several bytes for conditions which are intended to be true for each of them, for instance that 3 consecutive bytes are all zero, or that several bytes all have bit 7 equal to zero. LDY #00/ LDA (PTR),Y/ INY/ ORA (PTR),Y/ INY/ ORA (PTR),Y/ BNE ... branches if one or more bytes contains a non-zero value.

PHA

Push the accumulator's contents onto the stack. S := A, SP := SP – 1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$48 (72 %0100 1000)	PHA implied	o	3

Flags: | N V – B D I Z C |

Operation: A is put into the stack at the current position pointed to by the stack
pointer; the stack pointer is decremented. This diagram illustrates the
position before and after the 'push':-

```
$0100   --- MAXIMUM EXTENT OF THE STACK ---   $01FF
     |              ↑|        STACK IN USE         |
              SP (STACK POINTER)
     |            ↑|A|        STACK IN USE         |
            SP (STACK POINTER)
```
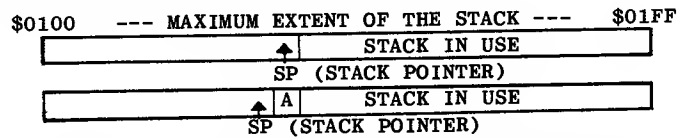
Uses: This instruction is used for temporary storage of bytes; examples include
intermediate values of calculations produced during the parsing of numeric
expressions, temporary store while A is processed for later recovery, storage
when swapping bytes, and storage of A,X and Y registers at the start of a
subroutine, to be recovered on exit. The example shows how a printout

```
PHA
AND #7F   ; MASK OFF BIT 7
JSR PRINT; OUTPUT 1 CHR
PLA
BPL LOOP  ; CONTINUE IF BIT 7=0
```
routine works which is designed to end
when the high bit of a letter in the table
is 1. The output requires the high bit
to be set to 0; but the original value is
recoverable from the stack and may be
used in a test for the terminator at the end of the message.

```
PHA
TXA
PHA
TYA
PHA
```
This next example shows how A,X and Y
can all be saved on the stack for future
recovery. Naturally these three registers
must be recovered in the reverse order:
PLA/ TAY/ PLA/ TAX/ PLA.

Note: 'Push' is a rather misleading term for the action of this instruction; and the
well-known 'stack of plates' analogy is also seriously misleading, and no
doubt responsible for the puzzlement with which the stack is often viewed.

PHP

Push the processor status register's contents onto the stack. S := PSR, SP := SP – 1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$08 (8 %0000 1000)	PHP implied	o	3

Flags: | N V – B D I Z C |

Operation: The operation is exactly similar to PHA, except that the processor status
register is put in the stack. The PSR is unchanged by the push.

Uses: Stores the entire set of flags, usually either to be recovered later and
displayed by a monitor program, or for recovery followed by a branch.
PHP   This leaves the stack in the condition it was found; it also
PLA   loads A with the flag register. The first example under PHA
shows the recovery of A, which sets either/neither of N and V. The same
effect can be achieved, with the full range of flags, with LDA CHR/ PHP.

PLA

Pull the stack into the accumulator. SP:=SP+1, A:=S

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$68 (104 %0110 1000)	PLA implied	o	4

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The stack pointer is incremented, then the RAM address to which it points is read and loaded into A, setting the N and Z flags. The effect is similar to LDA. This diagram illustrates the position before and after the 'pull':-

```
$0100   --- MAXIMUM EXTENT OF THE STACK ---   $01FF
      [            |  A|  STACK IN USE           ]
                  SP (STACK POINTER)
      [              |A|  STACK IN USE           ]
                  SP (STACK POINTER)
```
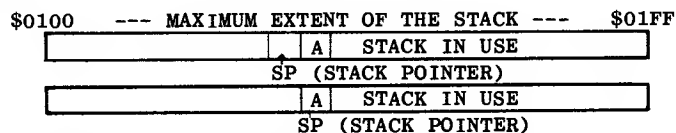
Uses: [1] PLA is the converse of PHA. It retrieves values put on the stack by PHA, in the reverse order. So for example:

PLA	This code leaves the stack unchanged, but leaves
PHA	A holding the contents of the current 'top' of the
	stack. Flags N and Z are set as though by LDA.
PHP	This next example shows how the processor status
PLA	register may be examined by loading it into the

accumulator from the stack. For example, if A now has bit 3 equal to 1, the decimal mode is set.

[2] A frequent use of PLA is to 'throw away' the top two bytes of the stack. This is equivalent to adding 2 to the stack pointer. This is done to 'pop' a return address from the stack; in this way, the next RTS which is encountered will not return to the previous JSR, but to the one before it. It assumes, of course, that the stack has not been added to since the JSR. The following short example illustrates the point:-

033A LDX #FF	Enter this trio of routines into a CBM with a monitor
033C JSR 0340	(i.e. not BASIC 1, unless a monitor is specially
033F BRK	loaded). However, use $0350 RTS at first. If the
0340 JSR 0350	routine is run, by .G 033A which goes to the
0343 LDX #01	machine-code address $033A and executes the code
0345 RTS	there, on BRK the registers are displayed, among
	them X, which equals #01. This is ordinary nested
0350 PLA	subroutine logic; the earlier value of X is over-
0351 PLA	written in the subroutine starting at $0340. But
0352 RTS	if you use routine $0350 as it appears here, the X

register holds #FF on BRK. The return address of $0342, as it is held in the stack, has been lost, so RTS goes straight to BRK.

PLP

Pull the stack into the processor status register. SP:=SP+1, PSR:=S

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$28 (40 %0010 1000)	PLP implied	o	4

Flags:

N	V	-	B	D	I	Z	C
x	x		x	x	x	x	x

Operation: The operation is exactly similar to PLA, except that the processor status register, not the accumulator, is loaded from the stack.

Uses: Recovers previously stored flags with which to test or branch. See the notes on PHP. This can also be used to experiment with the flags, perhaps trying to set bit 5 (which is set to 1) or to set V, for example.

ROL

Rotate memory or accumulator *and the carry flag* left one bit.

`◄C◄ 7 6 5 4 3 2 1 0 ◄◄`
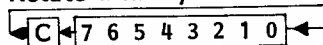
INSTRUCTION	ADDRESSING	BYTES	CYCLES
$26 (38 %0010 0110)	ROL zero page	oo	5
$2A (42 %0010 1010)	ROL accumulator	o	2
$2E (?? %0010 1110)	ROL absolute	ooo	6
$36 (54 %0011 0110)	ROL zero page,X	oo	6
$3E (62 %0011 1110)	ROL absolute,X	ooo	7

Flags:

N	V	-	B	D	I	Z	C
x						x	x

Operation: 9 bits, consisting of the contents of memory referenced by the instruction, and the carry bit, are 'rotated' as the diagram shows. In the process, C is changed to what was bit 7, bit 0 becomes the previous C, and the negative flag, bit 7, becomes the previous bit 6. In addition, Z is set or cleared depending on the new memory contents.

Uses: [1] Like ASL, ROL doubles the contents of the byte which it references, but in addition the carry bit may be used to propagate the overflow from such a doubling. Multiplication and division routines take advantage of this property where a chain of consecutive bytes has to be moved one bit leftward. ROR is used where the direction of movement is rightward, and often these commands are used together.

```
i.  ASL $4000
    ROL $4001
    ROL $4002

ii. ROL $7FE3,X
    CLC
    ROR $7FE3,X
```

The first example moves the entire 24 bits of $4000 - $4002 over by 1 bit, introducing 0 into the rightmost bit; if there is a carry, the carry flag will be 1. The second example demonstrates an alternative method for clearing bit 7 of a location to the more obvious LDA / AND #7F/ STA . It is however far 'slower', taking half as long again to execute. All rotations and shifts are slow, with the exception of operations on A which are as fast as the 6502 allows. If possible, therefore, rotations and shifts should use A. Parity bits are a good example of the type of application for which ROL is ideal. Zak's 6502 book has an example in which ONECNT holds the count of 1s in a byte, and A holds the character; what should follow is:

```
ROL A      ; BIT 7 OF THE CHR NOT YET KNOWN
LSR ONECNT ; PUTS 0 (EVEN), 1 (ODD) INTO C
ROR A      ; INCORPORATE C INTO BIT 7
```

[2] Like ASL, ROL may be used before testing N,Z, or C, especially N.

```
ROL A      ; ROTATE 1 BIT LEFTWARD
BMI BRANCH; BRANCHES IF BIT 6 WAS ON
```

ROR

Rotate memory or accumulator *and the carry flag* right one bit.

`►C► 7 6 5 4 3 2 1 0 ►` *NB: old 6502s lack this instruction, but all PET/CBMs have it.*

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$66 (102 %0110 0110)	ROR zero page	oo	5
$6A (106 %0110 1010)	ROR accumulator	o	2
$6E (110 %0110 1110)	ROR absolute	ooo	6
$76 (118 %0111 0110)	ROR zero page,X	oo	6
$7E (126 %0111 1110)	ROR absolute,X	ooo	7

Flags:

N	V	-	B	D	I	Z	C
x						x	x

Operation: 9 bits, consisting of the contents of memory referenced by the instruction, and the carry bit, are 'rotated' as the diagram shows. C becomes what was bit 0, bit 7 becomes the previous C, and Z is set or cleared depending on the byte's current contents. For applications, see ROL.

RTI

Return from interrupt.   SP:=SP+1, PSR=S, SP:=SP+1, PCL:=S, SP:=SP+1, PCH:=S

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$40 (64 %0100 0000	RTI implied	0	6

Flags:

N	V	-	B	D	I	Z	C
x	x			x	x	x	x	x

Operation: RTI takes 3 bytes from the stack, deposited there when the hardware triggered the interrupt, assuming that the stack has been tidied up before RTI with equal numbers of pulls following the pushes. The processor status flags are recovered as they were when the interrupt occurred, and the program counter is restored so that the program resumes operation at the byte at which it was interrupted. Note that A,X, and Y are not saved or recovered automatically in this way, but must be saved by the interrupt processing and restored immediately before RTI. If you follow the vector stored in CBM ROM at ($FFFE), you will see this operation taking place.

Uses: [1] Obviously, to resume after an interrupt. Unless you are using your own hardware to generate interrupts, or programming the VIA to generate interrupts, this instruction is unlikely to be useful to you.

[2] However, it is possible, as with RTS, to exploit the automatic nature of this command to execute a jump by pushing 3 bytes on the stack, imitating an interrupt, then using RTI to pop the addresses and processor status.

```
        LDA HI
        PHA              This routine, by simulating the stack contents
        LDA LO           left by an interrupt, jumps to 256*HI + LO (in
        PHA              decimal!) with its processor flags equal to what-
        LDA PSR          ever was pushed on the stack as 'PSR'.
        PHA
        RTI
```

RTS

Return from subroutine.   SP:=SP+1, PCL:=S, SP:=SP+1, PCH:=S, PC:=PC+1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$20 (32 %0010 0000)	JSR implied	0	6

Flags:

N	V	-	B	D	I	Z	C

Operation: RTS takes 2 bytes from the stack, increments the result, and jumps to the address found by putting it into the program counter. It is similar to RTI, but does not change the processor flags, since an important feature of subroutines is that, on return, flags should be usable. Also, unlike RTI in which the address saved is the address to return to, RTS must increment the address it fetches from the stack, which points to the second byte after a 'JSR'. (Presumably, the chip uses the routine for BRK and for JSR in common).

Uses: [1] To return after a subroutine. This is entirely straightforward:-

```
033A JSR 0350; CALL SUBROUTINE
033D RTS       ; RETURN TO BASIC

0350 STA $8000; PUT A IN SCREEN
0353 RTS       ; RETURN
```

SYS 826 calls the routine at 033A which is a short program, calling a subroutine before returning to BASIC. The subroutine starts at 0350 and continues until RTS is found; this example simply pokes A into the top-left corner of the screen before returning.

Notes: [1] See PLA for the technique for discarding subroutines' return addresses. Also see JMP for the essential identity of JSR .../ RTS and JMP... . Finally, as with RTI, a jump can be generated by pushing bytes onto the stack and executing RTS, even though no subroutine call was actually made.

SBC

Subtract memory with borrow from accumulator. A := A - M - (1-C)

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$E1 (225 %1110 0001)	SBC (zero page, X)	oo	6
$E5 (229 %1110 0101)	SBC zero page	oo	3
$E9 (233 %1110 1001)	SBC # immediate	oo	2
$ED (237 %1110 1101)	SBC absolute	ooo	4
$F1 (241 %1111 0001)	SBC (zero page),Y	oo	5*
$F5 (245 %1111 0101)	SBC zero page,X	oo	4
$F9 (249 %1111 1001)	SBC absolute,Y	ooo	4*
$FD (253 %1111 1101)	SBC absolute,X	ooo	4*

*Add 1 if page boundary crossed

Flags:

N	V	-	B	D	I	Z	C
x	x					x	x

Operation: It is usual to set the carry bit before this operation, or precede it by an operation which is known to leave the carry bit set. Then SBC appears to subtract the data referenced by the addressing mode from the accumulator. If the carry flag is still set, this indicates that the result did not 'borrow', i.e. that the accumulator's contents were greater than or equal to the data. When C is clear, the data exceeded the accumulator's contents and C shows that a 'borrow' is needed. Within the chip, A is added to the 2's complement of the data and to the complement of C.*This affects the N,V,Z, and C flags.

Uses: [1] Single byte subtraction. This has quite a number of applications:

```
SEC    ; CARRY FLAG IN KNOWN STATE
LDA CHR; ASCII NUMERAL ('0'=#30 &C)
SBC #2F; CONVERT TO BYTE 00 TO 09
JSR OUT; PRINT OR CALCULATE WITH VALUE
```

The first is a conversion routine which simply subtracts a fixed amount from an ASCII numeral to convert it into a byte value 0-9, perhaps for the purposes of calculation. Numerals have ASCII value #30-#39 (48-57 decimal) which subtraction converts to #0-#9.

```
         LDA HORIZ; LOAD CURRENT CORSOR POSN
         SEC    ; CARRY FLAG SET DURING LOOP
LOOP SBC #0A  ; SUBTRACT 10S UNTIL CARRY..
         BCS LOOP ; ..IS CLEAR (I.E. A IS NEG)
         EOR #FF  ; FLIP BITS AND ADD 1 TO
         ADC #01  ; CONVERT TO POSITIVE.
```

The next example is more elaborate and is a detail from PRINT. When processing the comma in a print statement, the cursor is moved to position 0,10,20, etc. Suppose the cursor is now at 17 horizontally; we subtract 10s until the carry flag is clear, when A will hold -3. The 2's complement is 3, and 3 spaces or cursor rights take us to the correct position on the screen. Note that ADC #01 adds 1 only; the carry flag is known to be zero by that stage.

[2] Double byte subtraction. The point about subtracting one 16-bit number from another is that the borrow is performed automatically by SBC. First C is set to 1; then the low byte is subtracted; then the high byte is subtracted, with borrow if the low bytes were such as to make this necessary.

```
SEC
LDA LO
SBC #6A
STA LO
LDA HI
SBC #02
STA HI
```

In this example #026A is subtracted from the contents of addresses (or data) LO and HI. The result is replaced in LO and HI. Note that SEC is performed once only. In this way, borrowing is performed properly. For example: suppose the address from which #26A is to be subtracted holds #1234. When #6A is subtracted from #34, the carry flag is cleared, so that #2 and 1 is subtracted from the high byte #12.

Subtraction is sometimes used twice, in a way which clears the carry bit for values in A within a certain range. In this way, JSR .../BCC .. may be used. See Chapter 14 on CHRGET for an example. Other examples include a check for alphabetic characters (A-Z) only.

Note: *Strictly speaking, 1's complement and C are added to A. For example: SEC/ LDA #A0/ SBC #55 takes #A0 and adds AA and 1=#4B with C=1. Whereas SEC/ LDA #A0/ SBC #BC takes #A0 and adds 43 and 1=#E4 with C=0.

SEC

Set the carry flag to 1.   C:= 1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$38 (56 %0011 1000)	SEC implied	o	2

Flags:

N	V	-	B	D	I	Z	C
							1

Operation: Sets the carry flag; this is the opposite of CLC, which clears it.

Uses: Used whenever the carry flag has to be put into a known state; usually
SEC is performed before subtraction (SBC) and CLC before addition (ADC)
since the numeric values the used are the same as in ordinary arithmetic.
See ADC and SBC for examples.

SED

Set the decimal mode flag to 1.   D:= 1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$F8 (248 %1111 1000)	SED implied	o	2

Flags:

N	V	-	B	D	I	Z	C
				1			

Operation: Sets the decimal flag; this is the opposite of CLD, which clears it.

Uses: Sets the mode to BCD arithmetic ('binary coded decimal') in which each
nybble holds a decimal numeral. For example, ten is held as #10 and ninety
as #90.   Two thousand four hundred and fifteen is #2415 in 2 bytes. ADC
and SBC are designed to operate in this mode as well as in binary, but
the flags no longer have the same meaning, except C.

Where indefinite precision is required with calculations buffers can be
allocated for storage and addition of large numbers. No precision will be
lost within the range specified. (It's possible to do a similar processing job
on numerals stored in single bytes; these of course occupy twice as much
space, and take longer to work with). This mode is unused in BASICs 1-4.

SEI

Set the interrupt disable flag to 1.   I:= 1

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$78 (120 %0111 1000)	SEI implied	o	2

Flags:

N	V	-	B	D	I	Z	C
					1		

Operation: Sets the interrupt disable flag; this is the opposite of CLI, which clears it.

Uses: When this flag has been set, no interrupts are processed by the chip, except
non-maskable interrupts (which have higher priority) and reset. With CBM
equipment, ordinary maskable hardware interrupts occur every 1/50th or 1/60th
of a second; setting I will cause the processing associated with this, i.e.
clock (TI and TI$), keyboard and cassettes, to cease until CLI. Maskable
interrupts are processed by ($FFFE), like BRK. If the vector in the very top
locations of the BASIC ROM is followed, the interrupt servicing routines can
be found. These are not (entirely) hardwired in
ROM: the vectors use an address in RAM before
jumping back to ROM. So the example here is a
typical initialisation routine to redirect the vector
into the user's own program, where it may set a
musical tone, process a repeat key, turn off STOP,
or whatever. See Chapter 13 for more detail.

```
033A  SEI
033B  LDA #45
033D  STA 90
033F  LDA #03
0341  STA 91
0343  CLI
0344  RTS
```

STA

Store the contents of the accumulator into memory.   M:=A

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$81 (129 %1000 0001)	STA (zero page,X)	o o	6
$85 (133 %1000 0101)	STA zero page	o o	3
$8D (141 %1000 1101)	STA absolute	o o o	4
$91 (145 %1001 0001)	STA (zero page),Y	o o	6
$95 (149 %1001 0101)	STA zero page,X	o o	4
$99 (153 %1001 1001)	STA absolute,Y	o o o	5
$9D (157 %1001 1101)	STA absolute,X	o o o	5

Flags: | N V - B D I Z C |
| |

Operation: The contents of A are sent to the address referenced by the opcode.
All registers and flags are unchanged.

Uses: [1] Transfer of blocks of data from one part of memory to another needs a temporary intermediate store; this can be A,X, or Y. This is alternately loaded and stored. See LDA. Another example in addition to the one given

```
        LDY #00
LOOP JSR LOADCHR; LOAD A WITH CHR
        STA (PTR),Y; STORE A INTO MEM
        INC PTR   ; INCREMENT POINTER
        BNE TEST
        INC PTR+1
TEST JSR TESTPTR; TEST FOR LIMIT
        BNE LOOP  ; CONTINUE IF NOT END
```

there is this outline of a routine, where LDA is carried out by some routine and there is a further routine to check whether all characters have yet been moved.

[2] Binary operations using the accumulator, notably ADC and SBC, are performed within the accumulator; a common bug in machine code programs is the omission to save the result:

```
        LDA $96; ST BYTE
        AND #$FD;BIT 1 OFF
        STA $96 ;REMEMBER THIS!
```

[3] Another very common use is storing isolated values during initialisation, to set the contents of certain locations to known values:

```
        LDA #89
        STA 94 ; SETS ($94)
        LDA #C3
        STA 95 ; TO $C389.
        LDA #17
        ...   ; &C.
```

STX

Store the contents of the X register into memory.   M:=X

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$86 (134 %1000 0110)	STX zero page	o o	3
$8E (142 %1000 1110)	STX absolute	o o o	4
$96 (150 %1001 0110)	STX zero page,Y	o o	4

Flags: | N V - B D I Z C |
| |

Operation: The contents of X are sent to the address referenced by the opcode.
All registers and flags are unchanged.

Uses: The uses are identical to those of STA; there is a tendency for X to be used as an index, so STX is less used than STA.

STY

Store the contents of the Y register into memory. M:=Y

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$84 (132 %1000 0100)	STY zero page	°°	3
$8C (140 %1000 1100)	STY absolute	°°°	4
$94 (148 %1001 0100)	STY zero page,X	°°	4

Flags:

N	V	-	B	D	I	Z	C

Operation: The contents of Y are sent to the address referenced by the opcode.
All registers and flags are unchanged.

Uses: STY resembles STX; the comments under STX apply.

TAX

Transfer the contents of the accumulator into the X register. X:=A

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$AA (170 %1010 1010)	TAX implied	°	2

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The byte in A is transferred to X. The N and Z flags are set as though
LDX had taken place.

Uses: This transfer is mostly used to set X for use as an index or a parameter,
or to temporarily hold A. These examples illustrate the type of thing. The

TXA	first is from a 'high resolution' screen plotting
EOR #FF	routine; the object is to plot a black dot in a
ORA 94	location with a coded value of 1,2,4 or 8 in $94.
EOR #FF	X on entry holds the position of the current
TAX	graphics character in a table. On exit X holds the
LDA TABLE,X	position of the new character. Intermediate calcul-
PLA	ations use the accumulator because there is no
TAX	'EOR with X' instruction. The second example is
PLA	a straightforward reconstruction of X and Y values
TAY	stored on the stack. This has to be done when
	returning from interrupt processing.

Note: Registers A,X, Y and the stack pointer are interchangeable with one
instruction in some cases, but not others. The connections are these:
Y ⇌ A ⇌ X ⇌ S .

TAY

Transfer the contents of the accumulator into the Y register. Y:=A

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$A8 (168 %1010 1000)	TAY implied	°	2

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The byte in A is transferred to Y. The N and Z flags are set as though
LDY had taken place.

Uses: See TAX; TAY is similar to this other instruction.

TSX

Transfer the stack pointer into the X register.   X:=SP

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$BA (186 %1011 1010)	TSX implied	o	2

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The stack pointer is transferred to X. Note that the stack pointer is always used in conjunction with $0100, that is, when the stack is accessed the high byte of RAM is always set to #1. The pointer itself is a single byte.

Uses: [1] To inspect the stack; in the case of CBM BASIC, for GOSUB and FOR tokens when processing NEXT and RETURN. The stack pointer is also used to estimate the amount of space left on the stack; again, CBM BASIC does this, typically printing ?OUT OF MEMORY ERROR if the stack is not sufficient for some manoeuvre. Since the pointer does not point at the last item pushed on the stack, but the byte below it, LDA $0101,X can be used to peek the last pushed item.

[2] This is sometimes used to store the stack pointer when a different (i.e. lower) part of the stack is temporarily moved to for processing.

TXA

Transfer the contents of the X register into the accumulator.   A:=X

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$8A (138 %1000 1010)	TXA implied	o	2

Flags:

N	V	-	B	D	I	Z	C
x						x	

Operation: The byte in X is transferred to A. The N flag and Z flag are set as though LDA had taken place.

Uses: See TAX.

TXS

Transfer the X register into the stack pointer.   SP:=X

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$9A (154 %1001 1010)	TXS implied	o	2

Flags: | N V - B D I Z C |
|---|

Operation: X is stored in the stack pointer. Now, PHA or PHP will place a byte
into the stack at $0100 + the new stack pointer, and PLA or PLP will pull
from the next byte up from this. Also RTI and RTS will return to addresses
determined by the stack contents at the new position of the stack.

Uses: [1] TXS initialises the stack on switchon, when the RESET line is activated,
and also in some BASIC commands like RUN and CLR. The stack extends
from $0100 - $01FF in principle; this is the area of RAM which is reached
as the stack pointer varies from #0 to #FF. Not all of this is used by PET/
CBM machines. Since the stack pointer is *decremented* when data is pushed
on to the stack - i.e. if memory is pictured starting at location zero on the
left and increasing rightward, the stack 'grows' to the left as data is push-
ed - the initial value is usually something like #FF on setting the machine's
starting values:

```
        LDX #FF
        TXS
```

[2] The other use is to switch to a new stack location. As a simple example,

```
        CLC              the routine presented here is an equivalent to
        TSX              PLA/ PLA which we've seen under RTS to be a
        TXA              'pop' command, deleting a subroutine's return
        ADC #02          address. Incrementing the stack pointer by 2 has
        TAX              the identical effect.  For a more complex example,
        TXS              see 'POP' in the BASIC reference section, which
```

is similar in conception but more complex in execution because of the
greater elaboration of BASIC compared to machine-code.

TYA

Transfer the contents of the Y register into the accumulator.   A:=Y

INSTRUCTION	ADDRESSING	BYTES	CYCLES
$98 (152 %1001 1000)	TYA implied	o	2

Flags: | N V - B D I Z C |
|---|
| x   x |

Operation: The byte in Y is transferred to A. The N flag and Z flag are set as
though LDA had taken place.

Uses: See TAX. The transfers TAX, TAY, TXA, and TYA all perform similar
functions.

CHAPTER 13: ROM ROUTINES AND THEIR USES

13.1 The RESET sequence.

As we saw in Chapter 11, the RESET line* in any 6502 is used to cause a jump to a standard address (FFFC). Most 6502-based equipment has a ROM routine which is jumped to on RESET; this includes CBM printer and disk systems. Disassembly of such routines can give useful information about a system. The input/ output chips' RAM locations and contents on reset, for example, show which registers are configured for input and which for output, which interrupts are enabled, and so on. BASICs 1 to 4 have similar reset routines, at FD38, FCD1, and FD16 in order. Chapter 15 lists the operations they perform. The routines are in two parts, like this:

(i) On RESET, or on JMP (FFFC), the routine at FD38, FCD1, or FD16 is run.

(ii) Most of the hardware-oriented aspects of the PET/CBM are initialised: the I/O chips are configured to match the system, the screen is cleared (to erase its 'random' garbage), the screen table set up in 40-column machines, the decimal flag cleared, and NMI, IRQ and other addresses set up.

(iii) Now the 'diagnostic sense pin' is checked; see Chapter 8 on 'Reset switches' for information about this.

(iv) Depending on the diagnostic sense pin, BASIC is initialised (this is usual) or MLM is entered, or, in BASIC 1, a diagnostic routine may be run.

(v) BASIC initialisation (at E0D2/ E116/ D3B6) sets the stack, USR address, CHRGET and the random number seed, BASIC start and end addresses, and other specific BASIC items. Finally, a loop tests RAM by writing #55 and #AA into RAM from $0400, until either $8000 is reached or the read-back value no longer equals the poked value, indicating end of RAM. This leaves the end-of-memory pointer set; by subtraction, the number of bytes free is computed and printed. So a 32K system prints 31743 bytes, since 1024 + 31743 = #7FFF. (If a RAM chip is *carefully* removed, you may get (say) 15359 bytes free, where 1024 + 15359 = #3FFF).

Any of these routines can be entered, bypassing the earlier initialisation. Chapter 5 has examples in SYS to clear the entire memory from $00, not just $0400. Normally the RAM below BASIC in untouched by RESET, which is why machine-code in the cassette buffers can sometimes survive a power-off and immediate power-on.

13.2 The interrupt routine.

The CBM interrupt-processing sequence has two branches, like RESET. One is for BRK commands, the other for interrupts generated by the screen, and used to control the keyboard, tape, and cursor. These branches are distinguished by the presence or absence of the B flag in the status register; both BRK and IRQ share the same vector, (FFFE). Disassembly reveals that A,X, and Y are saved on the stack (an interrupt saves the program counter and status register, but not the other registers), and an indirect jump is made to (90) for IRQ and (92) for BRK. Either of these addresses may be poked to point at user-written routines; Chapter 8 has several examples, involving repeat keys, keyboard redefinition, and so on; Chapter 9 also includes examples. The ROM in BASIC 1 uses ($0219) and ($021B) for IRQ and BRK.

Example: displaying a part of memory continually on the screen The fairly short machine-code routine presented here works like this:

```
      SYS 634  awaits input.  0100 000A displays 10 characters from $0100 onwards;
                              0200 0028 displays 40 characters from the input buffer.
```

It uses MLM routines to input a pair of hex addresses, the first of which is stored in (FD), the second in (FB). Some operations will overwrite these addresses; this can be avoided by using (say) $00 - $02, at the cost of a slightly longer program. The IRQ is diverted to run part of this routine, beginning A0 00 ..., which continually, i.e. every 50th or 60th of a second rewrites its data at the top of the screen. In this way, numbers can be watched being formatted; the input buffer can be watched; and so on. A maximum of 256 bytes can be displayed - again, there is no reason why more can't be used. Note that SYS 634 then 8001 00FF continually shifts the top of the screen.

```
      SYS 671 turns the routine off.
```

*Reference to 6502 data-sheets shows this process is as follows: the line is to be held low on switchon; it must be held low until the voltage V_{cc} reaches its operating level, and a little longer; when RESET now goes high, the chip resets itself in six cycles, sets the interrupt disable flag, and loads the program counter from (FFFC).

DISPLAY BYTES USING IRQ: BASIC 2 VERSION   BASIC 4 VERSION

.:	027A	20	A7	E7	20	97	E7	20	EB	027A 20 54 D7 20 44 D7 20 98
.:	0282	E7	20	A7	E7	78	A9	02	85	0282 D7 20 54 D7 78 A9 02 85
.:	028A	91	A9	90	85	90	60	A9	00	028A 91 A9 90 85 90 60 A0 00
.:	0292	B1	FD	99	00	80	C8	C4	FB	0292 B1 FD 99 00 80 C8 C4 FB
.:	029A	D0	F6	4C	2E	E6	78	A9	E6	029A D0 F6 4C 55 E4 78 A9 E4
.:	02A2	85	91	A9	2E	85	90	60	xx	02A2 85 91 A9 55 85 90 60 xx

Example: pause loop   By combining
a redirected interrupt with a ROM
routine to GET a character from the   027A   JSR FFE4   ;GET CHR. FROM KEYBOARD IN A
keyboard, we can write a pause   027D   CMP #40   ; IS IT @?
loop. This example is activated by   027F   BNE 0288   ;NO - CONTINUE INTERRUPT
'@', a key for which there is usually   0281   JSR FFE4   ;GET ANOTHER CHARACTER IN A
little use. The interrupt routine at   0284   CMP #40
this point enters a loop, in which it   0286   BNE 0281   ;KEEP LOOPING UNTIL @
will remain until '@' is pressed again.   0288   JMP E455   ;OR E62E WITH BASIC 2
Additional code can be written to
alter the address in ($90) to $027A, and to change it back to its normal value; or it
can be activated by changing IRQ in the monitor to 027A or whatever other address
this code is put into (it it relocatable). LIST, for example, is stopped by the routine,
and can be continued at will. BASIC 4 has routines of this sort built in.

13.3 Other ROM routines.

Disassembly of ROM routines   CBM ROMs are easily disassembled, and (except for
BASIC 1) have no peek protection. Chapter 15 has a guide to all CBM BASIC ROMS so
far issued. Nevertheless it is not very easy to use such routines, since they are all
(with few exceptions) very complex. The next page has an example of a disassembly
(of 'OPEN') written out in English; this sort of translation is essential if a ROM rout-
ine is to be used and reused. Less detailed examples include the flowchart for PRINT
(in Chapter 5) and for the machine-language monitor (in Chapter 10). Unannotated
disassembler listings are difficult to follow. Many important routines are collected in a
jump table near the end of memory called the 'Kernel'.* It is possible to deduce their
functions by looking through ROM for calls (i.e. JMP or JSR) to them; for example
FFE4 is GET. The kernel routines and their jump addresses for all ROMs are listed at
the end of Chapter 15; the entry points for BASIC keywords are noted in Chapter 5;
and the machine-code monitor and its subroutines are listed in BASIC 4 sequence,
starting at about D400 in the BASIC 4 column of Chapter 15.

      There are far too many routines to   cover exhaustively. Let's consider
GET and PRINT.

GET in BASIC is closely similar to GET#, which, however, calls a routine to set the
input file number. In effect, the device number is usually zero for GET, and may be
1 or 2 (tape), 3 (screen), or 4 or more with GET#. In any of these cases, JSR FFE4
fetches a single character into the Accumulator. If it is the null byte, no character is
assumed to have been found. GET also, of course, has an assignment routine, so that
GET X$ not only fetches a character, but causes X$ to be set up with length 1 to hold
the character. On disassembling FFE4, we find that ST is set zero and the device num-
ber is checked. This is a standard feature of CBM I/O: in OPEN on the next page,
we have LDA device number; BEQ or BNE tests for the keyboard (device #0). After
this, A is compared with #3; if equal, the device is the screen (device #3); if the
carry flag is clear, one of the cassettes (#1 or #2) is assumed; and if the carry flag is
set, the device number exceeds 3, and may be (for example) #4 (printer) or #8 (disk).
So, by controlling the device number, we can control the device which FFE4 gets its
character from. The pause loop, above, assumes the input device is the keyboard,
which is the usual default value. By disassembling, we can see that $AF holds the
device number ($0263 in BASIC 1); we can also find (for example) that the cursor's
position on the screen, and whether or not it flashes, are controllable when the screen
is used as an input device. Similarly, assuming a file is open to a device, we can get
characters in machine-code much faster than is possible with BASIC, at*least with disk
(since tape will be slow in any case). There is a password routine on the next page;
SYS 634 waits until a password has been entered, using FFE4 so it isn't echoed to the
screen. If it is wrongly entered, RESET is called. More sophisticated versions allow
several attempts to be made, and prompt the user with 'Enter password', but the

*Commodore literature, at the time of writing, seems to have adopted 'Kernal' as its
official spelling.

EXAMPLE OF ROM DISASSEMBLY: 'OPEN'

JSR inputs and stores logical file, device, and secondary address parameters.
LDA logical file number
BEQ print ?SYNTAX ERROR rejects logical file number 0.
LDY #offset for FILE OPEN message in table $F000 ff.
JSR checks table of up to 10 logical-file numbers for match with accumulator.
BEQ print ?FILE OPEN ERROR if the file number exists already.
LDX number of open files.
LDY #0
STY status byte ST. Sets ST to 0.
CPX #$0A
BEQ print ?TOO MANY FILES ERROR if X is 10 at present.
INC number of open files.
LDA logical file number.
STA File table,X. X still holds the previous value, i.e. 0-9.
LDA secondary address
ORA #$60 sets bits 5 & 6. Secondary addresses>95 repeat earlier values.
STA secondary address: stores the result.
STA Secondary address table,X. Also stores the result in the second table.
LDA device number
STA Device number table,X. Finally, stores the device in the third table.
BEQ RTS. If device number is 0, i.e. <u>keyboard</u>, file is now open.
CMP #3
BEQ RTS. If device number is 3, i.e. <u>screen</u>, file is now open.
BCC +3. Branch is taken if device is 1 or 2, i.e. cassette.
JMP sends name string to IEEE for device numbers > 3, <u>usually disk</u>.
    String is usually of form "d:filename,type,mode" and the receiving device
    processes it. If the device is present and answers, its file will be opened.
    <u>Tape</u>:
LDA secondary address
AND #0F removes bits 5 & 6 again, leaving secondary address=0,1, or 2.
BNE W1. Write tape if the branch is taken, that is if secondary address=1 or 2.
    <u>Read</u>:
JSR prints PRESS PLAY and waits for cassette key (unless a key's down now).
JSR prints SEARCHING and, if name has non-zero length, FOR FILENAME ...
LDA length of name
BEQ T1. If a name's not given, loads the first header.
JSR find a named header matches the first characters of the names
BNE T2. If the accumulator holds 0, the header wasn't found; in this case,
    print ?FILE NOT FOUND ERROR, abort files, and if in program mode
    print IN LINENUMBER with the current linenumber.
T1 JSR find first tape header (i.e. or next on tape)
    BEQ prints ?FILE NOT FOUND ERROR, as before, if no file can be found.
    BNE +8 unconditionally branches to T2 when the file <u>is</u> found.
W1 JSR prints PRESS PLAY & RECORD and OK on cassette key depression.
    LDA #4. Indicates the type of file (data).
    JSR writes tape header   <u>Write</u>.
T2   20 byte routine which sets the pointer to the cassette buffer to 0 for
    write, 191 for read. If writing to tape, puts #2 into the zeroth. byte
    of the buffer as a marker.

Old ROM BASIC 1 differs in some ways from this schema, which closely follows both BASIC 2 and BASIC 4. There is a rather misleading appearance that tape handling predominates in OPEN, because the IEEE processing is done elsewhere. OPEN is one of the commands from the 'kernel'; its address is $FFC0, from which the following addresses are jumped to:

ROM entry points:

       BASIC 1: $F52A (62762)
       BASIC 2: $F521 (62753)
       BASIC 4: $F560 (62816)

general idea should be clear enough
from the example:

The loop construction in
027E - 0282 is a standard wait-for-
entry, exactly analogous to BASIC's

100 GET X$: IF X$="" GOTO 100 ,

and usable because JSR FFE4 acts
like LDA, setting the zero flag when
a zero byte is loaded into A. The
'password' is 'AB'; in practice, a
longer routine can input longer
passwords; the resulting program

```
027A LDA #00    ;ERASE KEYBOARD BUFFER ...
027C STA 9E     ;0263 IN BASIC 1
027E JSR FFE4
0281 BEQ 027E   ;LOOP IF NO ENTRY
0283 CMP #41
0285 BNE 0291   ;RESET IF NOT CORRECT
0287 JSR FFE4
028A BEQ 0287
028C CMP #42
028E BNE 0291   ;RESET IF NOT CORRECT
0290 RTS
0291 JMP (FFFC)
```

will be more compact if it keeps the password in a table, and relies on indexing rather
than the straight-line style of programming here.

PRINT can be directed to any output device in a way analogous to GET. The kernel
corresponding to FFE4 is FFD2; on disassembling, this has a very similar device num-
ber check, but its location is $B0, not $AF, as this is the output device; its default
value is 3, since the usual output device is the screen. ($0264 is BASIC 1's location).
Operation is the opposite of GET: a value is loaded into A, the routine called, and the
ASCII equivalent printed. Chapter 15 has a lot of information about this. PRINT calls
this routine in a loop when a string is being printed to the screen; each character is
simply loaded, with the LDA (Zero-page),Y command, and individually output. This
method may be easier than relying on CA27/ CA1C/ BB1D in ROM

Demonstrations which use various ROM routines A short selection of programs which
use BASIC routines and demonstrate their operations follows. Most ROM routines refer
to standard zero-page and other locations; skilful use of ROM therefore usually in-
volves a certain amount of disassembly to investigate the most important parameters of
a routine, plus some searching for routines which will do as much work as possible
given minimum preliminary work. See for example POP and VARPTR in Chapter 5, each
of which largely relies on ROM routines, the first on RETURN, the second on LET. In
both cases the actual working of the ROM routines doesn't need to be understood fully.

(i) 'Receive line from keyboard'. This is an important routine, used, among
other things, to take in lines of BASIC and combine them into a program. To watch
it in operation, key in the 'Display bytes using IRQ' program (2 pages back) and set
it to display 80 bytes from $0200, which is the start of the input buffer. (BASIC 1's
buffer starts at $0A). The buffer is 81 bytes long; when a line is input, a zero byte
is put at the end, so 81 bytes are needed to store a line of 80 bytes maximum. In
the machine-language routine to show how
this works (right) I have separated out the
two functions of inputting a line and token-
ising it; a leading space causes tokenisation,
no leading space inputs the line without
changing it. Both processes can be watched
as they take place. (Note: use lower-case
mode, i.e. poke 59468,14, to ensure alpha-

```
033A JSR C468/ C46F/ B4E2; BASICS 1,2,4
033D LDA 0200
0340 CMP #20
0342 BEQ 0345
0344 RTS
0345 JMP C48D/ C495/ B4FB; BASICS 1,2,4
```

betic characters are readable). SYS 826 prints a flashing cursor, and awaits input;
when Return is pressed, the line is input, as you will be able to see. When the line
includes a leading space, you will also see the tokenisation process occurring.

(ii) 'Get linenumber from BASIC line'. To save space, we'll consider BASICs 2
and 4 only here. On exit from 'Receive line from keyboard', X and Y hold #FF and
#01. They are set to point to $01FF. We
can use these values with BASIC's GETCHR
routine as shown, with the 'Get linenumber'
routine. The effect of this is to store the
number in ($11); if no number is found, or
the number is zero, ($11) holds #0. To
prove it's worked, the final subroutine
prints the value of ($11), using another
ROM routine which prints 256*A + X.
Now SYS 826 awaits a line, and on Return

```
033A JSR C46F/ B4E2 ;GET BASIC LINE
033D STX 77
033F STY 78
0341 JSR 0070 ;GETCHR AFTER $01FF
0344 JSR C873/ B8F6 ;GET NUMERALS
0347 LDA 12
0349 LDX 11
034B JSR DCD9/ CF83 ;PRINT NUMBER
034E RTS
```

prints the value of its leading integer.

(iii) Search BASIC for linenumber.
GOTO, GOSUB, and other BASIC operations use the subroutine C522/ C52C/ B5A3 to find a specified linenumber. Confining ourselves to BASIC 2 and 4 (BASIC 1 uses different storage locations), this machine-code (right) hunts BASIC for the linenumber in (033A); if found, the carry flag is set on leaving the ROM routine, so this program uses a location to register whether or not a line actually exists in BASIC; this is 033E, which is zero when the line exists, but #FF otherwise. (033C) holds the pointer to the position of the line's start. Because both ($11) and ($5C) are liable to be overwritten, they are saved in these special locations, but machine-code programs usually will make use of the values without needing to do this. The short BASIC program which follows, entered at the end of any BASIC, lists all the linenumbers and their positions:

```
033A          ;LINENUMBER LO BYTE
033B          ;LINENUMBER HI BYTE
033C          ;POINTER TO LINE LO BYTE
033D          ;POINTER TO LINE HI BYTE
033E          ;LINE FOUND/ NOT FOUND FLAG
033F LDA 033A
0342 STA 11
0344 LDA 033B
0347 STA 12
0349 LDA #00
034B STA 033E
034E JSR C52C/ B5A3 ;BASIC 2 OR 4
0351 BCC 035E         ;NOT FOUND IF C=0
0353 LDA 5C
0355 STA 033C
0358 LDA 5D
035A STA 033D
035D RTS
035E DEC 033E ;FLAG = #FF IF NOT FOUND
0361 RTS
```

```
62500 FOR L=0 TO 65535: POKE 826,L-INT(L/256)*256: POKE 827,L/256
62510 SYS 831: IF PEEK(830)=0 THEN PRINT "LINE" L "STARTS AT" PEEK(828)
      + 256*PEEK(829): NEXT
```

(iv) RUN. This routine (at C775/ C785/ B808) has four lines of machine-code only; one jumps to RUN, two jump to RUN n where n represents a linenumber, and one is a branch to separate the two, depending on whether RUN is an isolated keyword or is followed by a number. The machine-code program (right) shows how the option which runs from a linenumber can be mimicked with the linenumber search program. Note that RUN assumes CHRGET points to the zero byte preceding a line: for this reason it is necessary to subtract #1 from the pointer ($5C), which the program does by adding #FFFE plus the carry flag to (5C) and storing the result in (77).

```
          ;SET ($11) = LINENUMBER
JSR C52C/ B5A3 ;BASIC 2 OR 4
BCC EXIT        ;NO SUCH LINE
LDA 5C
ADC #FE
STA 77
LDA 5D
ADC #FF
STA 78
JMP C78A/ B80D ;RUN FROM LINENUMBER
```

(v) Memory move. A routine at C2E1/ C2DF/ B357 is one of several in BASIC ROM which move blocks of memory; this one is used to open space in BASIC, so ROM calls which make use of it move memory from a lower to a higher point. The demonstration routine fills the screen (40 columns - $8000-$83E7) with bytes from memory, starting from the location stored in (033A). ($5C) holds this value; ($57) has to hold the top of the area to be moved + 1, so this is calculated by adding #03E8. Finally ($55) holds the top of the area to be moved to + 1; this is $83E8 in the example. This can of course be modified to move other blocks of RAM than those of length #03E8 bytes, and into other locations than the screen top, for example by poking values from BASIC.

```
033A      ;BOTTOM OF AREA TO BE MOVED
033B      ;
033C CLC
033D LDA 033A
0340 STA 5C
0342 ADC #E8
0344 STA 57
0346 LDA 033B
0349 STA 5D
034B ADC #03
034D STA 58
034F LDA #E8
0351 STA 55
0353 LDA #83
0355 STA 56
0357 JMP C2DF/ B357 ;BASIC 2 OR 4
```

Memory move routines aren't as simple as they might appear at first sight: the problem occurs if a region to be moved overlaps with the region it is to be moved to. Suppose the first four bytes of ABCDEF are to be moved to the region now occupied by CDEF. If the memory-move operates by taking bytes from the left and working to the right, the result will be ABAB, not ABCD. The order, here, should be right to

left. The short BASIC routine is a bug-free example of what is needed. If you are not aware of this potential problem, baffling errors may result.

```
993 REM *************************************************************
994 REM * MEMORY MOVE IN BASIC.                                     *
995 REM * PARAMETERS: SA=START ADDRESS OF BLOCK TO BE MOVED         *
996 REM *             EA=  END ADDRESS OF BLOCK TO BE MOVED         *
997 REM *             TA=  START ADDRESS OF BLOCK TO BE MOVED TO     *
998 REM *************************************************************
999 REM
1000 IF TA)SA AND EA)=TA THEN 1020
1010 C=0: FOR I = SA TO EA: POKE TA+C,PEEK (I): C=C+1: NEXT: RETURN
1020 C=EA-SA: FOR I = EA TO SA STEP -1: POKE TA+C,PEEK (I): C=C-1: NEXT:RETURN
```

(vi) <u>String comparison.</u> CF1E/ CF1O/ C0CE is the start of the string comparison subroutine; its parameters are listed in Chapter 15. Like many ROM routines - and this feature serves as a bridge to the next section of this chapter - the relevant part of the routine is embedded in other processes, not separated out as its own subroutine. For this reason, the routine must be relocated into RAM and modified there. If the central part of 'String comparison', i.e. after the parameter-setting code and up to the exit from the comparison loop, is relocated, it may be followed by RTS and used in isolation. In this way, it is possible to confirm that X is set to #FF/ #0/ #1 according as the first string is </=/> the second. SORT (Chapter 5) relies on the relocated comparison routine for its operation.

13.4 Examples of modified ROM routines.

13.4.1 PRINT USING. The number formatting buffer, in which numerals are placed for printing after conversion from the floating-point accumulator or elsewhere, extends roughly from $00FF to $010F. 'Display bytes using IRQ' can show how this buffer is used in practice, and the table below gives specimen results from a variety of number outputs, including string conversions, calculations, and simple print statements. The notes on PRINT USING in Chapter 5 explain how the normal number printing routine, which is made up of three consecutive subroutines, can be copied, but with the addition of an extra routine to alter the buffer from the way it appears in the diagram to a new, formatted and justified arrangement.

EXAMPLES OF NUMBER FORMATTING

WHEN PRINTING:	OFF	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F
0		sp	0	@	Unchanged	....											
45		sp	4	5	@	0	0	0	0	0	0	0	Unchanged ...				
24.1234		sp	2	4	.	1	2	3	4	@	0	0	Unchanged ...				
999 999 999.1		sp	9	9	9	9	9	9	9	9	9	@	Unchanged ...				
9 999 999 999		sp	1	E	+	1	0	@	0	0	0	0	Unchanged ...				
2↑12.1		sp	4	3	8	9	.	9	8	4	1	1	@	Unchanged	...		
LOG(.5)		-	.	6	9	3	1	4	7	1	8	@	Unchanged ...				
ASC("T")		sp	8	4	@	0	0	0	0	0	0	0	Unchanged ...				
TI*		sp	2	1	6	6	8	9	1	@	0	0	Unchanged ...				
TI$*	1	0	0	3	4	4	@	Unchanged	....								
STR$(44)	sp	4	4	@	0	0	0	0	0	0	0	Unchanged ...					
STR$(1/3)	sp	.	3	3	3	3	3	3	3	3	3	@	Unchanged ...				
STR$(-pi)	-	3	.	1	4	1	5	9	2	6	5	@	Unchanged ...				
.009		sp	9	E	-	0	3	@	0	0	0	0	Unchanged ...				
-1E-23		-	1	E	-	2	3	@	0	0	0	0	Unchanged ...				
12345678.9		sp	1	2	3	4	5	6	7	8	.	9	@	Unchanged	...		
123456789		sp	1	2	3	4	5	6	7	8	9	@	Unchanged ...				
SIN(pi)		sp	7	.	3	1	4	5	9	0	4	E	-	1	0	@	
11 111 111 111		sp	1	.	1	1	1	1	1	1	1	1	E	+	1	0	@
.8		sp	.	8	@	0	0	0	0	0	0	0	Unchanged ...				

*These values are specimens only
'@' represents the null character

PRINT USING is too long to explain fully here; the buffer-editing routine formats the data in either floating-point or integer form, by inserting or dropping the decimal point; it also checks for the presence of E and the null byte, among others, and has exits which it takes if a number appears to be of the incorrect form. It makes no attempt to format a number containing 'E', for instance. On leaving the routine, ($61) is made to point to $0100, the start of the buffer, from which point the string-print routine takes characters, printing them until the null byte is encountered.

13.4.2 LIST.

Chapter 5 has a BASIC lister which generates program listings in which the cursor control characters appear as [DOWN],[RVS], etc. Apart from the problems of lower-case listings with CBM printers, this is probably the most wished-for feature of LIST. It is fairly easy to implement, once LIST has been moved from ROM to RAM, because LIST is a compact routine and a patch is quite easy to put in. A BASIC 2 version (see below), called by SYS 7*4096 or SYS 28672, lists in exactly the same way as the ordinary LIST, except for the special characters; these are held, with their equivalent output, in two tables after the machine-code. This sort of routine must have a control over the length of a printed line, since any line with many special characters in it will be printed out to a length which may cause some characters to be lost. The annotated BASIC 4 version shows how a BASIC loader can improve a program's versatility; it relocates its code into any BASIC 4 machine, and allows easy changing of the target characters to be specially listed. It too has a linelength control; a short specimen shows the sort of effect which can be achieved.

B*

```
       PC   IRQ  SR AC XR YR SP
.;    0401  E62E  32 04 5E 00 F8
.:
.:    7000  A9 01 85 5C A9 04 85 5D
.:    7008  A9 FF 85 11 85 12 A0 01
.:    7010  84 09 B1 5C F0 43 20 E1
.:    7018  FF 20 B0 70 C8 B1 5C AA
.:    7020  C8 B1 5C C5 12 D0 04 E4
.:    7028  11 F0 02 B0 2C 84 46 20
.:    7030  D9 DC A9 20 A4 46 29 7F
.:    7038  20 BA 70 C9 22 D0 06 A5
.:    7040  09 49 FF 85 09 C8 F0 11
.:    7048  B1 5C D0 10 A8 B1 5C AA
.:    7050  C8 B1 5C 86 5C 85 5D D0
.:    7058  B5 4C 89 C3 10 28 C9 FF
.:    7060  F0 D6 24 09 30 20 38 E9
.:    7068  7F AA 84 46 A0 FF CA F0
.:    7070  08 C8 B9 92 C0 10 FA 30
.:    7078  F5 C8 B9 92 C0 30 B5 20
.:    7080  BA 70 D0 F5 53 00 8E 85
.:    7088  70 A2 08 DD E0 70 F0 09
.:    7090  CA 10 F8 AE 85 70 4C 38
.:    7098  70 8A 0A 0A 0A AA BD F0 70
.:    70A0  70 F0 06 20 BA 70 E8 D0
.:    70A8  F5 AE 85 70 4C 3B 70 00
.:    70B0  20 E2 C9 A9 06 8D 84 70
.:    70B8  60 EA 48 20 45 CA EE 84
.:    70C0  70 AD 84 70 C9 5A F0 02
.:    70C8  68 60 20 B0 70 20 CA FD
.:    70D0  20 CA FD 20 CA FD 68 60
.:    70D8  00 00 00 00 00 00 00 00
.:    70E0  11 12 13 1D 91 92 93 9D
.:    70E8  00 00 00 00 00 00 00 00
.:    70F0  5B 44 4F 57 4E 5D 00 00
.:    70F8  5B 52 45 56 53 5D 00 00
.:    7100  5B 48 4F 4D 45 5D 00 00
.:    7108  5B 52 49 47 48 54 5D 00
.:    7110  5B 55 50 5D 00 00 00 00
.:    7118  5B 52 56 53 4F 5D 00 00
.:    7120  5B 43 4C 45 41 52 5D 00
.:    7128  5B 4C 45 46 54 5D 00 00
.:    7130  00 00 00 00 00 00 00 00
.:    7138  00 00 00 00 00 00 00 00
```

(handwritten annotations on listing: "08", "E0 70", "F0 70", "5A F0", "Controls Length of Line = 90 characters. (For a 20-column List only, change to 20 = $14).")

MACHINE-CODE LIST FOR BASIC 2.

SYS 7*4096 runs this routine, which lists the entire BASIC program in memory, starting at $0401. It processes characters within quotes. There are (here) eight characters which are specially dealt with; they are stored in a table starting at $70E0. The corresponding output is held in another table, starting at $70F0. So character $11 (17 decimal), the first item in the first table, appears as [DOWN], which is the first item in the second table.

A zero byte has been used to terminate printing, so an output item can have a maximum length of 7 only.

OPEN 4,4:CMD4:SYS7*4096

then PRINT#4:CLOSE4 lists to a printer.

This routine is suitable for a 32K machine; see chapters 13 and 14 for information on relocating machine code.

RELOCATING LOADER FOR SPECIAL 'LIST' – BASIC 4.0

```
10 DATA 169,0,133,48,133,50,133,52,169,0,133,49,133,51,133,53,96
11 DATA 169,1,133,92,169,4,133,93:REM 1) LOWER MEMORY, 2) SET LINENUMBERS
12 DATA 142,-350,162,8,221,-259,240,9,202,16,248,174,-350,76,-440,138,10
13 DATA 10,10,170,189,-243,240,6,32,-297,232,208,245,174,-350,76,-437
14 DATA 0,32,223,186,169,6,141,-351,96,0,72,32,70,187,238,-351,173,-351,201
15 DATA 90,240,2,104,96,32,-307,169,6,141,56,125,169,32,32,210
16 DATA 255,206,56,125,208,246,104,96
1000 REM
1001 REM            ******************************************
1002 REM            ** USER-DEFINABLE LIST FOR BASIC 4.0  **
1003 REM            ******************************************
1004 REM
1050 T = PEEK(52) + 256*PEEK(53) : REM CURRENT TOP OF MEMORY
1060 L = T - 513                 : REM ASSIGN 513 BYTES
1100 REM
1101 REM ******************************************************************
1102 REM *       BASIC 4.0 MEMORY-MOVE OF 'LIST' ROUTINE INTO RAM.       *
1103 REM *       $B657 - $B6DD   (46679-46813)                          *
1104 REM ******************************************************************
1105 REM
1110 X = 0
1120 FOR J = L + 25  TO  L + 159
1130 POKE J, PEEK (X+46679)
1140 X = X+1 : NEXT
1160 FOR J = L TO L + 24: READ X : POKE J,X: NEXT  :REM STARTUP ROUTINES
1170 FOR J = L + 164  TO  L + 251                  :REM PROCESSING ROUTIN
1180 READ X%: IF X%<0 THEN Y=X%+T: X%=Y/256: Z=Y-X%*256: POKE J,Z: J=J+1
1190 POKE J , X%
1200 NEXT
1250 REM
1251 REM ****************************************************************
1252 REM * CHANGE BRANCH AND JSR COMMANDS IN 'LIST' TO RUN OWN ROUTINES *
1253 REM ****************************************************************
1254 REM
1260 REM * SEARCH BYTE TABLE ROUTINE
1261 REM
1265 POKE L+110, 53: REM CHANGE BRANCH DESTINATION (VALUES 00-7F)
1266 POKE L+114, 49: REM CHANGE BRANCH DESTINATION (VALUE FF =PI)
1267 POKE L+118, 45: REM CHANGE BRANCH DESTINATION (VALUES 80-FE)
1269 REM
1270 REM * CRLF ROUTINE
1271 REM
1275 X = T-307     : REM CRLF POSITION RELATIVE TO TOP OF MEMORY
1276 X% = X/256    : Z = X - X%*256
1277 POKE L+43, Z : POKE L+44, X%
1279 REM
1280 REM * PRINT BYTE ROUTINE
1281 REM
1285 X = T-297     : REM PRINT CHARACTER ROUTINE RELATIVE TO MEMORY TOP
1286 X% = X/256    : Z = X - X%*256
1287 POKE L+74, Z : POKE L+75, X%
1288 POKE L+156, Z: POKE L+157, X%
1300 REM
1301 REM ****************************************************************
1302 REM * PUT NEW, LOWER TOP OF MEMORY INTO MEMORY POINTER ROUTINE *
1303 REM ****************************************************************
1304 REM
1305 X% = L/256    : Z = L - X%*256
1310 POKE L + 1 , Z : POKE L + 9, X%
1400 REM
1401 REM ****************************************************************
1402 REM *   USER'S TABLES OF BYTES AND OUTPUT (EG 157 PRINTED [LEFT])  *
1403 REM *   CAN BE CHANGED SUBJECT TO MAXIMUM OF 16 ITEMS OF LENGTH 7  *
1404 REM ****************************************************************
```

```
1408 DATA TABLES,10,17,18,19,29,145,146,147,157,160,255: REM 10 =ITEMS IN TABL
     E
1410 DATA [DOWN],[RVS],[HOME],[RIGHT],[UP],[RVSO],[CLR],[LEFT],[USPC],[PI]
1420 FOR J = 1 TO 1E9 : READ X$ : IF X$ () "TABLES" THEN NEXT
1430 READ X%                    : REM NUMBER OF ITEMS IN THE TABLE
1440 POKE L + 168 , X% - 1      : REM SETS LOOP IN M/CODE FROM 0 TO X%-1
1450 FOR J = L + 254 TO L + 253 + X%
1460 READ X : POKE J,X : NEXT : REM BUILD TABLE OF BYTES IN MEMORY
1470 FOR J = L + 269 TO L + 268 + X%*8 STEP 8
1480 READ X$: FOR X = 1 TO LEN (X$) : POKE J+X, ASC(MID$(X$,X,1)) : NEXT
1490 POKE J + X , 0: NEXT     : REM POKE NULL TERMINATING BYTE AFTER EACH
1500 REM
1501 REM ****************************************************************
1502 REM *               PRINT INSTRUCTIONS AND ADDRESSES            *
1503 REM ****************************************************************
1504 REM
1510 PRINT "[CLR][RVS][DOWN][DOWN] ROM4 LIST BY RAY WEST "
1520 PRINT "[DOWN][RVS]LIST    SYS,[LEFT]"; L+17
1530 PRINT "[DOWN]POKE"; L+227 ; "TO CHANGE LINELENGTH"
1540 PRINT "[DOWN][DOWN]SAVE FROM" ; L ; " TO" ; L+512
1550 PRINT "             ($" ;; X=L; GOSUB 5000
1560 PRINT " TO $" ;; L=X+512; GOSUB 5000 ; PRINT")"
1570 PRINT"[DOWN]SECURE IN MEMORY WITH SYS" ; X ;PRINT "WHICH LOWERS TOP-OF-ME
     MORY
1580 PRINT"[DOWN][DOWN]";; LIST 1410
4900 END
4999 REM   DECIMAL TO HEX CONVERSION
5000 L=L/4096:FORJ=1TO4:L%=L:PRINTCHR$(48+L%-(L%)9)*7);;L=16*(L-L%):NEXT:RETUR
     N
```

```
10 DATA 169,0,133,48
   ,133,50,133,52,169
   ,0,133,49,133,51,1
   33,53,96
11 DATA 169,1,133,92
   ,169,4,133,93:REM
   1) LOWER MEMORY, 2
   ) SET LINENUMBERS
12 DATA 142,-350,162
   ,8,221,-259,240,9,
   202,16,248,174,-35
   0,76,-440,138,10
```

SAMPLE SHOWING EFFECT OF CHANGING LINELENGTH (=24) →

13.4.3 TRACE. The TRACE routine in Chapter 5 is another application of LIST, in this case written to display single lines at the screen-top. It operates by intercepting the GETCHR routine; this is a standard technique, explained in full detail in the next Chapter. Its controlling keys are determined by the keyboard decoding table, and apply to BASICs 1 and 2 and the 40-column BASIC 4. There is insufficient space to include two more versions, so BASICs 1 and 2 only appear in TRACE in Chapter 5. The flowchart of the routine (see next page) shows how CHRGET has intermediate instructions; this version of 'trace' does not distinguish between statements, but by linenumbers, so that a line is listed for as long as it is being executed. The trace is turned on by pressing RVS, and off again by pressing the same key, so keyboard control of the routine is good. A,X, and Y are saved and restored, so CHRGET's running isn't disturbed. The status flags are set at a later stage, and need not be stored. Note that a single-step is available; when the trace is on, '=' causes a BASIC line to be executed, and no further lines are executed until either '=' is pressed again, or the routine is reset by the speed change command, '[' followed by a number from 0 - 9. Lines are executed every 5 seconds with 0, down to every .5 second when 9 is chosen. This, however, is overridden by the space key, which removes the action of the delay loop, and traces through the program very rapidly. LIST is not very easy to incorporate in programs like this, because it uses many zero-page locations which are normally allocated for other purposes. This is probably the reason why LIST is not allowed in program mode without terminating the program run. My version of trace

sidesteps this problem by saving the zero-page, and restoring it when LIST has done its work. This approach also enables the horizontal and vertical screen positions to be retained even after the cursor is homed and the top of the screen cleared for LIST. There is insufficient space for full documentation or explanation here. However, the memory map of the routine is arranged like this:

       (i) Initialisation routine to alter CHRGET and lower memory pointers.

      (ii) Switch-off routine to restore CHRGET.

    (iii) Space for 9 bytes. These are: A,X, and Y storage; Trace flag (#0 off, #FF on); Step flag (#0 off, #7F on); current linenumber, stored as low byte then high byte; delay parameter; countdown for delay loop, starting from delay parameter value.

    (iv) Routine as in the flowchart, excluding ...

     (v) LIST routine, slightly modified (e.g. not to print crlf at the start).

    (vi) 256 spare bytes for the stored zero-pages.
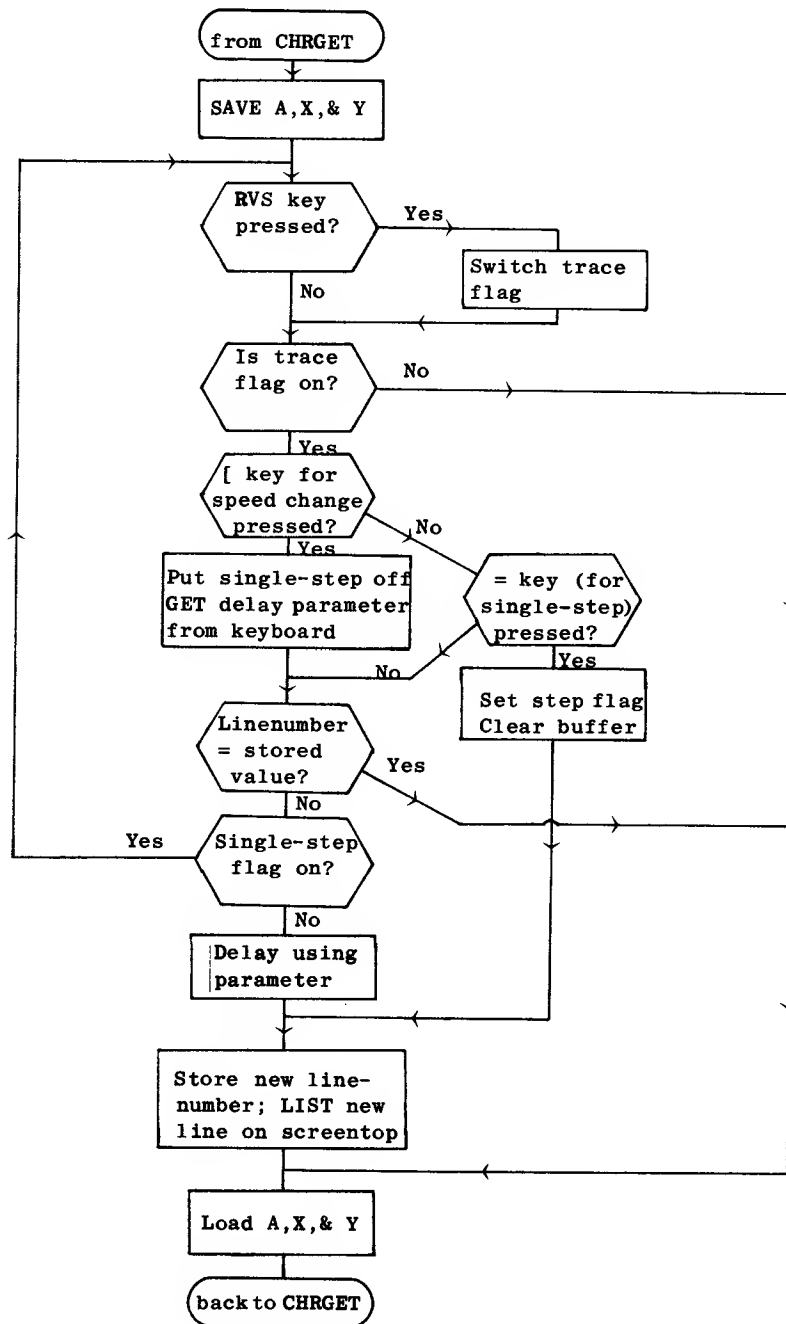
CHAPTER 14: EFFECTIVE 6502 PROGRAMMING

14.1 Assemblers.

We have seen the improvements in machine-code readability brought about by the use of opcodes, and by the use of hexadecimal notation, over the fundamental 8-bit binary storage of this system. An *assembler* carries this improvement much further, by allowing a fully algebraic notation to represent machine-code. Before going into detail, let's consider what is involved. A diagram on the next page illustrates features found in most assembler listings, without representing any one actual assembler's output. Note that the *object code* is identical to that produced by a simple disassembler. This is not surprising, since the same fundamental data underlies both approaches and enforces some uniformity. The main novelty is the so-called *source code*. Around the core of familiar opcodes is a collection of names and symbols, some of which (e.g. LDA ADRTAB,X) are punctuated to resemble addressing modes. This code is usually held as a *source file*, which is in effect a sequential file of everything labelled 'source code' on the diagram. The job of the assembler is to convert the file into object code, an example of which appears on the same diagram. Object code is not usually relocatable without some effort; it is designed to be run where it is, from $2000 in the example. The great versatility of assemblers is illustrated by the command in linenumber 12 of the code, in which the starting-point of the assembly is assigned as $2000. A simple change to *=$3000, followed by assembly, generates code identical in its effect, but positioned to start at $3000. In the same way, new instructions can be inserted by editing the source file and reassembling; this is far less laborious than attempting the same process manually. Note, though, that a source file may be much longer than the machine-code it generates: compare the number of bytes in a typical assembler line with the total length of the line. A ratio of between 20:1 and 50:1 is common. This means that long programs of 1K or more are already difficult to fit in RAM and have to be separated into sections. Conversely, disassemblers which generate labels, producing a version of source code without comments but with assembler format, are likely to be unable to cope with long programs, for instance the BASIC ROM. Some widely-available software for the CBM was in fact developed on other machines.

    CBM's assembler (issued some years back originally; try to get the latest version including the source files LOADERSRC, ED16SRC and ED32SRC, and U-LD16SRC and U-LD32SRC, which have useful documentation) is widely used and has these features: it is disk based,* designed for use with CBM disk units and printer, and without EPROMs or other hardware, so there are potential conflicts between assembled code and the set of programs constituting the assembler package. Everything is in RAM, as this diagram shows:

ASSEMBLER		EDITOR		
	Normal BASIC storage (for 32K)	Screen		BASIC ROM

$0000   ←——— LOADERS ———→ $FFFF
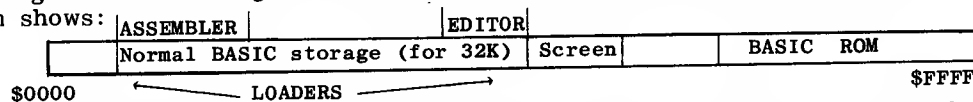
There are three essentially different programs in the package: an editor, an assembler, and several loaders. The editor is pictured at the high end of RAM. It actually loads into the low end and relocates on being run, so it can coexist with the assembler, if it (the editor) is loaded and run first. The function of the editor is to enable the user to set up a source file; the line numbered format is similar to BASIC, lines being typed in and modified from the keyboard, and saved to disk and perhaps printed out. To facilitate this process, line renumbering, block deletion, automatic numbering and so on are provided. Files can be loaded from disk, edited, and stored back on disk, to correct errors which have shown up. PUT and GET are the commands to save and load, respectively. (There may be an undocumented CPUT which stores the file without surplus spaces). All this is very standard in editors.

    The assembler reads a source file into RAM (or uses a source file already there, starting at $2000) and proceeds to convert it into object code, which may be stored directly into RAM or saved on disk as an object file. The data saved is formatted in a manner similar to a program file, with the length of the routine followed by its starting address, and then all the data bytes written as hexadecimal numbers.

    The loaders read the object file just described, and poke its contents into RAM. If the machine-code program is being loaded high in RAM, the low-memory loader is the

---

*I'm uncertain whether it's impossible, or merely difficult, to use cassette tape.

LINE-NUMBER	AD-DRESS	OBJECT CODE	[LABEL/ OPCODE or DIRECTIVE/ OPERAND/ COMMENT] ---------SOURCE CODE------------------------		
2			;		
4			;ROUTINE TO AWAIT A KEY, THEN EXECUTE CORRES-		
6			;PONDING CODE, USING TABLED VALUES		
8			;		
10			GETCHR=$FFE4		;TYPICAL 'EQUATES' DIRECTIVE
12	2000		*=$2000		;TYPICAL STARTING-POINT DIRECTIVE
14	2000		.PAGE		;TYPICAL TOP-OF-FORM DIRECTIVE
16	2000		;		
18	2000	20 E4 FF	START	JSR GETCHR	;STANDARD KERNEL 'GET' INTO ACC'R
20	2003	F0 FB		BEQ START	; WAIT UNTIL KEY PRESSED
22	2005	A2 02		LDX #2	;TABLE HAS THREE VALUES ONLY
24	2007	DD 21 20	LOOP	CMP CHRLIS,X	;COMPARE VALUES IN TURN,
26	200A	F0 05		BEQ FOUND	; UNTIL FOUND OR NOT FOUND
28	200C	CA		DEX	
30	200D	10 F8		BPL LOOP	;LOOP FROM X=2 TO X=0 INCLUSIVE
32	200F	30 EF		BMI START	;KEY NOT IN TABLE; GOTO START
34	2011	8D 20 20	FOUND	STA STORCH	;STORE THE ASCII CHARACTER
36	2014	8A		TXA	;STANDARD JUMP ROUTINE FOLLOWS,
38	2015	0A		ASL A	;IN WHICH THE STACK HOLDS BOTH
40	2016	AA		TAX	;BYTES OF THE DESTINATION, AND
42	2017	BD 25 20		LDA ADRTAB+1,X	;RTS CAUSES THE JUMP.
44	201A	48		PHA	; HIGH BYTE ON STACK ...
46	201B	BD 24 20		LDA ADRTAB,X	
48	201E	48		PHA	; AND LOW BYTE.
50	201F	60		RTS	;JUMP TO ADDRESS NOW ON STACK
52	2020		STORCH	*=*+1	;USES ASSEMBLER LOCATION POINTER
53	2021	41 42 43	CHRLIS	.BYTE 'ABC'	;SETS UP TABLE OF ASCII BYTES
54	2024	29 20 6F	ADRTAB	.WORD A-1, B-1, C-1;SETS UP TABLE OF ADDRESSES - 1		
56	202A	A0 00	A	LDY #0	;START OF PROCESSING FOR ROUTINE A

Features of a typical assembler listing.

one to use, while code destined for low RAM needs the high-memory loader. We may summarise the process of writing machine-code with an assembler in three stages:

   (i) The editor creates a source-file, usually stored on disk.

   (ii) The assembler creates an object-file from the source file, again on disk.

   (iii) A loader puts the program in memory; from here it can be saved as an ordinary machine-code routine by SAVE or .M .

It is not necessary to store the intermediate stages; it is merely advisable. Accidental loss of a long program, usually when incompletely debugged machine-code executes, is common.

By way of contrast, assemblers may be available in hardware. An EPROM called 'Mikro' illustrates this. It modifies BASIC to include its own instructions, which include a command to assemble; and it uses BASIC's line input facility as an editing system.

Most assemblers use the 'two pass' system, in which forward addresses are calculated on a second pass. To understand why this is necessary, imagine that you are assembling the example above, and have arrived at line 24. Its address has not yet been reached, so its value (i.e. the value of CHRLIS) is temporarily left unfilled. If there is no such operand the assembler prints an error message (or should do), which probably will be one of many. First-time assemblies without any errors are rare. Unlike BASIC, which can run fairly successfully with syntax errors dotted about, assembler is intolerant of errors in its source file. Often, removing errors becomes a sub-goal in itself, the triumph of finally achieving no 'errors' leading the programmer to fail to notice that the resulting program doesn't quite do what it should.

Assemblers vary in the way they scan source code. Some assume fairly strict column formatting, and may *require* source code to be arranged tidily; others don't check, beyond expecting one or more spaces as separators. For this reason, line 18

is rejected by some assemblers, because **START** seems to contain the opcode **STA**.

Assembler features. Assemblers for the 6502 and other chips typically have features like those listed below, most of which make an appearance in the specimen listing:

(i) **Labels.** These mark entry-points to which branches, jumps or subroutine calls are made. Often there is a maximum length of six characters.

(ii) **Opcodes.** Invariably standard.

(iii) **Directives.** These commands have a similar effect to opcodes, carrying out a single fairly simple function. Often they are preceded by a '.', which is picked up by the assembler's parsing; sometimes they resemble opcodes, having three-letter mnemonics. The most important directives are probably these:

* or ORG. This sets the starting point or origin form which the object code is to be assembled. * has a further use, as a location pointer, illustrated in line 52 of the specimen, where it reserves 1 byte as a storage location. Similarly, *=*+50 reserves 50 bytes, and .BYTE *-LABEL calculates the difference between the current location and an earlier label, storing it in one byte.

= or EQU. The 'equates' directive is self-explanatory. Most directives of this type are collected at the start of source code, where they can be both easily seen and checked, and easily altered.

.BYTE .TEXT BYT TXT. These provide variations on a theme; in some assemblers the functions are combined in .BYTE, in others .BYTE and .TEXT differ.
.BYTE enters single bytes into RAM, as these examples show:
.BYTE 31,$EA,%0100 0001   puts three bytes 1F EA 41 into RAM,
.BYTE COUNT + 1   puts the value of COUNT + 1 into RAM; e.g. if COUNT = 4, the hexadecimal number 05 is stored in RAM.
.BYTE 'HELLO' and .TEXT 'HELLO' are alternative forms of the function which stores five bytes (48 45 4C 4C 4F) in RAM, and generally sets up ASCII tables.

.WORD takes a 16-bit number, storing it in RAM with the low byte first and high byte second. The 'word-length' of a computer is usually the minimum length it is designed to handle, e.g. 4 bytes in IBM machines, 2 bytes in DEC. In 8-bit machines, 'word' and 'byte' are sometimes used synonymously, but assembler convention assigns 2-byte words to the 6502. Line 54 in the specimen listing has an example. Note that A = $202A, so A-1 = $2029, and this is stored, with bytes reversed, at the start of the 6-byte table labelled 'ADRTAB', some of which is omitted by the listing.

.DBYTE assigns two consecutive bytes to memory in the normal order. Since this can be accomplished in any case with .BYTE, this directive may be absent.

.END marks the end of the source code.

(iv) **Symbols.** The names (GETCHR, CHRLIS, LOOP, or whatever), including labels, are called 'symbols'. An assembler usually constructs a 'symbol table' on its first pass, filling in forward references from it on the second pass.

(v) **Operands.** The 'operand' following an opcode is a symbol or absolute value punctuated in the standard way, i.e. possibly including $, %, @ to signify hex, octal and binary numbers, # to signify immediate mode, ' for ASCII values, and ( ) , X Y.

(vi) **Comments.** Often signalled by a semicolon, which causes the assembler's parser to ignore the remainder of the line, these, like BASIC REMs, help make a program readable.

The CBM assembler. This has .PAGE and .SKIP which turn to a new page and skip a line (for easier reading) respectively. .OPT allows control over the printing of information after assembly. .OPT NOLIST, NOERRORS, NOMEMORY, NOSYMBOLS, NOGENERATE causes *only* assembler error messages to appear on the screen, with no other output either to screen or printer. CBM's assembler, like some others, allows an operand syntax including < and >, to load low and high bytes of 16-bit addresses. If .WORD M has set up the two bytes corresponding to the value of the symbol M, then LDA #>M /LDY #<M loads A and Y with the high and low bytes.

Assembly can be stopped with the Stop key; the program then awaits entry of M or B, after which the monitor or BASIC is entered. Conditional assembly, where values set at assembly-time modify the assembler output, is valuable in producing slightly different versions of similar routines, for example to fit different capacities of RAM. Examples occur in the source code (q.v.). Other special features include .LIB and .FILE directives. .LIB permits library software to be used, which means that a named file can be read and assembled into another source file during the course of its assembly. .FILE transfers assembly to another file, so a string of segments of code can be assembled one after the other. Finally, let's look at an assembler function which

cannot quite be carried out with this assembler. A *macro* is a directive which causes an assembler to expand a pseudo-command into several machine-code commands, which carry out the functions of the pseudo-command. An example might be MACRO INC ADDR, which would increment the 16-bit ADDR, by inserting INC #<ADDR/ BNE * + 4/ INC #>ADDR at the point in the source code where the macro instruction appeared. The point is that parameters are allowed, so MACRO INC OTHER would be expanded in a similar way, but with values appropriate to the address of OTHER. CBM's .LIB files don't work in quite this way, as there is no scope for varying parameters easily. But library software is not very different from macro generating software. Consider this example, which reverses the low and high nybbles in the accumulator.

The code works for any value, and leaves the processor status flags unchanged, even where this doesn't matter, so it can be inserted in any position where the operation of exchanging nybbles is wanted. The directive .FIL will do this, and provided that the filename is reasonably meaningful, the resulting source code should be improved in readability.

    (At first sight, four rotate instructions might be expected to reverse the nybbles, but, since the carry flag is included in the rotation, this is not the case. An extra bit would end up between two halves of the original byte. The code (right) gets round this by setting the carry flag at each stage to equal the current rightmost bit).

```
PHP
PHA
ROR A
PLA
ROR A
PHA
ROR A
PLA
ROR A
PHA
ROR A
PLA
ROR A
PHA
ROR A
PLA
ROR A
PLP
```

14.2 Conversion of machine-language programs between ROMs.

All PETs/ CBMs and VIC are sufficiently similar for machine-code interconversion to be likely to succeed. Generally, the later versions of BASIC include earlier ones' features as subsets, so upward conversion tends to be fairly easy, while downward conversion may not be possible. The steps are as follows: first, the program is disassembled. All jumps and subroutine calls to ROM are likely to be different in other ROMs, with the exception of kernel routines. Some of these calls may be made in disguised form by pushing bytes on the stack and executing RTS or RTI, and if they are, these too must be changed, although as a rule this method is used to jump to tables within the routine itself. Zero-page values, and those for the region $200 - $400, may have different functions; this is particularly the case with BASIC 1 as against BASICs 2 & 4, and again with VIC. Without an intelligent disassembler to create labels, it can be very tedious indeed to convert 2-byte instructions into three bytes. Other difficulties include special values, for example of keyboard decode tables, which vary (sometimes) between machines, so that LDA E812 for example is interpreted differently. ROM routines may have slightly different effects in their various ROMs. Thus, the 'print string' routine at CA27/ CA1C/ BB1D is different in BASIC 4. Chapter 15 has details of all PET/CBM ROMs to date, and many standard entry points are listed. Intermediate entry points can usually be found with some detective work on disassembled listings of the relevant parts of each ROM.

    Some interconversion problems are obviously insoluble: VIC colour programs can't be imitated on PETs, CRT controller chips can't be programmed in 8-inch screen machines, disk commands aren't available on BASIC 2 (unless they're in RAM), an ESCAPE key may not exist on the keyboard, BASIC 1 cannot be used with NMI. In cases of this sort parts of a program will have to be modified if the original workings are to be retained; otherwise, slightly different keyboard instructions and output formats from the original will result. It is impossible to lay down firm rules, since programming methods and styles vary, so that while many routines are trivially easy to convert, a twist in the method used makes another far more difficult.

    Machine-independent routines (as regards the PET/ CBM and perhaps VIC, but almost certainly not other machines too) can be written, or at least approached, by taking advantage of standard features. Thus, LOAD is likely to have a similar effect on all these machines, so if non-kernel ROM routines are stored in RAM the resulting code will be independent of ROM. Some of BASIC 2, for instance, may be transplanted into other BASICs, where its performance will be known. The same applies to some calculation and string routines. Apart from this, kernel routines for input, output, testing Stop, and so on, can obviously be used until such time as Commodore decides to change this aspect of ROM. It must be said that many important aspects of programs are difficult to move between machines; 22-, 40-, and 80-column screens

present, between them, a considerable challenge to the would-be programmer of machine independent code. In cases like this, the easiest course may be simply to have several programs which run independently; an identifying ROM feature, tested at the start of execution, causes the appropriate program of the several co-resident programs to run. Alternatively, an approach less wasteful of space is to collect the divergent parts of the program into subroutines with ROM version tests.

14.3 Coexistence of machine-code with BASIC: CHRGET, wedges, and utilities.

14.3.1 The CHRGET routine.
BASIC is almost always held in RAM; during execution, further RAM pointers are necessary to keep track of the stage reached at every point in the run. CHRGET processes one of these pointers, which is the location in RAM of the current byte of BASIC. A line like 100 ?"HELLO" is scanned by CHRGET from beginning to end, and in each case the accumulator contains a character - the token for PRINT, ASCII for ", ASCII for H, and so on. (The link addresses and line numbers are processed differently). This routine is therefore important to BASIC. As we shall see, extra BASIC commands can be inserted into BASIC by altering CHRGET.

It is easy to confuse the operation of CHRGET with the interrupt, as both deal with fast-moving events at the core of BASIC. The interrupt is a regular event, which occurs absolutely independently of BASIC, while CHRGET is necessary to process all BASIC bytes from start to finish. Thus, extensions to BASIC must use CHRGET, and TRACE, if it is to record every line processed, must also use CHRGET. Any other processing for which regular updates are wanted is an interrupt function; examples include repeat keys, cursor flash rate controls, Stop key testing, and music played while a normal program runs.

When the PET/CBM or VIC is turned on, part of the reset routine moves CHRGET from ROM to RAM. It is held at E0B5/ E0F9/ D399, and moved by a loop at E0E5/ E12F/ D3C7. For speed of operation, it is invariably held in the zero-page, at $70 in BASICs 2 and 4, at $C2 in BASIC 1, and at $73 in VIC. The version here (right) is that of BASIC 2 or 4. All the versions are similar, but since this is a self-modifying routine the address incremented at the very start of CHRGET varies.

Many 6502-based microcomputers with BASIC use an identical routine. It shows Microsoft's skilful use of the flags as indicators. By (for example) putting characters 0-255 into the screen RAM and pointing ($77) at their start, the effects of CHRGET can be determined empirically, and they are summarised in the following table:

A	0070	INC	77	;INCREMENT 2 BYTES
	0072	BNE	0076	
	0074	INC	78	
B	0076	LDA	0401	;LOAD FROM RAM
	0079	CMP	#3A	;':'
	007B	BCS	0087	;BRANCH IF ACC>=#3A
C	007D	CMP	#20	
	007F	BEQ	0070	;SKIP SPACES
D	0081	SEC		;SET CARRY FLAG
	0082	SBC	#30	;WHEN ACC IS OUT OF
	0084	SEC		;RANGE #30-#39,
	0085	SBC	#D0	;CLEAR IF #30-#39
	0087	RTS		

CHRGET and CHRGOT

JSR $0070 (labelled A) returns with the accumulator holding the *next* BASIC byte.
JSR $0076 (labelled B) returns with the accumulator holding the *current* BASIC byte.
Note that the accumulator never returns with a space character, however, which is always skipped by the loop labelled C.
Subroutines A and B are sometimes called 'CHRGET' and 'CHRGOT'.

Carry flag, C
Cleared if the accumulator holds #30 - #39, i.e. an ASCII numeral. (BCC succeeds).
Set if the accumulator holds any other value. (BCS succeeds in this case).

Zero flag, Z
Cleared if the accumulator holds any value other than null or colon. (BNE succeeds).
Set whenever the accumulator holds #0 or #3A. BEQ thus tests for end-of-statement.

Negative flag, N and internal overflow flag, V
These are not important, but, for the record:
N is cleared when A holds #00 - #B9, and set when A holds #BA - #FF.
V is always cleared, except in the one case when A holds #3A (the colon) *and* V was previously set.

Summary: (a) CHRGET skips spaces, (b) detects numerals such as linenumbers by JSR 0070/ BCC, (c) detects the end of a statement by JSR 0070/ BEQ, and (d) does *not* detect tokens by setting the negative flag, as it might have been expected to do.

14.3.2 Wedges. Extensions to BASIC all use so-called 'wedges'; this generic name applies to any RAM processing which is triggered via CHRGET. Toolkits, DOS support, trace utilities, additional keywords prefixed by !,$,@, and others, all rely on this method. Brad Templeton's EPROM 'Power' has an automated keyword-defining feature. The process is *basically* quite simple, but as usual there are plenty of hazards lying in wait for the unwary programmer.

Let's first look at some modifications of CHRGET which are not wedges.

(i) Deletion of the test for spaces. We can eliminate the loop at C on the diagram from BASIC. If we poke, in sequence, $7E with $EA (this is NOP, no operation), $7D with $EA, $80 with $EA, and $7F with $EA, the entire space-skipping loop is replaced by NOPs. The sequence must be followed; remember that POKE from BASIC itself uses CHRGET, so the subroutine must make sense at each intermediate step. CMP #EA always fails, since large values have already been taken out by the branch after CMP #3A; and, with BEQ 0070 preceded by BCS #3A, this branch too always fails. So:

POKE126,234:POKE125,234:POKE128,234:POKE127,234

with BASICs 2 and 4 eliminates the space test. Now, any BASIC including a space will give ?syntax error - including the POKEs listed above. This shows how BASIC which has been CRUNCHed, or written without spaces, can be run faster than normal, by altering CHRGET from machine-code to remove the redundant four bytes. (SEC can also be dropped if SBC #30 is replaced by SBC #2F).

(ii) Use of the CHRGET in ROM. Even if RAM is changed, the ROM routine remains immutable. We can exploit it in a variety of ways. A subroutine call to entry-point A will increment ($77), returning an irrelevant value in the accumulator. A call to CMP #3A will process the current accumulator contents in GETCHR's normal way, clearing C if the accumulator holds a numeral, for example; this is a compact, though non-ROM-independent, method for performing such tests. Entry-point D can be called from RAM; 0081 JMP ROM-D leaves CHRGET unaffected, apart from taking longer, and liberates three zero-page bytes - not RTS, which is still used by a branch.

Now let's look at some programming considerations relevant to wedges.

(i) Registers. X and Y are not altered by CHRGET; A, which is loaded with a new value, is. A wedge routine may need to save X and/or Y, if these are used in its processing. In the case of TRACE, and other extensions to BASIC which are intended to leave the program running normally, A will also need to be saved, and perhaps PSR.

(ii) Return address. When JSR 0070 or JSR $0076 is executed from ROM, the return address is saved on the stack. Therefore, a wedge with TSX/ LDA 0101,X /,,,/ LDA 0102,X can recover the return address - 1, and find where the call was made from. This is sometimes used (e.g. in DOS support) to exclude some calls, for example those in program mode as opposed to direct mode. If a wedge includes a JSR command, the extra return address may need to be popped from the stack, depending on the way the subsequent programming is carried out.

(iii) Current BASIC address. The address in ($77) can sometimes be used from within a wedge: an obvious example is testing for direct mode, where the high byte in $78 is #2, because the buffer extends from $0200 (except in BASIC 1).

(iv) Positioning a wedge within CHRGET. Only two basic types of wedge are possible; those with a JMP or JSR at the start of CHRGET, and those with JMP(s) or JSR(s) after 0076. These are fixed by ROM. Calls to 0070 and 0076 both occur from ROM, so each must be respected. Moreover, location ($77), the pointer to BASIC, is used in ROM, and therefore cannot easily be moved; this rules out 0076 JMP or JSR. Any location after line 0076 can include a wedge. 0079 JMP is an obvious possibility; 0081 JMP can be used, but will only be entered if the accumulator holds a value less than #3A, unless 007B is changed to 007B BCS 0081 and the routine exits through the ROM equivalent of 0079. The two types are illustrated here:

```
CHRGET JMP WEDGE ──→WEDGE INCREMENT ($77)      CHRGET INC 77        WEDGE CMP #CB
       xx xx xx          JSR 0076                     BNE 0076         ╱     BEQ +3 ───────────┐
CHRGOT LDA BASADR        CMP #CB                       INC 78         ╱      JMP ROM-CMP #3A│
       CMP #3A           BEQ +3 ──────────┐     CHRGOT LDA BASADR╱           ...PROCESS...←─┘
       BCS EXIT          JMP 0076         │            JMP WEDGE ╱           JMP 0070 or 0076
       CMP #20           ...PROCESS...←───┘            xx xx xx
       BEQ CHRGET        JMP 0070 or 0076              xx xx xx
       ...ETC...                                       xx xx xx
```

Early wedge in CHRGET.   **Later wedge, after CHRGOT.**

For the sake of an example, I have written each wedge to test for the GO token. This means that GO might be used to trigger a computed GOTO or GOSUB. Other possible symbols include any not usually found in BASIC, !,$,@, have been used, and other possibilities include shift-space, cursor-control characters, and space itself. A whole table of processes might follow such a symbol; thus @D 1,3 might plot a point at 1,3, where 'D' means 'Draw'; !C might clear the whole of memory; $D,1 might display a disk directory, and each of these commands might be only one of several alternatives. The initial special character makes processing somewhat easier, but isn't necessary, and keywords like DUMP or FIND can be checked for and acted upon when found.

Both examples use JMP commands. JSR is often used, and, where applicable, RTS can be used, for example in the wedge in the first example. However, if subsequent processing doesn't return to the same place, a construction of the type PLA/PLA/JMP 0070 will be necessary, where the return address is removed, leaving only the return address from which CHRGET was called.

Examples. These three examples illustrate some of the points mentioned, and introduce a few refinements which we haven't yet dealt with:

(i) <u>Use of ! from BASIC to reverse the screen.</u> This is straightforward; one of the routines in Chapter 9 can be inserted to perform the reversal. First, enter the routine (right) which processes the wedge.

Then enter the subsidiary machine-code t modify CHRGET, and run it. If this is done first, BASIC will crash, producing effects similar to those which occur when the IRQ vector is changed to some non-existent subroutine.

```
0300  CMP #21
0302  BEQ 0307 ;BRANCH IF ! IN ACC'R
0304  JMP E0BE/ E102/ D3A2;BASICS 1,2,4
0307  LDA 78
0309  CMP #02
030B  BEQ EXIT ;BRANCH IF DIRECT MODE
030D  TYA
030E  PHA         ;SAVE Y ON STACK
030F  --REVERSE SCREEN USING A & Y--
      PLA
      TAY         ;RECOVER Y
EXIT  JMP 0070 ;NEXT BASIC INSTRUCTION
```

Note that the wedge processing has a test for direct-mode; if this is in force the 'command' is ignored. The object of this is to prevent unwanted direct-mode screen reversals. This, of course, hardly matters, but it might be important in other cases. Conversely, *only* direct-mode might be required, as it is in the DOS support programs. Now,

```
.M 0079 0079
.: 4C 00 03 xx xx xx xx xx
```

converts CHRGET to its new form (right). LDA #4C/ STA 79/ etc can of course also be used. (BASIC 1 has different addresses). The mechanism by which the wedge operates should, I hope, be clear. Usually, CHRGET won't find !, and following the routine from

```
0070  INC 77
0072  BNE 0076
0074  INC 78
0076  LDA BASADR
0079  JMP 0300
```

the start on this assumption shows that the operation of CHRGET is exactly as normal, except for a small slowing caused by the extra code.

(ii) <u>TRACE.</u> The version in Chapter 5 inserts a wedge at entry-point D, of the form JMP WEDGE where the wedge is high in RAM. The code then follows the sequence STA/ STX/ STY/ LIST LINE, WHERE NECESSARY/ LDA/ LDX/ LDY / JMP E0C6 or E10A or D3AA. These latter alternatives apply to BASICs 1,2, and 4, and are the ROM entry-points which correspond to D. In this way, CHRGET is left fundamentally unaltered, but processing takes place using parameters (e.g. current linenumber) of BASIC. This is a slightly risky routine for use with TRACE, because entry-point D can only be reached if the BASIC byte is in the range #0 - #$39. The wedge is therefore often bypassed. In practice, all BASIC lines include an end-of-line null character and/or ASCII numerals in GOTO, and the added versatility given by a wedge which can leave others in place to run normally is an advantage.

(iii) <u>Computed GOTO and GOSUB.</u> The favoured form of wedge is the replacement of 0070 INC 77 by JMP WEDGE , where the address WEDGE is typically in high RAM or in EPROM. Unlike wedges lower down in CHRGET, this assures priority of whatever new coding is to be introduced. The logic is slightly different from the previous examples, because the wedge replaces GETCHR rather than GOTCHR. In other words, wedge processing here must include incrementing the contents of ($77).

An elegant way to do this is to use the three bytes after JMP. A subroutine to incre-
ment ($77)'s contents can be put here, so JSR 0073 functions exactly like CHRGET. The
subroutine can be INC 77/ BNE +3/ INC 78 or, what is perhaps slightly easier, the ROM
routine. The revised CHRGET routine and its wedge has this structure:

```
0070  JMP WEDGE            WEDGE  JSR 0073 ;LOAD A WITH NEXT BASIC CHR.,
0073  JSR E0B5/E0F9/D399          ;SET STATUS FLAGS, AS GETCHR
0076  LDA BASADR etc.          ...Perform processing; return either
      ...unchanged...          with JMP 0076 or with RTS, if A and
                               status flags are correctly set. ...
```

The following example is a computed GOTO using the GO token; it is written for BASIC 2
but easily modified to BASIC 4. There is a shorter, but less easy to understand, version
by B Templeton [in BASIC 2: JSR CC8B/ JSR D6D2/ JSR C7B0/ JMP 0076] which reappears
in the GOSUB routine below, apart from the additional stack manipulations. It is poss-
ible to combine computed GOTOs and GOSUBs in the same wedge, using this.

```
0070  JMP 0300              0300  JSR 0073 ;GETCHR. NEXT BASIC BYTE IN A
0073  JSR EOF9              0303  CMP #CB  ;IS IT A 'GO' TOKEN?
0076  ...unchanged...       0305  BNE 0321 ;NO. EXIT TO RESET STATUS FLAGS
                            0307  JSR EOF9 ;YES: INC POINTER TO NEXT BYTE,
.M  0070 4C 00 03 20 F9 E0 xx xx   030A  JSR CC9F ;INPUT AND EVALUATE EXPRESSION,
                            030D  JSR D6D2 ;CONVERT FPACC#1 INTO INTEGER,
With this wedge in place, try this:   0310  JSR C52C ;SEARCH FOR LINENUMBER IN ($11),
                            0313  BCC 0324 ;CARRY CLEAR IF LINE NOT FOUND;
    0 PRINT "1": GO SQR (64)    0315  LDA 5C   ;($5C) POINTS TO LINK ADDRESS,
    4 PRINT "3": GO X*X + 4     0317  ADC #03  ;SO ADD #4 [CARRY IS SET]
    8 PRINT "2": X=4: GO X      0319  STA 77   ;AND PUT RESULT IN ($77).
   20 PRINT "4": END            031B  LDA 5D
                            031D  ADC #00
which prints 1 to 4 during execution.   031F  STA 78
Some odd anomalies tend to occur with   0321  JMP 0076 ;CONTINUE BASIC
wedges unless there is specific pro-     0324  JMP C7EB ;'UNDEF'D STATEMENT ERROR'
tection within them against direct-mode
```

entry and entry from the input buffer.
If either of these is omitted, entering
programs when the wedge is enabled may give oddities. Lines like 100 :! may be nec-
essary. The DOS support ('Universal Wedge') is instructive in this respect. Another
type of anomaly is the behaviour of conditional statements; IF X=1 THEN @123 , a typ-
ical wedge, may *always* execute, irrespective of the value of X, unless rewritten in
the form IF X=1 THEN: @123.

Since the wedge is processed before CHRGET
is able to pass its results to ROM, ordinary tokens
can be altered: #$89 ('GOTO') can itself be mod-
ified, for instance. Computed GOSUB (right) is
more complex than GOTO. On disassembling GOSUB
in ROM, one finds a lot of stack activity followed
by GOTO. RETURN unravels the stack information.
It is important with a wedge to save the return
address of GETCHR (which is called by GETCHR's
final RTS), for use by the exit JMP 0076 from the
wedge. In the example it is stored in RAM.

Subroutines can be called by name, rather
than linenumber. If a date-processing subroutine,
say, starts at 10000 and DA=10000. then GOSUB
DATE is usable. Such names must contain no res-
erved BASIC words, of course, and their values
must be correct. They will not, for example, in
general survive renumbering correctly.

The use of location $300 in these examples
has no particular significance and is for convenience
only. Note that all wedges inevitably have a slowing
effect on BASIC, so a routine to 'kill' the wedge
when it's not in use may be worth including in the
overall program. Also, of course, a routine to turn
it on is convenient; SYS 40960 or SYS 45056
has this function in many hardware add-ons.

```
0300  JSR 0073 ;GET NEXT CHARACTER
0303  CMP #8D  ;GOSUB TOKEN?
0305  BNE 0337 ;IF NOT, EXIT.
0307  PLA      ;RECOVER RETURN
0308  STA 033A ; ADDRESS AND
030B  PLA      ; STORE IT.
030C  STA 033B
030F  LDA #03
0311  JSR C31B ;CHECK STACK DEPTH
0314  LDA 78
0316  PHA
0317  LDA 77
0319  PHA
031A  LDA 37
031C  PHA
031D  LDA 36
031F  PHA
0320  LDA #8D
0322  PHA
0323  JSR EOF9 ;INCREMENT (77)
0326  JSR CC8B ;EVALUATE EXPRESSION
0329  JSR D6D2 ;CONVERT TO 2-BYTES
032C  JSR C7B0 ;UPDATE (77) ADDRESS
032F  LDA 033B ;REPLACE RETURN
0332  PHA      ; ADDRESS FOR RTS
0333  LDA 033A ; ON STACK
0336  PHA
0337  JMP 0076 ;CONTINUE
033A           ;2 BYTES STORAGE
```

14.3.3 BASIC utilities. Chapter 5 has a number of examples of machine-code programs which process BASIC. Two examples here exemplify suitable methods for dealing with this type of problem.

(i) Search and replace. A fairly simple routine for any BASIC, which changes single characters, follows. It can exchange all occurrences of PRINT to PRINT#, or of PEEK to USR, for example; or, within quotes, all shift-spaces may be converted into spaces, or all cursor-downs into (say) reverse characters. The logic is shown in the flowchart. Note (a) how the end-of-line test is necessary to prevent link addresses and linenumbers being wrongly changed, (b) how the end-of-program is tested, (c) the use of the quotes flag to show whether or not some particular region of BASIC is within quotes, and therefore may require special treatment.

The routine uses addresses $0 - $2 in the zero-page, but no other RAM apart from its own (relocatable) code.

```
$00=QUOTES FLAG   (#0 OFF,  #FF ON)
$01=POINTER (LOW BYTE)
$02=POINTER (HIGH BYTE)
```

```
0300   A2 00 86 00 A0 01 A9 04
0308   85 01 85 02 E6 01 D0 02
0310   E6 02 A1 01 F0 1A C9 22
0318   D0 08 48 A5 00 49 FF 85
0320   00 68 E4 00 (DO E6) C9 (99)
0328   D0 E2 A9 (98) 81 01 D0 DC
0330   B1 01 C8 11 01 F0 11 88
0338   18 86 00 A9 05 65 01 85
0340   01 8A 65 02 85 02 D0 CA
0348   60 xx xx xx xx xx xx xx
```

(#99) ('PRINT') IS CHANGED TO
(#98) ('PRINT#') IN THIS EXAMPLE.

(DO E6): IGNORE CONTENTS OF QUOTES,
(FO E6): CHANGE ONLY QUOTE CONTENTS,
(EA EA): CHANGE BOTH, ON SYS 768

N.B.: The quote character itself cannot be changed using this routine as it stands.

(ii) Hashtotal. A BASIC or machine-code hashtotal is often helpful in checking whether load errors may have occurred or whether the correct version of a program has been loaded. We need a routine to combine each byte from the start to the end in a repeatable way. The example program (next page) creates a single-byte hashtotal, by exclusive-ORing every byte from ($28) to ($2A) and printing the result. This value could be stored in RAM and checked automatically by the program itself. The routine uses no zero-page pointers, but instead is self-modifying. The hashtotalling process stops when this modified address equals the end-of-program pointer in ($2A). Obviously, other zero-page start and end pointers can be substituted, such as ($7A) for BASIC 1 or ($2B) for VIC. The print routine (this prints linenumbers, and can be tracked down at the end of RESET, where it prints the number of bytes free) differs between ROMs. I have used SYS 700 because the location is easy to remember, at least in decimal. The routine needs six address changes if it is to be relocated.
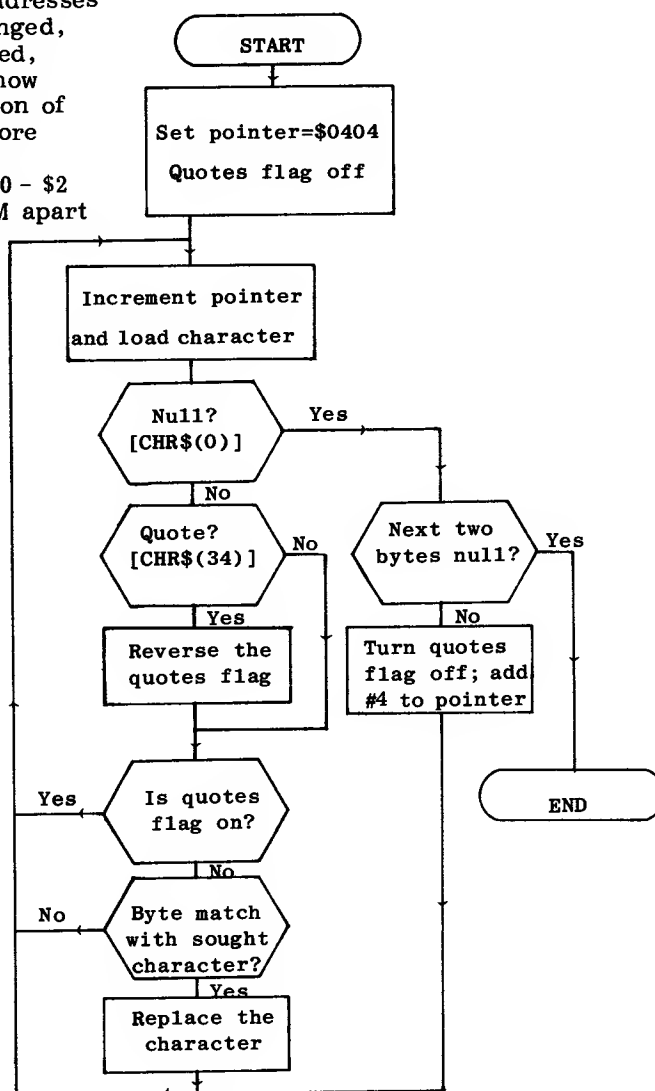
Flowchart: replace only characters outside quotes.

```
        02BC  LDA 28    ;STORE START-OF-PROGRAM POINTER IN RAM ADDRESS
        02BE  STA 02C9
        02C1  LDA 29
        02C3  STA 02CA
        02C6  PHA
LOOP    02C7  PLA       ;EXCLUSIVE-OR POINTER CONTENTS WITH HASHTOTAL SO FAR
        02C8  EOR xxxx
        02CB  PHA
        02CC  INC 02C9  ;INCREMENT POINTER
        02CF  BNE 02D4
        02D1  INC 02CA
        02D4  SEC       ;COMPARE POINTER WITH END-OF-BASIC. BRANCH TO LOOP WHILE LESS
        02D5  LDA 02C9
        02D8  SBC 2A
        02DA  LDA 02CA
        02DD  SBC 2B
        02DF  BCC 02C7
        02E1  PLA       ;RECOVER HASHTOTAL (STORED ON STACK AS SBC USES ACCUMULATOR)
        02E2  TAX
        02E3  LDA #00
        02E5  JMP DC9F/ DCD9/ CF83; PRINT 256*A + X, WHICH NOW = HASHTOTAL
```

14.4 Machine-code loaders in BASIC: ordinary loaders and relocating loaders.

14.4.1 Ordinary loaders. When machine-code is to be stored in some fixed place in RAM – notably the cassette buffer(s), which, except for buffer#2 in BASIC 4 disk operations, are only used by BASIC during tape input/ output – a simple series of pokes provides an easy way to load the individual bytes. Section 4.1.9 has a BASIC routine which uses the keyboard buffer and screen together to convert consecutive bytes into decimal BASIC values, for later poking back into RAM. It may be easier and less space-consuming to use hexadecimal strings; the two subroutines below are complementary, the first generating strings like that in line 0 of the second program, which in turn, given a starting address, reconstitutes the code in RAM.

```
61491 REM ##############################################################
61492 REM ## ROUTINE WHICH STORES MACHINE CODE FROM RAM INTO A BASIC STRING. ##
61493 REM ## ** SEE THE FOLLOWING ROUTINE FOR TYPICAL PROGRAM TO RECONSTRUCT ##
61494 REM ## **     THE CONTENTS OF RAM FROM WITHIN ANOTHER BASIC PROGRAM.    ##
61495 REM ##############################################################
61496 REM
61500 INPUT " RAM START LOCATION";S
61510 INPUT "   RAM END LOCATION";E
61515 INPUT "STARTING LINENUMBER";L
61520 PRINT "[CLEAR]";MID$(STR$(L),2);"MC$=MC$+";CHR$(34);:G=PEEK(54)+256*PEEK(55)
61530 FOR J = S TO E
61540 IF POS(0)+ PEEK(196)>74 THEN PRINT CHR$(34);"[HOME] [DOWN] [DOWN]L=";L;"+1:S=";J;":E
      =";E;":GOTO";G
61550 IF POS(0)+PEEK(196)>74THEN POKE623,19:POKE624,13:POKE625,13:POKE158,3:END
61560 P=PEEK(J):Q%=P/16:P=P-Q%*16:REM Q% IS HIGH BYTE, P LOW BYTE, IN DECIMAL.
61570 C=P:GOSUB 61600:C=Q%:GOSUB 61600:REM Q%>9 THEN Q%=Q%+16: PRINTCHR$(Q%);
61580 NEXT
61590 PRINT CHR$(34): POKE 623,19: POKE624,13: POKE158,2: END: REM LAST LINE
61598 REM
61599 REM ** CONVERT DECIMAL FROM 0-15 INTO HEX FROM 0-9, A-F AND PRINT DIGIT **
61600 C = C + 48: IF C > 57 THEN C = C+7
61610 PRINT CHR$(C);
61620 RETURN

0 MC$=MC$+"E600F0034C2EE6A9FF8500A578EE0C038D48E84C2EE600":REM EXAMPLE ONLY
61745 REM
61746 REM ##############################################################
61747 REM ## TYPICAL PROGRAM TO CONVERT A BASIC HEX STRING BACK INTO MEMORY ##
61748 REM ##############################################################
61749 REM
61750 INPUT "START LOCATION OF CODE"; S
61760 FOR J = 1 TO LEN (MC$) STEP 2
61770 Q% = ASC( MID$(MC$,J,1) ) :  REM ASCII VALUE OF HIGH BYTE
61780 P = ASC( MID$(MC$,J+1,1) ):  REM  ASCII VALUE OF LOW BYTE
61790 Q% = Q% - ASC("0"): Q% = Q% + 7* (Q%>9) : REM DECIMAL VALUE OF HIGH BYTE
61800 P = P - ASC("0"):   P = P  + 7* (P>9)    : REM DECIMAL VALUE OF LOW BYTE
61810 POKE S + J/2 , 16 * Q% + P               : REM NOW POKE IN TRUE VALUE
61820 NEXT
61830 END
```

14.4.2 Relocating loaders. Machine-code which works correctly in any part of memory (subject to constraints imposed by the other software and hardware) is called *relocatable*. Code of this sort can be put into RAM by a straightforward set of pokes, with a variable starting-point from which the bytes are written. This can't be done, without modifications, with most code using absolute addresses. A relocating loader pokes in code, correcting the relevant bytes. As an example of its use, consider BASIC using several routines in high RAM; a relocating loader can painlessly put (say) several different keyboard redefinitions there, all fitting tidily into the space, and all working correctly. In the same way, a loader can put in its code into machines of differing RAM capacity. Relocating code may also be machine-independent, but this is more difficult.

Which instructions relocate? All implied mode and immediate mode instructions, all branches, and all accumulator mode instructions relocate. For example, TSX, RTS, CLC, and LDX #00, LDA #FF, and BEQ +6 and ROL A can be poked in byte form anywhere in memory without affecting their disassembled equivalent in any way. The problems arise with addresses. With BASIC, zero-page instructions can usually be considered to relocate, because their functions are fixed, and an instruction like LDA ($2A),Y has to be retained wherever the code is. ROM addresses are fixed too. Addresses which have to be varied look like this: 7000 JSR 70E4/ LDA 70B0,X/ CMP 7100/ JMP 7050 and so on, which after relocation becomes 6000 JSR 60E4/ LDA 60B0,X / etc.

Many approaches are possible to writing such code: here, I'll assume the code is to be put into the top of RAM, and is to be loaded in decimal from BASIC. The well-known use of negative numbers as distinguishing marks is used; machine-code versions can't do this, and may use zero bytes instead, followed by a routine enabling a check for real zeroes as opposed to code zeroes. Supermon 4 (q.v. - appendices) has an example.

We can use the following loader, which may be embellished in various ways, typically to print out initialisation addresses, special locations, and instructions. The peeked values apply to BASIC>1; others may be substituted:

```
100 T=PEEK(52)+256*PEEK(53)    :REM TOP OF MEMORY FOR BASIC 2 & 4
110 L=T-N                      :REM N=NUMBER OF BYTES OF CODE; L=LOWERED MEM.TOP
120 FOR J=L TO T-1: READ X%    :REM DATA HELD IN (SAY) LINES 0 AND FOLLOWING
130 IF X%<0 THEN Y=X%+T: X%=Y/256: Z=Y-X%*256: POKE J,Z: J=J+1 :
                               :REM Y IS RELOCATED VALUE CALCULATED FROM NEG.X%
140 POKE J,X%: NEXT            :REM COMPLETE PROCESS FOR ALL VALUES
150 POKE 52,L-INT(L/256)*256: POKE 53,L/256: CLR:REM RESET TOP-OF-MEMORY
```

To convert code into data which this program can use, follow these steps:

(i) Enter the code into RAM (and preferably test it).

(ii) Print (or write out) the disassembled version. A disassembler giving decimal values of locations is helpful.

(iii) Mark all the absolute addresses which need changing during relocation.

(iv) Replace each of them by its offset from the end of the program; i.e. count from the end of program plus one backwards, the result being a negative number from -1 to -30000 or so. See the example; this is easier than it might seem.

(v) Convert the bytes into data statements and enter them. Note that each new negative value replaces *two* bytes as a rule.

(vi) Enter the value of N in line 110.

(vii) Test the loader: run it several times, and check that each routine is independent and correctly set up.

Example. The nonsense program (right) has a subroutine call, a table of byte values, and a branch. The branch, because of its relative addressing mode, relocates; so does the table, and the single-byte, implied-mode instructions and the immediate-mode instruction. So the only addresses to be relocated are those circled.

32	126	2	027A	JSR (027E)
96			027D	RTS
162	2	(-11)	027E	LDX #2
221	134	2	0280	CMP (0286),X
202			0283	DEX
208	250		0284	BNE 0280
96		(-3)	0286	RTS
65	66		0287	.BYTE $41,$42

Counting back from the end, we find that 027E is the 11th byte, and 0286 the third; so -11 and -3 respectively replace all occurrences of these two addresses. The DATA statement is therefore

```
0 DATA 32,-11,96,162,2,221,-3,202,208,250,96,65,66
```

and the number of bytes in the program is 15, so line 110 becomes

```
110 L=T-15
```

Our loader should now be capable of placing its code into memory as the diagram (right) shows, with adjacent versions of the routine abutting exactly. Another temporary line of BASIC:

        145 PRINT "TOP" T "TO" L

will show the continual diminution of RAM as the routines accumulate in the top of RAM.

    Refinements on this process include:
(i) Test for type of ROM, perhaps with a few pokes to modify ROM addresses,
(ii) Test for size of memory, which may be too small,
(iii) Automation of some of these processes, including calculation of negative values and of number of bytes in the program,
(iv) Inclusion of the memory-lowering pokes into the initialisation routine itself. In this way, a pre-relocated machine-code program can be loaded as a file, and when initialised, will set the memory-pointers so that it cannot be overwritten by BASIC strings.

```
3BC4   JSR 3BC8
3BC7   RTS
3BC8   LDX #02
3BCA   CMP 3BD0,X
3BCD   DEX
3BCE   BNE 3BCA
3BD0   RTS
3BD1   EOR (42,X)
3BD3   JSR 3BD7
3BD6   RTS
3BD7   LDX #02
3BD9   CMP 3BDF,X
3BDC   DEX
3BDD   BNE 3BD9
3BDF   RTS
3BE0   EOR (42,X)
3BE2   JSR 3BE6
3BE5   RTS
3BE6   LDX #02
 .... and so on ....
```

14.5 Pure machine-code techniques.

'Hand assembly' This is the name usually given to a hybrid technique for machine-code programming, in which the final code is not fitted together as a solid chunk in the manner of an assembler, but instead is distributed in RAM in a way convenient to the programmer, in separate subroutines. For example, a disk-processing program may start at $3000 and have 50 or so lines of program terminating with a message to be printed on successful completion. Major subroutines, to read, write, compare, move data, and so on, could be at $3100, $3200, $3300, and other addresses in this series. Provided that the documentation keeps a record of the function of each subroutine, code built up in this way is fairly easy to check (subroutines can be tested individually) and quite easy to modify, without having to reassemble the entire program. Moreover, skeletal trial programs can be written and run, and later elaborated upon and made user-friendly by expanding parts of the code as required. The sequence of the parts is not changed by this process, as it may be when assembler programs are patched. Different depths of subroutines can be represented by their locations; as a COBOL program might have controlling modules prefixed by A-, their subsidiary routines prefixed by C-, and their elementary commands to read or write single records prefixed by E-, so $3000 ff may contain the highest-level control programs, $4000 ff their subroutines, and $5000 ff the elementary subroutines. Since enormous machine-code programs are rare, there is not often a shortage of RAM for the purpose. There is of course no reason why this procedure shouldn't be carried out with an assembler; all that's needed is a fair sprinkling of commands like *=3000 and *=3100. Inevitably, JMP or JSR commands (or stack pushes with RTS or RTI) have to be used to communicate between routines; branches cannot reach far enough. (This qualification is removed with the 6809 chip, which not only has 'long branches' but 'branch saving return address'. These features make 6809 code much more easily relocatable than 6502 code).

Running with SYS. LOAD assumes BASIC; RUN is expected, the CBM having no command to run pure machine-code. If we have a machine-code program, how can we write it so that RUN executes it? The answer is to put in a SYS call to the code. To see how this can be done in the general case, let's suppose we have a routine starting at $3000. RUN executes a program from its starting-point; we therefore precede the machine-code with a BASIC program. The easiest way to do this is to insert bytes like these:

ACTUAL BYTES:	00 0C 04 00 00	9E 31 32 32 38 38 00 00 00
BASIC EQUIVALENT:	ɴ Link Line#	SYS 1 2 2 8 8 ɴ ɴ ɴ

where ɴ denotes the null byte. The link address (=$040C here) points to the first of the two end-of-program marker bytes, on the assumption that $0400 is the address of the start of this machine-code. The link address cannot be zero. The arrow marks

the start of BASIC, as determined by the pointers in ($28) in BASICs >1, and ($7A) in BASIC 1. Unfortunately, the byte previous to this must be a null byte, imitating an end-of-line, and this usually only happens with the standard loading and saving procedure where $0400 holds #0. So entry of

```
.0400 00 0C 04 00 00 9E 31 32 ;SYS 12288 is assumed here, but obviously other
.0408 32 38 38 00 00 00 xx xx ; values, from SYS1039 up, are possible.
```

and
```
0028 01 04 xx xx xx xx xx xx  ;Top of memory underlined; must > $3000, e.g.FF 39.
```

followed by SAVE will convert the machine-code into a pseudo-BASIC program, which can be RUN. In this case, all the bytes from $0410 to $3000 are wasted; to prevent this the code could be relocated down, or another loader used to put a zero byte in the position corresponding to $0400, boring though this may be.

It is not strictly necessary to include three null bytes after the BASIC SYS and END program, but it tidies LIST.

14.6 Debugging machine-code.

This list, which is naturally not exhaustive, includes many errors which experience shows to be common in 6502 programming. Errors in the design itself are best cured at the earliest stages; careful analysis and dry-running of code with both typical and abnormal data should ensure that a program is fundamentally sound.

Simple errors of carelessness. These may remain undetected for a long time in mach-code, because no ?syntax error warning ever appears. Examples include:
   (i) Transcription errors. Typically 7038 for 703B.
   (ii) Omission, or inclusion, of immediate-mode #, as in LDA #01/ PHA/ LDA 02/ PHA.
   (iii) Use of wrong ROM addresses, for example FFE4 as output, or, with ROM
      addresses not in the kernel, those for a different ROM from that in use.
   (iv) Branch errors are quite likely to occur in code not written by assembler.

Addressing mode errors include
   (i) Confusion of order of low and high bytes of an address.
   (ii) Failure to understand the method of working of indirect addressing.
   (iii) The attempt to use indexed zero-page addressing to extand above $FF.
      LDA $AB,X wraps around back to zero if X exceeds #54.
   (iv) Other indexing errors, for example: LDA 0102,X/ STA 0100,Y/ DEX/ DEY/
      BEQ -10 loads garbage if X drops to #FF and below, which is untested.
   (v) Program design may be weakened through failure to appreciate limitations of
      the chip; e.g. tripling A by TAX/ ASL A/ ADC X is impossible.
   (vi) Indirect jump has a bug: JMP (03FF) takes its address from 03FF and 0300.

Calculation errors involve addition, subtraction, and negation, for example:
   (i) It's easy to forget that only the accumulator is holding the result.
      LDA #2/ ADC 1234 adds 2, but the contents of 1234 are unchanged. To
      change 1234, STA 1234 is necessary.
   (ii) The carry bit may give problems: the rule is usually to clear it before add-
      itions, and set it before subtractions.
   (iii) A 2's complement is always 256 minus the original byte, or 65536 minus a
      16-bit integer, and so on. This is EOR #FF + #1.

Errors with status flags. There is a logic behind the setting of flags, but it is not easy to get used to it. This example: LDA AB/ CMP #07/ BEQ +1/ RTS/ STA 08 stores a zero value in 08, but this does *not* set the zero flag, although loading the value from 08 does set the flag. Other problems include the negative flag with CMP, and the fact that incrementing a value from 127 decimal to 128 makes the value change from 'positive' to 'negative'.

Stack errors: the rule is to have the same number of stack pushes and pulls if a subroutine is to return in the normal way. If a subroutine stack is pulled before being pushed, it is important to return the correct values on the stack before RTS unless special processing is being performed.

Errors in which a program is modified include programs partially overwritten by BASIC, or by cassette activity, or by BASIC 4 in cassette buffer #2, or by the program itself. 2-byte pointers may be updated while they are still in use, so that they temporarily point to a wrong area of memory. Often, A,X, or Y is changed by a subroutine or interrupt, and has to be saved in RAM or on the stack.

14.7 IEEE-488, VIA, and PIA: interfaces to the outside world

The IEEE bus (General Purpose Interface Bus). This standard 'bus', permitting interchange of data between devices, is described in the IEEE document IEEE-488, published in 1978 and itself based on a 1975 standard. Any devices, subject to certain limitations, become plug-compatible, and able to transmit and receive ASCII data. Why should this be a problem? In the first place, innumerable variations are possible in the signals controlling the data flow. A standard set of commands indicating which lines are to be active and when is needed. Secondly, if all the devices were timed by some central clock, data could be passed without problems of synchronization between the devices: the same order of magnitude as obtains within RAM could apply to data transmission rates. But independent pieces of equipment are not synchronized, and a fairly elaborate system of checking is used to determine when data is to be sent. This process is called 'handshaking'. The two-way capability of the bus makes for complication. IEEE equipment tends to be costly. Moreover (this happens in many computer-related fields) large chunks of the standard may remain unused. The IEEE standard's design parameters were apparently based on the characteristics of equipment already in use, and this design approach has obvious risks.

Description of the bus. The IEEE-488 bus is a cable of 16 wires. 8 wires carry data, usually in ASCII form with bit 7 used as a parity check. The data lines transmit one bit each, so that an 8-bit byte is sent as a unit; the result is sometimes described 'bit parallel, byte serial'. Handshaking is carried out between each byte. Not surprisingly, this slows the rate of transmission, which in any case will be slow if one or more of the receiving devices processes its data at a relatively leisurely pace. 1 megabyte per second is the maximum allowed by the design; CBM equipment has a maximum of about 5000 bytes per second. 3 of the remaining 8 wires control the handshake. The CBM's ROM from \$F000 upwards includes its IEEE processing, in addition to tape and monitor programs. A fair proportion of its IEEE work is concerned, as we shall see, with setting these handshaking lines and generating error messages and ST (status flag) values if the returning signals don't behave in the correct handshaking manner. Finally, 5 wires are concerned with bus management. Only 2½ of these are used in the PET/CBM. This is caused by Commodore's design system, in which the computer gets priority over all other devices on the bus. There are quite severe restrictions on cable length between pieces of equipment: not more than 5 metres between devices, and not more than 20 metres overall, are the figures usually quoted. For this reason, VIC has a different connector, based on the RS232, which has a much better linelength. The RS232 (for example) can operate with two wires only. However, since interface boxes are available, this makes no great difference, except that the price of the total equipment package is raised. The advantage of the IEEE emerges when electronic and scientific equipment of a technical type (i.e. not printers) requires control by a CBM computer. Before describing the programming of this bus, we'll look at the IEEE port as it appears on all PET/CBM machines. The meaning of the mnemonics will (I hope) become less obscure as we proceed. The early manuals for the computers, for example part no. 320856-3, contain hardware data on the pin connections, their hardware addresses, handshaking and the management bus, ST, and a much-reproduced table of IEEE commands. There's also some account of CMD, GET#, INPUT#, and PRINT#, which of course are all BASIC's way of moving data on the bus.

The IEEE port.   The IEEE port is in the middle of the back of the PET/CBM. Its pins, and the corresponding IEEE connector as it appears (say) with a disk unit, labelled with IEEE mnemonics, are arranged like this:-
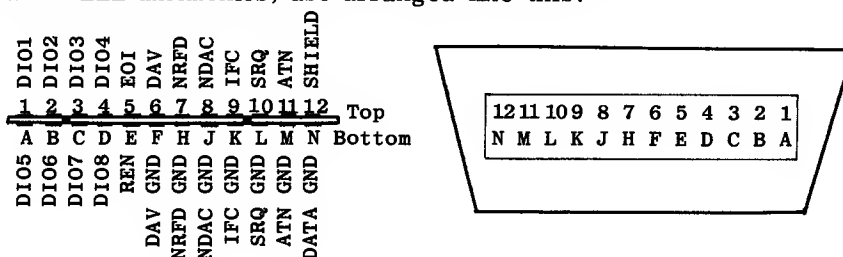
IEEE mnemonics and concepts. As we have seen, there are three conceptually separate sets of wires or lines in the IEEE bus. These are called Data Lines, Data Byte Transfer Control Lines (for handshaking!) and General Interface Management Lines (which the controller uses). Each of the 16 lines has a mnemonic.

DATA LINES

DIO1 - DIO8 are 8 data lines ('Data input/output') which carry single bytes of data and of commands.

DATA BYTE TRANSFER CONTROL BUS

DAV   ('Data valid'). Tells listener that new data is on the bus.
NDAC ('Not data accepted') Tells talker data hasn't been read yet.
NRFD ('Not ready for data') Tells talker not to talk yet.

GENERAL INTERFACE MANAGEMENT BUS

ATN   ('Attention') Distinguishes commands to devices from data.
EOI   ('End or Identify') Indicates that the current byte is the last.
IFC   ('Interface Clear') Clears all devices on switchon or reset.
REN   ('Remote Enable') Gives control to other device (not used with CBM).
SRQ   ('Service Request') Allows a device to request service (not on CBM).

All devices on the bus are controlled, at any given time, by a single 'controller'. The other devices may be 'talkers' or 'listeners'. A 'talker' transmits only; some technical measuring devices are of this type. A 'listener' receives data only; many printers and plotters illustrate this. Another type, the 'talker/listener', as you will not be surprised to read, can perform both activities; Commodore disk drives and modems illustrate this. The bus may be arranged with devices in a 'star' pattern or 'daisy-chained' together, or a combination of these; it doesn't matter to the bus. These devices may be any mixture of talkers and listeners. Much of the time the devices may be inactive or switched off. A 'talker' doesn't have to talk all the time.

The next important concept to grasp is the *active low* principle which the IEEE uses. Unlike all the remaining operations of the CBM, on the IEEE 'true' is low, (0), and 'false' is high (1). This applies to data and commands. In machine code, therefore, data is EORed with #$FF before transmission. With the CBM, the output register is $E822, so EOR #$FF /STA $E822 precedes data transmission. Another example (see next page) is the values assigned to the IEEE locations when the CBM is reset or powered on; each bit which is configured for output by the initialisation system is set high, rather than the alternative convention of the low value. This convention is determined by hardware considerations. Any one device can hold a line in the low state by keeping the line impedance high, irrespective of other devices' states, and this is useful when assorted devices with a range of response times have been connected to the same bus system. The 'active low' principle is responsible for the double negatives which tend to be a confusing part of discussion about this bus, particularly when it concerns the control bus commands which use lines which wait to be released by all the devices. For example, a device listening on the bus and ready to receive data sets 'Not ready for data' false, by setting the line high. This of course is the same as 'Ready for data'.

Finally, a vital concept without which nothing will make sense. The 'ATN' line – read as 'Attention' – distinguishes between commands and data. When it is low, (true), each byte sent is treated by all devices as a command, not as data. If the command refers to a particular device, that device becomes a talker or a listener and waits for the handshaking process to begin. Now the point is this:– only one byte carries the information telling the devices which device is to talk, say. How can the devices distinguish a 'talk' command from a 'listen'? In fact each command byte is partitioned up, so that the range within which the command byte lies determines its meaning. For example, if it is from 0-31 decimal, the command is a special type which we've not discussed and isn't used on CBM machines. If it is from 32-62, the command is a listen address; if from 96-126, a secondary address. As an example, consider the Commodore disk unit; this is device #8 (unless modified). The unit is made a listener by (i) Setting ATN true (0); (ii) ORA #$20 with the device number, 8, and so setting the relevant 'listen address' bit; (iii) Sending #$28 as a command; (iv) setting ATN false (1). Further transmissions will be understood as ASCII. A secondary address will often be sent too.

CBM IMPLEMENTATION OF THE IEEE-488 BUS*

Function:	Description:	Location:		Bit number:	Value on setup:
CONTROL LINES:					
ATN in	Attention	$E821	(59425)	7	0
ATN out		$E840	(59456)	2	1
DAV in	Data Valid	$E840	(59456)	7	0
DAV out		$E823	(59427)	3	1
EOI in	End or Identify	$E810	(59408)	6	0
EOI out		$E811	(59409)	3	1
NDAC in	Not data accepted	$E840	(59456)	0	0
NDAC out		$E821	(59425)	3	1
NRFD in	Not ready for data	$E840	(59456)	6	0
NRFD out		$E840	(59456)	1	1
SRQ in	Service request	$E823	(59427)	7	0
DATA LINES:					
	Input	$E820	(59424)	0-7	
	Output	$E822	(59426)	0-7	FF

PIA 1		
E810	.a..	a (#40) = EOI in
E811	.... b...	b (#08) = EOI out
E812		
E813		
PIA 2		
E820	iiii iiii	input register
E821	c... d...	c (#80) = ATN in, d (#08) = NDAC out
E822	oooo oooo	output register
E823	e... f...	e (#80) = SRQ in, f (#08) = DAV out
VIA		
E840	gh.. .jkl	g (#80) = DAV in, h (#40) = NRFD in, j (#04) = ATN out, k (#02) = NRFD out, l (#01) = NDAC in
E841 to E84F		

*References include:

i) IEEE Std 488-1978 describes the 'GPIB' (General Purpose Interface Bus) and includes a full specification.

ii) Gregory Yob's three part article in Kilobaud-Microcomputing (July - Sept.'80), 'get your PET on the IEEE bus', has a lot of information in about 23 pages. This includes hardware examples (e.g. Hewlett-Packard clock and signal generator, 'Blinkin' Lites'machine), BASIC routines to illustrate the workings of the bus, explanations of IEEE activity during input/output (e.g. INPUT# and PRINT#) and machine-code routines including some ROM locations.

iii) 'PET and the IEEE-488 Bus (GPIB)' by Fisher and Jensen (McGraw-Hill 1980) deals mainly with old ROM PETs. The book largely consists of detailed breakdowns of the BASIC I/O commands and lists of references- including instruments using the IEEE and a bibliography. It is hardware oriented; software examples include flowcharts, a BASIC diagnostic program to report faults on the bus, and a single machine-code example, a reprint of a routine to drive an astronomical telescope.

Machine Code programming of CBM's version of the IEEE: Examples.

Machine Code:-	Meaning:-	Machine Code:-	Meaning:-
BIT $E810 BVC SET ST	Branch taken if input EOl is high (false). 'Branch if not end of input message'.	LDA $E840 BPL -5	Wait until DAV in is high (false) 'Data not valid'.
LDA #$34 STA $E811	Set output EOl low (true)	L BIT $E840 BMI L	Wait until DAV in is low (true). 'Data valid'.
LDA #$3C STA $E811	Set output EOl high (false), 'Not end of message'.	LDA $E840 AND #$40 BEQ -7	Wait until NRFD in is false, i.e. until 'Ready for data'.
LDA $E820 EOR #$FF	Get a character from the input register & reverse it.*	LDA $E840 ORA #$02 STA $E840	Set NRFD out high (false). 'Ready for data'.
LDA #$34 STA $E821	Sets ATN in low, and NDAC out low, 'data not accepted'.		
LDA #$3C STA $E821	Sets ATN in low, and NDAC out high, 'data accepted'	LDA #$FD [2] AND $E840 STA $E840	Set NRFD out low (true). 'Not ready for data'.
EOR #$FF STA $E822	Store data (reversed) in output buffer.*	LDA $E840 AND #$41 CMP #$41 BEQ ERROR	Branch taken if both NRFD in is high (false) and NDAC in is high (false). l.e. 'Ready for data' and 'Data accepted' are both true.
LDA #$34 STA $E823	Sets SRQ in low (true) and DAV out true, 'data valid'.		
LDA $E823 ORA #$08 STA $E823	Set DAV high (false). 'Data not valid'.	LDA $E840 AND #$01 BEQ -7	Wait until NDAC in is high. 'Data accepted'.
LDA $E823 AND #$F7 STA $E823	Set DAV low (true). 'Data valid'.	BIT $E84D BVS ERROR	Uses a VIA timer to detect time out. ST=1 if write, 2 if read. [3]
LDA #$3C STA $E823	Sets SRQ in true and DAV out false. 'Data not valid'.		
LDA $E840 ORA #$04 STA $E840	Set ATN high (false). 'Send data, not lEEE commands'.		
LDA $E840 AND #$FB STA $E840	Set ATN low (true). 'Send lEEE commands, not data'.		

This table is intended for use as an aid in understanding disassembled code. Each lEEE location appears in sequence, with the handshaking and control line mnemonics approximately in alphabetical order. Not all the possible permutations and combinations are listed, but those which are occur frequently in CBM ROM. Analogous 6502 code exists within Commodore devices, to handle data transfer from the point of view of those devices.

---

*The equivalent in BASIC to reverse byte X is 255-X.
[2] This variation has of course the identical effect to LDA $E840/ AND #$FD.
[3] In BASIC 4, time out may be ignored by poking 1020 with any value > 127.

IEEE Command Groups

		--- HIGH NYBBLE ---									
		0	1	2	3	4	5	6	7	E	F
	0			0	16	0	16	0	16	0	16
−	1	GTL	LLO	1	17	1	17	1	17	1	17
−	2			2	18	2	18	2	18	2	18
−	3			3	19	3	19	3	19	3	19
LOW	4	SDC	DCL	4	20	4	20	4	20	4	20
NYBBLE	5	PPC	PPU	5	21	5	21	5	21	5	21
−	6			6	22	6	22	6	22	6	22
−	7			7	23	7	23	7	23	7	23
−	8	GET	SPE	8	24	8	24	8	24	8	24
	9	TCT	SPD	9	25	9	25	9	25	9	25
	A			10	26	10	26	10	26	10	26
	B			11	27	11	27	11	27	11	27
	C			12	28	12	28	12	28	12	28
	D			13	29	13	29	13	29	13	29
	E			14	30	14	30	14	30	14	30
	F			15	UNL	15	UNT	15	31	15	31
		ACG	UCG	LAG		TAG		SCG		SCG²	SCG³

ACG (Addressed Command Group) includes:
GET=Group Execute Trigger   GTL=Go To Local
PPC=Parallel Poll Configure   SDC=Selected Device Clear
TCT=Take Control
UCG (Universal Command Group) includes:
DCL=Devices Clear   LLO=Local Lockout
PPU=Parallel Poll Unconfigure SPD=Serial Poll Disable
SPE=Serial Poll Enable
LAG (Listen Address Group) includes UNL=Unlisten All Devices
TAG (Talk Address Group) includes UNT=Untalk All Devices
SCG (Secondary Command Group) holds CBM secondary addresses, except
²Secondary address for CLOSE, ³Secondary address for .OPEN and SAVE.

Subdividing the command byte sets limits on the number of devices controllable by the bus. 31 primary devices are allowed; secondary addressing was introduced to enable extra devices to be connected, according to Fisher and Jensen, so that 31 x 31 = 961 is the absolute maximum. Commodore's use of the secondary address as a means of controlling the primary device is therefore rather unorthodox. What does all this imply in BASIC? Firstly, the OPEN command for devices numbered 4 or more (excluding keyboard, cassettes, and screen) is designed to prepare BASIC for future communication with the IEEE bus. OPEN X,Y,Z,"STRING" makes three entries in each of three tables in RAM, unless these tables are full already. PEEK locations 593, 603, and 613 to take a look at this. If a file has been opened, these locations will typically be 5,4, and 97. OPEN 5,4,1 will give these figures, the first being the 'logical file number', the second the device number - here, 4, a printer - and the third the secondary address with its high nybble set to 6, adding 96. (BASIC 1 has locations 578, 588, 598 instead). If OPEN includes a "STRING" this is sent along the bus and processed by the receiving device: normally this is a disk command, for instance "0:FILE,SEQ,READ" or "#" or "PROGRAM". This of course sets up a similar set of table entries within the disk drive's own RAM. Now when PRINT#X,"MESSAGE" is executed, the device number Y, and secondary address Z, corresponding to X are looked up in the tables. Y has its high nybble set to 2 by ORA #$20, corresponding to LISTEN. ATN is set low (true) and these bytes sent as commands; when ATN is reset high, all further output is ASCII data generated by the PRINT statement and formatted in the normal CBM way. Finally, PRINT# X sends an UNLISTEN to the bus directed at the printer. (It will also send UNTALK if there is an IEEE output device too; but usually the keyboard or screen provides output). Note that OPEN and CLOSE have special secondary addresses allocated to them, as appears in the final two columns of the table above. This is the reason for ROM routines like this: LDA secondary address/ ORA #$F0. Again, it's a peculiarity of Commodore that these control bytes are interpreted this way by CBM equipment.

Program Examples with the IEEE bus.

[1] **CMD.** This BASIC keyword uses identical syntax to PRINT#, and operates in a very similar way, the only difference being that the device is not UNLISTENed.* For this reason it is used to keep open a file to disk or tape when a program is to be LISTed as a sequential file. Generally, PRINT# is easier to use, unless a lot of PRINT statements have to be changed, since UNLISTEN or other IEEE command may be issued by some other part of BASIC. CMD may be worth trying if data is to be sent simultaneously to several destinations. In the same way that OPEN4,4: CMD4:INPUT"NAME";N$ within a program prints out NAME to the printer, not the screen, several files may be opened and printed to simultaneously; try for example OPEN1,1,1:CMD1:OPEN3,3:CMD3:OPEN8,8,8,"0:FILE":CMD8:PRINT "HELLO" which prints HELLO to tape #1, screen, and disk at one time.

[2] **ATN.** (Not arctangent!) Setting the ATN line low, sending a command, and setting it high again may be used to direct data to a recipient device. As an example, consider C Brannon's 'Keyprint' program to print the screen contents to a Commodore printer (showing graphics and other Commodore features). The aim is to tell device 4 to listen, then set ATN high, then to output characters one at a time, e.g. with $FFD2. When the page is finished, $FFCC, the routine to UN-LISTEN the printer, is called, and control returned to the interrupted program. ATN out is bit 2 of $E840. Unfortunately it is not enough to use machine code to load the IEEE output buffer with #44 - the talk command for device 4 - with its bits reversed, then lower ATN and put it high again, since this ignores the bus' handshaking. The easiest method is to use ROM routines, although this has the drawback of causing the program to be untransferable between different ROMs without a few changes. LISTEN shares the same ROM area as TALK and in fact these routines are the very first in the F000-FFFF ROM. To cause device 4 to become a listener, the current device location ($F1 in BASIC 1, $D4 in BASIC>1) must contain #4, then LISTEN is called. ($F0BA in BASIC<4, $F0D5 in BASIC 4). Now ATN has to be set high. A routine which does this (i.e. sets bit 2 of $E840 high) exists at $F132 (BASIC 1), $F12D (BASIC 2), or $F148 (BASIC 4). Also the current CMD location has to be set to #4, so that $FFD2 outputs its characters to the correct device. This location is $0264 in BASIC 1 and $B0 in BASIC>1. So with BASIC 4:

```
LDA #$04 ; DEVICE NUMBER 4=PRINTER
STA $D4  ; CURRENT DEVICE
STA $B0  ; CMD LOCATION (CURRENT OUTPUT)
JSR $F0D5; 'LISTEN'
JSR $F148; PREPARE FOR DATA OUTPUT
...      ; PRINT CHARACTERS WITH $FFD2
JSR $FFCC; SEND UNLISTEN
...      ; CONTINUE
```

In this example, a secondary address was not sent. It could easily have been; 5 bytes prior to the second subroutine, which sets ATN false, is the entry point from which the contents of A are output to the IEEE before ATN is set false.

[3] **The PET as controller.** The first published account of spooling with the PET seems to be T M Peterson's article in Compute! (Vol.3, #1 and reprinted in CCN and Transactor). This method may not be foolproof. Jim Butterfield, by coincidence in the same issue of the magazine, wrote that the logic is not accurate enough for spooling to be possible. Peterson's method, for BASICs 2 and 4, is as follows:-

---

*The commands LISTEN, TALK, UNLISTEN and UNTALK use the imperative voice, so to speak. To make the point clear we can consider human analogies: conversationalist X may say to conversationalist Y,"Listen. I want to tell you that ..." and this use of LISTEN is similar to the IEEE's. So is: "You've got five seconds to talk, or else..." where the recipient of this message is being sent a TALK command. CBM equipment allows only 65 milliseconds (.065 sec), however, before a so-called 'time out error'.

Like all analogies, this one breaks down at some points. The controller ensures that one talker only is allowed on the IEEE bus, although there may be many listeners. In human communication on the other hand, no such restriction holds.

Spooling is a technique used to overcome speed limitations of printers: large computer installations store their output on disk, then later disgorge the whole lot, often at night. And the printers can be used when the processor is working but has no printing to do. In principal this can be done with IEEE devices. The sequence is: (i) Set ATN low (true), so the devices wait for commands. (ii) Send UNLISTEN so that all devices in LISTEN mode no longer listen. (iii) Send TALK to device X; X is now the only talker. (iv) Send LISTEN to device Y, (v) Now, set ATN high (false) again, having set up a listener and a talker. These two devices will now talk and listen until the bus is used for something else. On the PET, suppose we have a sequential file on disk, which could contain data, or a program LISTed as a sequential file. The spooling technique goes like this:

```
OPEN 7,8,9,"0:SPOOL,SEQ,READ": REM FIGURES CHOSEN FOR UNAMBIGUOUSNESS
POKE 165,64+8: SYS 61668 : REM SYS 61695 IN BASIC 4. HIGH NYBBLE 4=TALK
POKE 165,96+9: SYS 61668 : REM SYS 61695 IN BASIC 4. HIGH NYBBLE 6=S.ADD.
OPEN 5,4: CMD 5,;:POKE 176,3:POKE 174,0: REM MAKE PRINTER A LISTENER
```

POKE 176,3 makes the screen the output device, so another program (not using the bus) may be run. POKE 174,0 sets the number of files to 0, so files 5 and 7 are erased. When the spooling is over, POKE 174,10: CLOSE 7 will close the disk file. (Or you can OPEN 7,8,9: CLOSE 7).

[4] Handshaking. The charts of implementation of the IEEE on the PET show each control line (where used) except for SRQ having an 'input' and an 'output' connection. This means that during handshaking, values set by the PET use the 'output' location, but values being tested by the PET use the 'input' location. So the machine code which branches to itself when testing a line always uses an input location, while code which sets a value always uses an output line. This distinction is of course a product of the hardware buffering methods employed. As an example, let's consider the ROM routine which outputs a character on the bus. This is situated immediately after the routine to send TALK and UNTALK, which sets ATN low before dropping into the routine and thus outputting a command. In BASIC 1 it's at $F0F1, in BASIC 2 at $F0EE, and in BASIC 4 at $F109. For copyright reasons it cannot be reproduced here, but the logic can be deciphered into this:-

    i. Set DAV false. (I.e. puts 1 into DAV out's bit in $E823)
    ii. Check if both NRFD and NDAC are false. If so, the program stops with a ?DEVICE NOT PRESENT ERROR. (I.e. uses bits from NRFD in & NDAC in, for the test).
    iii. Put reversed data in the output register $E822.
    iv. Wait until NRFD is true.
    v. Set DAV true, and start the clock in the VIA.
    vi. Wait until NDAC becomes false. If this doesn't happen before the timer clocks up 65536 microseconds, the status flag byte is set - in fact, #1 is ORA'd into it, which is why ST of 1 means a time out error on write. Note however that BASIC 4 has a patch put in which enables this time out feature to be disabled. POKE $03FC (1020 in decimal) with any value greater than 127 to make the device wait indefinitely until the data has been accepted. Commodore could, but didn't, include an option allowing the user to select his own time-out interval.
    vii.DAV is set false.
    viii. Finally, the output register is set null (with #FF!).

This is the three-line handshake as implemented on the CBM, using the three lines of the data byte control bus. Hewlett-Packard will supply details of this handshaking procedure. This, however, is approaching the hardware side of CBM, which is not my intention. Before leaving this topic let's briefly see how to write one's own handshaking routines for this bus. In view of the opportunities, there seems to be a surprisingly small amount of published work on device control with the PET/CBM. One popular set of routines, by John Cooke, has appeared in Commodore publications, Fisher & Jensen, Gregory Yob's articles, and, without acknowledgement, in the 'PET Revealed', and this is about all. However, provided

the details of the handshake are known, there should be little difficulty in writing
routines which carry out the equivalent machine code. As a simple subexample,
suppose we wish to set NRFD false, wait until DAV is true, then recover data
from the input register. We look up these facts:

     i. NRFD out is bit 1 of $E840.
     ii. DAV in is bit 7 of $E840.
     iii.The input register is $E820.

And the corresponding machine code might be:-

```
      LDA $E840
      ORA #$01   ; FORCES BIT 1 HIGH (IE FALSE)
      STA $E840  ; NOW, NRFD IS FALSE.
LABEL LDA $E840
      BMI LABEL  ; LOOPS UNTIL HIGH BIT 0 (TRUE)
      LDA $E820
      EOR #$FF   ; REVERSE DATA; NOW IN USUAL FORMAT
```

[5] The status byte and ST. ST (also appears in the BASIC keywords reference
section) is reserved in BASIC, so PRINT ST yields a value often zero, but, if not,
providing information on a read or write transaction on the IEEE bus or with the
cassette tapes, which don't use the bus, but are programmed to look similar for
consistency. ST is not a normal variable held in RAM. Instead, when ST is found
in a BASIC statement, the value held in a single byte is found and converted to
ST, which therefore can't (normally) exceed 255. This byte is $020C in BASIC 1,
and $96 in BASIC>1. Confining ourselves to IEEE transactions only, ST has only
4 values apart from 0, which are

     ST=1 Time out error on write
     ST=2 Time out error on read
     ST=64 End of message
     ST=-128 Device not present.

These messages vary in value. ST =-128 may in BASIC cause the program to crash
anyway. The time out errors can be useful; ST=2 shows the data hasn't been read
although this may be obvious from the data itself. ST=1 is a bit incalculable. For
example, some newer CBM printers give this 'error' even when working correctly.
And ST=64 can often be made redundant by the use of an end-of-file marker. In
any case, EOI may not be reliable with some devices. However, in machine code,
routines to read disk files often use ST's byte location as an easy test for end-of-
file. If it is non-zero, the file is presumed to have been read completely.

[6] ROM routines for use with the IEEE in data transfer. When using disk, modem
or printer, the handshaking is taken care of, and best left alone. But the ROM
subroutines for processing data in machine code are of interest, providing as they
do the possibility of faster data processing than is available with BASIC. All the
'kernel' ROM routines (those in common between all the CBM ROMs) operate with
the IEEE and are often quite easy to use. Important RAM locations are:

Length of message (e.g. "0:PROG" has length 6)	$D1 in BASIC>1, $EE in B.1.
Logical file number	$D2 in BASIC>1, $EF in BASIC1.
Secondary address	$D3 in BASIC>1, $F0 in BASIC1.
Device number (primary address)	$D4 in BASIC>1, $F1 in BASIC 1.
Input device number, for input	$AF in BASIC>1, $0263 in B.1.
Output device number, for output	$B0 in BASIC>1, $0264 in B.1.

As an example, consider a machine-code routine to read CBM sequential files. We
can open the file from BASIC, then read with machine-code: OPEN 2,8,3,"DATA" to
read sequentially from the default device, for instance. Then LDX #02/ JSR $FFC6
sets the device for input to the CBM, and JSR $FFCF inputs a single byte from the
device. When reading is complete, JSR$FFCC closes the file. To open a file from
machine-code requires that the parameters in the table above are set, and that
GETCHR points to the start of the string. Then JSR $FFC0 calls the OPEN routine
used by BASIC. IEEE routines themselves can be called, although the resulting
code is not transferable between BASICs. For example, in BASIC 2, if a file is
open, LDA #08/ STA $D4/ JSR $F0B6/ LDA #$63/ STA $D3/ JSR $F128 performs two
functions, firstly setting device #8 (the disk drives) to talk - a file is presumed to
be open - and outputting the secondary address 3. Now, JSR $F18C inputs a single
character along the IEEE bus. This method is used, with secondary address 15,

by 'Universal Wedge' for DOS, to read characters from the error channel. Commands may be send to the disk using the error/command channel with secondary address 15, which of course has to be OPENed with OPEN 15,8,15 or some other logical file number early in the proceedings. This routine illustrates the method:

```
        LDA #$08    ;DEVICE NUMBER (PRIMARY ADDRESS)
        STA $D4     ;STORE IT.
        LDA #$6F    ;SECONDARY ADDRESS OF 15 (HAS HIGH NYBBLE = 6)
        STA $D3     ;STORE IT, TOO.
        JSR $F0D5   ;SEND 'LISTEN'. (THIS IS BASIC 4. BASIC 1=$F0BA, BASIC 2=$F0BA).
        LDA $D3     ;LOAD SECONDARY ADDRESS
        JSR $F143   ;SEND IT; ALSO  SET ATTENTION LINE HIGH (FALSE). THIS IS BASIC 4;
LABEL   LDX #$00    ;BASIC 1=$F12C, BASIC 2=$F128.
        INC $77
        LDA ($77,X) ; LOAD NEXT CHARACTER IN BUFFER FROM $0200ff.
        BEQ EXIT    ;ZERO BYTE MARKS END OF COMMAND STRING
        JSR $F19E   ;HANDSHAKE THE BYTE OUT. (BASIC 1=$F167, BASIC 2=$F16F)
        JMP LABEL   ;CONTINUE LOOP, OUTPUTTING CHARACTERS.

EXIT    JSR F1B9    ;SEND 'UNLISTEN'. (THIS IS BASIC 4. BASIC 1=$F17E, BASIC 2=$F183).
```

Other features, notes, and bugs related to the IEEE bus.

[1] Functions not implemented by CBM. A large number of IEEE functions don't exist on the CBM, but can be programmed along the lines already discussed. It appears from Fisher and Jensen that *any* function can be programmed (pp.135 ff.). Presumably this can only be accomplished after hardware modifications should a function require the use of one of the interface lines not currently wired for the purpose. These lines are the IFC line, the REN line, and the SRQ line, which is wired for input only. IFC (interface clear) is a reset line; on switchon it is set low as a hardware process. REN (remote enable) is grounded, hence 'true', to retain CBM control over the devices. SRQ (service request) for the same reason is not wired for output from the CBM, which is the controller.

[2] Bugs. BASIC 1, not surprisingly, has a number. LOAD, SAVE and VERIFY don't work properly with disks (and have tape bugs too). The hardware connections to the PIAs and VIA cause some problems because of interactions. When the screen scrolls $E811 was poked to blank the screen; this also sent an EOI out. This bug was carried over into BASIC 2. BASIC 4, as we've seen, has a special location to enable the time out feature to be switched off; 65 milliseconds was in any case an arbitrary figure. If it is off, though, the stop key is the only exit should a device not respond. All ROMs prior to BASIC 4 have a bug in their UNTALK/ UNLISTEN routine ($FFCC), where ATN is not set low when #$5F, the command for UNTALK, is sent. G Huckell (Compute! Jan.'81) wrote that a device may become wrongly enabled, or single characters lost or treated as commands, because of this. When using $FFCC, therefore, on earlier ROMs, it is advisable to set ATN like this:

```
        LDA $E840
        AND #$FB
        STA $E840; ATN NOW LOW (TRUE)
        JSR $FFCC
```

Huckell also wrote that CBM equipment is 'immune' from this problem. Probably only those users who are trying to connect other IEEE equipment need concern themselves with this. Relocation of some ROM routines into RAM may be the best way of actually writing in the patch.

14.8 PIAs and VIA.

These three chips control the keyboard, cassettes, screen, IEEE bus, and user port. Both are 40-pin devices with two ports, invariably called A and B. The ports are independently controlled and (apart from certain small differences) almost identical, so each chip can be considered to be made up of two similar halves. The PIA or 6520, ('Peripheral Interface Adapter'), is memory-mapped into four bytes, of which two are ports; the VIA or 6522 ('Versatile Interface Adapter') occupies 16 bytes, of which three are ports, two of them alternatives of port A. In all PET/CBMs they occupy these locations:

```
PIA 1 ... E810 - E813    ... 59408 - 59411
PIA 2 ... E820 - E823    ... 59424 - 59427
VIA   ... E840 - E84F    ... 59456 - 59471
```

[CRT controller in 12-inch screen CBMs ... E880 - E881].

The peripherals are not wired in a very systematic manner, as the section on the IEEE bus showed. Before describing the various parts of these chips and explaining their programming, I draw the reader's attention to the following program. This is written for BASIC 2, and is a relocatable routine which loops, displaying eight-bit byte patterns on the screen, with their addresses. The range of addresses is controllable by changing the marked bytes. And different ROMs can be catered for: disassemble the routine and change the four ROM addresses from BASIC 2 to BASIC<>2, with the help of Chapter 15:

```
E775 Print A as two nybbles.
FDCA Print two spaces.
FDCD Print one space.
FDD0 Print carriage return.
[FFD2 and FFE1 are kernel
routines].
```

```
.: 033A A9 13 20 D2 FF A9 E8 85
.: 0342 02 A9 40 85 01 A5 02 20
.: 034A 75 E7 A5 01 20 75 E7 20
.: 0352 CA FD A0 00 B1 01 85 00
.: 035A A9 30 06 00 90 02 A9 31
.: 0362 20 D2 FF C8 C0 08 F0 09
.: 036A C0 04 D0 EC 20 CD FD D0
.: 0372 E7 20 D0 FD E6 01 A5 01
.: 037A C9 50 D0 C9 20 E1 FF B0
.: 0382 B7 00
```

DISPLAY BYTES FROM $E840 - $E84F.

This breaks to the monitor on Stop; put #$60 into $0383 to return to BASIC. The loop repeatedly homes the cursor and displays the VIA's contents, so the timers for example can be seen moving. If the loop is removed and the routine prefixed by SEI, it can be incorporated into the interrupt to give a continual display of the chip's contents. Either PIA can be watched when required.

14.8.1 The PIA.

This chip, though simpler than the VIA, is nevertheless considerably complex. Let's look at its features and the names and abbreviations given to each of them. First, we have the two *ports*, A and B. These are 8 bits held in a single byte or *register;* the individual bits are referred to as *PA0* to *PA7* in port A, and *PB0* to *PB7* in port B. Each bit can be configured for either input or output; very often all 8 bits are configured identically. Peeking or poking, and the machine-code equivalent, is used to take data from the registers and write it into the registers respectively. The registers may be called *I/ORA* or *I/ORB*, input/output register A or register B. Each of the two ports has two *control lines;* each occupies one pin, so the ports have 10 bits each if these lines are used. Port A has control lines *CA1* and *CA2*, and port B has *CB1* and *CB2*. CA1 and CB1 are usable only for input; CA2 and CB2 may be defined either for input or output. The ports therefore occupy two bytes of the memory-map; the other two bytes, which include flags to check the status of the control lines, are called *control registers*. CRA ('control register A') and CRB ('control register B') correspond, unsurprisingly, to ports A and B respectively. When the control register is appropriately set, its port no longer receives or sends data, but is treated instead as a *data direction register*, the pattern of bits being loaded into it defining its bits as inputs (when bit = 0), or outputs (when bit = 1). *DDRA* and *DDRB* are the data direction registers for ports A and B. Their locations are the same as those of the ports; bit 2 in the control register determines whether the register is currently treated as a data direction register or a port. When bit 2 is zero, DDR is assumed, and the value in the direction register when bit 2 rises to 1 defines which bits will be treated as input and which as output until the system is redefined. This arrangement is economical, if a little confusing. Note that on switching on, the chip's internal mechanism sets all registers to zero, so that a data direction is assumed in

which all bits are inputs. This prevents hardware connected to the system from being turned on with the computer. PIAs have two interrupt request lines, *IRQA* and *IRQB*. These are normally high, but may be programmed to become low when a change is detected in CA1, CA2, CB1, or CB2. All these interrupts, if they occur, can be distinguished by software; in the PET/CBM, only one interrupt (corresponding to the screen refresh) is enabled, so the system of flags in the control registers is not used. At least, this is generally true, with the exception of tape operations, which detect and control the cassettes by means of interrupts and the control-line CA1 on PIA1 and CB1 on the VIA. Interrupts can be defined to take place upon either of two types of *transition:* 'active low' means that a transition from high (1) to low (0) triggers the interrupt, and 'active high' means the opposite - that the relevant control-line must rise from low (0) to high (1). If a triggering transition takes place, it is called an *active transition*. Transitions in the other direction are not active. Even if the interrupt request enable is off, a flag is set in the chip whenever an appropriate transition happens; these flags cannot be turned off in the normal way, with a poke, but instead a peek is used! Reading the data from the register (i.e. the port) resets the interrupt flags. The control lines are intended for use in handshaking applications and, as we'll see, the IEEE commands described in the previous section are of this type. Note that the control lines are *not* present as bits in the PIA. The programmer can tell the chip to set one or other control-line for output, and set it low or high, or detect changes in an input, but there is no bit in either register which *directly* reveals any control-line's current status. This is a rather confusing point until it is understood.

Having made a stab at a verbal explanation, let's look at the same material diagrammatically in the hope of reinforcing whatever learning may have taken place. We'll consider PIA 1; PIA 2 is internally identical, but has different RAM locations and, because it is connected differently, has other functions than those of PIA 1:

RAM ADDRESS:		BITS:	7 6 5 4 3 2 1 0	
E810 (59408)	CA1 [INPUT] + CA2 [I/O] +		☐☐☐☐☐☐☐☐	PORT A or DDRA
E811 (59409)			▣▪▪▪▪▪☐▪▪☐	CONTROL REGISTER A
E812 (59410)	CB1 [INPUT] + CB2 [I/O] +		☐☐☐☐☐☐☐☐	PORT B or DDRB
E813 (59411)			▣▪▪▪▪▪☐▪▪☐	CONTROL REGISTER B

The ports. As we've seen, these are relatively straightforward. Bit 2 of either control register switches its own port between a data direction register when the bit is zero, to a port when the bit is 1. Example: how do we configure port A for output of all 8 bits? First, bit 2 of CRA must be set to zero; then #FF is stored in DDRA; then bit 2 of CRA is reset to 1. This program (or the BASIC equivalent) will do the trick:

     LDA E811/ AND #FB/ STA E811/ LDA #FF/ STA E810/ LDA E811/ ORA #04/ STA E811*

In the same way #00 configures port A for input on all 8 lines, #AB configures bits 2,4, and 6 for input and the rest for output.

The control registers. We can ignore bit 2, which we now know about. The only other function of the control registers is control of the lines CA1 and CA2 (by CRA), and control of CB1 and CB2 (by CRB). CA1 and CB1 are for input only, and have one fewer controlling bit than CA2 and CB2, 3 bits as opposed to 4. The diagram shows how the seven bits are divided. Bits 7 and 6, the two high bits, are interrupt flags of control - lines 1 and 2 respectively; the BIT instruction can test both, which helps explain the fragmented layout.

Control lines as inputs. When bit 5 of a control register is zero, control-line 2 is configured for input. This option is not available for control-line 1, which is always an input. We can deal with these situations together, because each line is controlled in the same way when control-line 2 is an input. Remembering that bits 6 and 7 are flags, not controllable by direct poking of data, we have only bits 0,1,3, and 4 left. Of these, bits 0 and 1 control line 1, and bits 3 and 4 control line 2. Their effects are to set the direction of active transition, and to enable or disable the interrupt request line. (The interrupt flags are always set on active transition, but IRQ need not be). This table summarises the situation:

BIT NUMBER:	1 [CTRL-LINE1] or 4 [CTRL-LINE2]	0 [CTRL-LINE1] or 3 [CTRL-LINE2]
BIT SET TO 0:	Sets active transition negative	Disables IRQ output
BIT SET TO 1:	Sets active transition positive	Enables IRQ output

*Note: this program is an illustration only; it may be inadvisable to reconfigure PETs.

Control-lines CA2 and CB2 as outputs. When bit 5 of either control register is set high, CA2 or CB2 respectively (or both) become configured for output. The two ports are now somewhat asymmetrical if handshaking is used. If it isn't, both ports behave identically. Let's look at this situation first:

BIT 5=1 [CTRL-LINE 2 OUTPUT] and BIT4=1 ['MANUAL OUTPUT' WITHOUT HANDSHAKING]

Now, BIT 3 HIGH SETS CONTROL-LINE 2 HIGH;
     BIT 3 LOW SETS CONTROL-LINE 2 LOW.

This is the so-called manual output, where CA2 or CB2 can be set high or low as the programmer pleases.

Output with handshaking is the most complex option:

BIT 5=1 [CTRL-LINE 2 OUTPUT] and BIT 4=0 [OUTPUT WITH HANDSHAKING]

Now, control-line 1 is configured (as always) for input, and control-line 2 for output. It is this which makes handshaking possible, the input bringing a signal from the device, and the output line sending a signal. The port itself can be used either to read or write, and CA2 handshakes on reading, CB2 on writing. That is, CA2 is used with LDA and similar instructions, CB2 with STA-type instructions. The sequences are these:

BIT 3 LOW with CA2: CA2 is now controlled by two events:
                             (i) CA1 active transition sets it high,
                             (ii) <u>Read</u> operation sets it low.

BIT 3 LOW with CB2: CB2 is controlled by two events:
                             (i) CB1 active transition sets it high,
                             (ii) <u>Write</u> operation sets it low.
BIT 3 HIGH: Causes 'pulse output', CA2 or CB2 going low for one cycle only after read or write operation. (This pulse may be too short for some uses)

Before looking at examples from the PET, the reader might like to examine this summary diagram of the PIA, which includes most of the features mentioned. If it seems rather confusing, please don't blame me!

7	6	5	4	3	2	1	0	PORT A or DDRA.
CA1 active transition flag 1=on 0=off	CA2 active transition flag 1=on 0=off	CA2 <u>direction</u> 1=out	CA2 Hand-shake=0	Control on Read=0 Pulse = 1	Port A <u>control</u>: DDRA=0 IORA=1	CA1 Active: High=1 Low =0	Control IRQ on = 1 IRQ off= 0	CONTROL REG- ISTER A.
			Manual=1	CA2 high=1 CA2 low =0				
		0=in	Active: High=1 Low =0	IRQ on = 1 IRQ off= 0				

Port B and DDRB are identical, except that CRB = xx10 0xxx implies read handshake.

Examples. We can follow the reset vector from ($FFFC) in ROM to find how the PET/ CBM initialises its PIAs. Diagrams on the next page show how the ports are connected, and the uses of the control lines, and these may be compared with the initialisation logic to see how each PIA works. Considering PIA 1 first, ignoring other I/O chips:

```
RESET ... ALL REGISTERS NOW HOLD 0 ...
     LDA #0F
     STA E810   ;THIS IS CURRENTLY DDRA, SO WE HAVE FOUR INPUTS AND FOUR OUTPUTS
     LDA #3D
     STA E813   ;DDRB IMPLICITLY LEFT WITH #0, I.E. ALL INPUTS. CONTROL REGISTER
                ;B ENABLES CB1 INTERRUPT WITH ACTIVE LOW, AND SETS CB2 HIGH
     BIT E812   ;SEEMS TO BE INTENDED TO CLEAR INTERRUPT FLAGS IN E813
     LDA #3C
     STA E811   ;SWITCHES TO PORT A FROM DDRA; DISABLES INTERRUPTS; SETS CA2 HIGH
```

Thus, bits 4-7 in port A are for input, and bits 0-3 for output; this last batch of bits is used with port B (configured for output) when reading the keyboard during

the interrupt processing sequence. Note that an interrupt is enabled on a CB1 transition. This is the interrupt which drives the keyboard processing. Now let's look at the second PIA, again ignoring the other chips' initialisation:

```
RESET ... ALL REGISTERS NOW HOLD 0 ...
      LDX #FF
      STX E822    ;THIS IS CURRENTLY DDRB FOR THIS CHIP, SO IT'S CONFIGURED FOR
                  ;OUTPUT ON ALL 8 BITS
      LDA #3C
      STA E821    ;DDRA IMPLICITLY LEFT WITH INPUTS. CA2 IS SET FOR OUTPUT & HIGH
      STA E823    ;CB2 IS SET FOR OUTPUT, AND IS SET HIGH
      STX E822    ;PUT #FF AS OUTPUT OF PORT B, BECAUSE IEEE 'LOW' IS 1 AND V.V.
```

PIA 2 is used only by the IEEE bus. Its programming in ROM reflects this: the input port (A) is read and EORed with #FF to flip its bits, and conversely data is reversed and stored in the output port (B). The output control lines CA2 and CB2 (=NDAC and DAV out) are set alternately high and low by storing #3C and #34 into their control registers, setting bit 3 on and off. PIA 1 has more variety: a search program (e.g. Hunt from Supermon) can track down the forty or so occurrences of ROM calls to addresses E810 – E813, revealing tests for input bits in port A, keyboard reading routines, tape routines which disable the CB1 interrupt and later reenable it, and some CA2 outputs which are relics of BASIC 1.

PIA 1

			INPUTS			OUTPUTS	
E810	59408	PORT A	Diag. sense	IEEE EOI in	Cassette #2	sense #1	Keyboard row (0-9)	
E811	59409	CONTROL REGISTER A	CA1 trans'n flag	[CA2 trans'n flag]	CA2=output to blank the screen =EOI	(old PETs only) out (CBMs)	Port A or DDRA switch	CA1=cassette #1 read line
E812	59410	PORT B				INPUTS		
				Contents of keyboard row				
			(Usually, all bits set, or all but one)					
E813	59411	CONTROL REGISTER B	CB1 trans'n flag	[CB2 trans'n flag]	CB2=output to cassette #1's motor: 1=on,0=off		Port B or DDRB switch	CB1=screen retrace line in

PIA 2

			INPUTS				
E820	59424	PORT A	Input buffer for the IEEE bus				
E821	59425	CONTROL REG'R A	CA1 trans'n	[CA2 transn]	CA2 line = NDAC out	Port A/ DDRA	CA1=IEEE	ATN in
E822	59426	PORT B	Output buffer for the IEEE bus				
E823	59427	CONTROL REG'R B	CB1 trans'n	[CB2 transn]	CB2 line = DAV out	Port B/ DDRB	CB1=IEEE	SRQ in

14.8.2 The VIA.

This input-output chip is another 40-pin device, which includes all the PIA's features as a subset of its own. As we shall see, the arrangement is a little different. The PIA is a predecessor of the VIA, so if the previous section of the chapter has been understood you will be well equipped to tackle the rather greater complexities of the VIA. All this is something of an electronic engineer's specialism, and is not needed in most programming unless it's essential to write or decipher I/O routines. The usual sources of information on chips of this sort are free data sheets from the manufacturer, and in fact many books on the subject quote from these with little attempt at comprehensible explanation. I've tried to present all the important aspects of the VIA in a readable form, starting with a description of the extra registers possessed by the chip, a diagram of the PET/CBM's individual implementation of them, and finally program examples showing how each type of facility is used. The examples are in machine-code: BASIC equivalents are easily written, usually by direct conversion into peek and poke commands operating on decimal addresses.

The VIA has two ports, port A and port B, each of which has a separate data direction register. Port A can be written or read or both from two separate locations; there are two port As. The same data appears in each; the difference is that one has a handshake effect with CA2, the other having no such effect. Rather confusingly, port B appears first in the RAM addresses, followed by the handshaking port A. CA1 and CA2 are control lines for port A, CB1 and CB2 for port B, and, as with the PIA, the CA1 and CB1 lines are always input lines, while CA2 and CB2 may be configured for either input or output. Note that *every* PIA and VIA has *its own* control lines; the similarity of the names should not (although it might) cause you to think that the name 'CA1' say refers to a unique wire somewhere.

The VIA occupies 16 RAM addresses, 12 more than the PIA. As we've seen, there is an extra 8-bit port and two data direction registers to account for 3 new addresses. There are also 6 registers occupied by timers, 1 by the shift-register, and 4 by the control registers, which between them include the PIA's CRA and CRB. We'll see very shortly what these registers do and how they do it, but let's first look at familiar parts of the chip, which resemble the PIA. The ports and data direction registers are similar (but do not need a bit to switch from one to the other), as are the control lines. On reset, values are set zero, and the usual conventions apply with respect to bit settings: A bit value of 1 (a) sets a line high, (b) configures a line for output, (c) defines an active transition as positive (i.e. 0 to 1), (d) indicates that an active transition has occurred, or (e) enables an interrupt to happen when a line receives an active transition. Zero bit values of course mean the opposite. As in the PIA, flags which show transitions cannot be set by pokes, but instead are set only by hardware transitions and cleared only by peeking or poking certain related locations.

The *user port* (the central connector at the back of the machine) is connected to the VIA: pin B (on the underside, second from the left) is CA1; pins B - L are port A; and pin M is CB2. CA1 is an input line which may be used to handshake with port A, which is the reason for its inclusion. CB2 is connected to the shift register, and can be used to deal with serial processing. Of the other control lines, CA2 is responsible for the graphics or lower-case switch in the character-generation, and CB1 is used to signal input from cassette #2.

Taking the new features of this chip in sequence, we have the following:

Timers. The VIA is equipped with two 16-bit *timers*. These are timers 1 and 2, or *T1* and *T2*. Each takes up two 8-bit registers. Each is set for input on power-on, in which mode counting takes place; when a value is loaded into either timer it is set for output, decrementing once every clock-cycle. A maximum cycle of about 1/15th second (from #FFFF to #0) can be timed. When a timer reaches zero an interrupt flag is set, but an interrupt occurs only if it is enabled. Timer T1 has a special feature, namely a *latch*. This is a second 16-bit register which allows a value to be stored until it is moved to the timer proper. When T1 reaches zero, the latched value is reloaded and the process repeated, so the time-intervals between timing-out are variable within a large range, though with a 1/15th second maximum. In this way, T1 takes up 4 bytes, and T2 two. The rule to remember is that reading the low byte of either timer (but not the latch) clears its own interrupt flag; and writing to the high byte clears the flag and starts the timer counting. This means that sequences of interrupts, and *one-shot* interrupts can be used, and that new timer values must be loaded with the low-byte first if exact timing is required, e.g. a lapse of #1234 microseconds exactly.

Ports A and B can also be *latched*, so that on an active transition of CA1, the value in port A is retained indefinitely (or until the next active transition on that pin) and similarly for CB1 and port B. This is useful of course in many input applications where it may be impracticable to continually read the value at a port.

The shift-register. This 8-bit register is connected to CB2. On command, the *shift register* performs 8 shifts, having the effect either of moving out 8 bits singly to CB2, or of inputting 8 bits from CB2 one at a time. The command is analogous to ASL, where CB2 is equivalent to the carry flag and the shifted location (say A) corresponds to the shift register; if this is repeted eight times, the byte contained in the shift register has been output in serial form, one bit at a time. There isn't a command quite analogous to shifting in; LSR takes in a zero bit.

The shift register can be timed by T2 (see the music example in Chapter 9), and at the same rate as the 6502, using the 'phase two clock', $\phi 2$. Alternatively another external clock may time it. This is therefore a versatile register, which - with suitable hardware expertise - extends the user port's usefulness a great deal.

Control registers of the VIA.

The Auxiliary Control Register (ACR) controls the timers, the shift register, and the latch status of ports A and B. The diagram reflects the conceptual arrangement of bits 7 - 0 across the page. The shift register control has three bits, and therefore eight combinations, which explains its apparently excessive prominence. (Note: Timer 1 has effects on bit PB7 of port B; I've ignored them here for simplicity, since they aren't used and are unlikely to be).

	ACR7	ACR6	ACR5	ACR4	ACR3 ACR2	ACR1	ACR0
E84B 59467	TIMER 1 CONTROL		TIMER 2 CONTROL	SHIFT REGISTER CONTROL		PORT B LATCH	PORT A LATCH
	0=PB7 UNUSED	0=ONE SHOT 1=CONTIN- UOUS	0=ONE SHOT 1=COUNT SET NO. OF PB6 PULSES	000=SHIFT REG. DISABLED 001=SHIFT IN BY TIMER 2 010=SHIFT IN BY ø2 011=SHIFT IN, EXT.CLOCK 100=FREE RUN BY TIMER 2 101=SHIFT OUT BY TIMER 2 110=SHIFT OUT BY ø2 111=SHIFT OUT,EXT.CLOCK		0=DIS- ABLED 1=EN- ABLED ON CB1 TRANSN (IN/ OUT)	0=DIS- ABLED 1=EN- ABLED ON CA1 TRANSN (IN)

The Peripheral Control Register (PCR) controls the operating modes of the four control lines CA1, CA2, CB1, and CB2. CB2 and CA2 are allocated three control bits in this register. CB1 and CA1 are allocated one each. This register therefore is very like *both* CRA and CRB of the PIA, but without the interrupt flags (which have been moved to the *interrupt flag register* (below) and the active transition bit for CA1; the switch between DDR and Port is omitted. The interrupt enable flags are also moved, to the *interrupt enable register*. This means that CA2 and CB2 have an extra bit, and its only effect is on the clearing of the interrupt flag, which with pattern 0x0 may be cleared by reading or writing the port, but with 0x1 is cleared only by writing bit 1 into the correct bit of the interrupt flag register.

	PCR7	PCR6	PCR5	PCR4	PCR3	PCR2	PCR1	PCR0
E84C 59468	CB2 CONTROL			CB1 CONTROL	CA2 CONTROL			CA1 CONTROL
	Direct- ion: 1=OUT	Hand- shake=0	on Write =0	Active transn: High=1 Low =0	Direct- ion: 1=OUT	Hand- shake=0	on Read =0	Active transn: High=1 Low =0
		Manual=1	CB2 hi=1 CB2 lo=0			Manual=1	CA2 hi=1 CA2 lo=0	
	0=IN	Active High=1 Low =0	Clear IFR =1 IFR/ORB=0		0=IN	Active High=1 Low =0	Clear IFR =1 IFR/ORB=0	

The Interrupt Flag Register (IFR) and the Interrupt Enable Register (IER).
These registers are symmetrical with respect to each other and can be considered together. The first indicates whether an active transition has occurred, and, if so, which VIA device caused it. It also signals whether an interrupt took place - if the corresponding interrupt was not enabled by the IER, the flag, though set, won't cause an interrupt. IER7 controls the function of the rest of IER: when 0, each bit set to 1 *clears* its corresponding interrupt enable: when IER7=1, each bit set to 1 *sets* its interrupt enable bit.

	IFR7	IFR6	IFR5	IFR4	IFR3	IFR2	IFR1	IFR0
E84D 59469	1=IRQ 0=NO IRQ	TIMER 1 OUT	TIMER 2 OUT	CB1 TRANSN	CB2 TRANSN	SHIFT REGISTER	CA1 TRANSN	CB1 TRANSN

	IER7	IER6	IER5	IER4	IER3	IER2	IER1	IER0
E84E 59470	1=1 ENABLES 0=1 DISABLES	TIMER 1	TIMER 2	CB1	CB2	SHIFT RGR	CA1	CA2

E840 59456 PORT B	DAV in	NRFD in	Retrace in,	Tape#2 motor[2]	Tape data out	ATN out	NRFD out	NDAC in
E841 59457 PORT A*	USER PORT with CA2 handshake							
E842 59458 DDRB (values on set-up shown)	0	0	0	1	1	1	1	0
E843 59459 DDRA	USER PORT DATA DIRECTION REGISTER							
E844 59460 TIMER 1 LO								
E845 59461 TIMER 1 HI	(set to #FF on system power-on)							
E846 59462 T1 LATCH LO								
E847 59463 T1 LATCH HI								
E848 59464 TIMER 2 LO								
E849 59465 TIMER 2 HI								
E84A 59466 SHIFT REG'R								
E84B 59467 ACR (set to #00 on power-on)	TIMER 1 CONTROL	TIMER 2 CONTROL	SHIFT REGISTER CONTROL			PORT B LATCH	PORT A LATCH	
E84C 59468 PCR (set #0C or #0E on power-on)	CB2 CONTROL[3] (USER PORT PIN M)		CB1[3] CNTRL	CA2 CONTROL[3] (GRAPHICS MODE)			CA1[3] CNTRL	
E84D 59469 IFR (set to #00 on power-on)	IRQ on/off	T1 INT	T2 INT	CB1 INT	CB2 INT	SH-REG INT	CA1 INT	CA2 INT
E84E 59470 IER (set to #80 on power-on)	enable/disable	T1	T2	CB1	CB2	SH-REG	CA1	CA2
E84F 59471 PORT A*	USER PORT without CA2 handshake							

*E84F (59471) is the preferred user port register, since CA2 controls screen graphics.
[2]The motor is on when this line is low, and off when it is high.
[3]CA1 is connected to pin B of the user port. Pins B - L correspond to port A, which is invariably E84E. CB2 (connected to the shift-register) also connects with pin M of the user port; square-wave tones (see Chapter 9) use these facts. CB1 signals input from cassette #2. CA2 controls screen graphics: it is configured for output, and, when low, gives lower-case charcters and others. When high, the mode is upper case/ graphics.

Implementation of the VIA in the PET/CBM system.

Examples of VIA programming.

A PET/CBM's VIA contents typically resemble the diagram (right). Note that the act of peeking some registers resets the corresponding interrupt flag if it is set, so IFR may not be accurate. Port A is configured for input, as it is on switching on. Port B is configured for output in the usual way, except that bit 4 is set for output, to enable fast-screen printing with BASIC<4 *only*. Both timers are running; T1 has been set to #FFFF. The shift register holds #FF, in place of the usual #0, having been used for square-wave music. It is at present disabled: ACR is #0, its normal value. PCR holds decimal 12, so the machine is in upper case/ graphics mode. No use is being made of CB2. IFR shows that no IRQ has taken place, and no flags to denote transitions on any of the 7 lines are set, but these would in any case have been cleared by the program, which reads from E840 through E848. IER shows that no transitions will generate interrupts.

	7654	3210	BIT NUMBER
E840	1001	1110	PORT B
E841	1111	1111	PORT A
E842	0011	1110	DDRB
E843	0000	0000	DDRA
E844	1100	1100	T1 LOW
E845	1111	1111	T1 HIGH
E846	1111	1111	T1L LOW
E847	1111	1111	T1L HIGH
E848	0010	0101	T2 LOW
E849	1101	0000	T2 HIGH
E84A	1111	1111	SHIFT REG
E84B	0000	0000	ACR
E84C	0000	1100	PCR
E84D	0000	0000	IFR
E84E	1000	0000	IER
E84F	1111	1111	PORT A

TYPICAL VIA CONTENTS

Programming the ports Port B handles a great deal of IEEE character input and output, in addition to some tape handling. 7 bits are therefore initialised on power-on, and there is little reason to change them via DDRB. Bit 5 can be converted to output mode; in BASICs < 4 this accelerates screen writing, because screen retrace is no longer awaited before a character is poked into screen RAM. Port A is unused by the CBM, although some hardware (e.g. Compu/think) uses it. In principle it is easy to use: the eight bits of port A are connected to the external device, and CA1, also on the user port, signals data transmission by its transition from (say) low to high, which, when detected by the PET/CBM, reads the data from the port, perhaps having latched it. Conversely, CB2 can be configured for output and used to signal that data is ready at the PET/CBM's port. These hardware topics are not within the scope of this book.

Programming the timers Both timers are used by the PET/CBM (although timer 1's latch is ignored). Timer 2 is exclusively used with tape, to time the reading and writing of bits. Timer 1 is used to time out the IEEE response (before setting ST) and also with tape, although not to such an extent as timer 2. (Both timers also contribute to RND with argument zero, but this is rather a marginal use. It does not apply to BASIC 1, which has the wrong addresses for the purpose). There is one more function, namely the timing of the screen-scroll delay in BASICs<4, which uses timer 1 in this way:

```
        LDA #FE     ;SETS HIGH PART OF TIMER 1 TO #FE, IGNORING LOW PART, WHICH
        LDY #08     ; IS SET AT #FF ALREADY ...
  DELAY STA E845    ;... AND ALSO (i) CLEARS T1 INTERRUPT FLAG, (ii) STARTS TIMER 1
  L     BIT E84D    ; COUNT, IN ONE-SHOT MODE. T1 INTERRUPT IS DISABLED.
        BVC L       ;TEST IFR FOR BIT 7 ON, I.E. T1 TIMED OUT
        DEY
        BNE DELAY   ;PERFORM 8 LOOPS. AT EACH LOOP THE TIMER RESTARTS.
```

This was dropped in BASIC 4, as it affects the IEEE bus; an ordinary nested set of loops replaced it. It illustrates these points about VIA timers:

(i) To load a value into a timer, load the low byte first, then the high byte. When this second step occurs, both bytes are transferred from the 'timer' to the 'counter' within the chip, and the countdown begins.

(ii) Starting the timer clears the timer's interrupt flag

(iii) Reading the low register (not performed in the example) also clears the flag.

(iv) T1's latch enables T1's value to be read at any time.

Note that the delay loop takes about 255*8*256 μsecs = ½ second or so. If LDA #x/ LDY #y/ JSR DELAY is used, variable pauses from 16 seconds to thousandths of a second can be generated.

BASIC can be timed, provided the operations aren't slower than about 1/15th second: **POKE 59460,255:POKE59461,255::PRINT PEEK(59460) + PEEK(59461) - K** shows the method, where K is set to print 0. Any BASIC inserted between the colons will be timed by the system's clock and is therefore accurate to 1 microsecond. The value 'K' varies with spaces in the BASIC line.

Programming the shift register We've seen (Chapter 9) how the CB2 pin of the user port can be used to generate tones. Now we can investigate the rationale for this. In BASIC, this gives a tone: **POKE 59466,X:POKE 59467,16: POKE 59464,T: POKE 59465,0** where X<>0 and X<>255. T controls the pitch. 59466 is the shift register. This is loaded with a bit pattern, and the shift register is enabled in free-running mode, each bit shifting on T2 time-out. Finally, timer 2 is started, after loading its low byte with a timing parameter. CBM ROM does not make use of this register.

Programming interrupts Interrupts in T1 and T2 are used in IEEE handling and tape; CB1 interrupts are also used with tape. CBM ROM does not use interrupt signals from CB2, the shift register, CA1, or CA2. We'll look at two examples here: (i) Single-step, and (ii) using a timer to control the keyboard. To program the IER, note that IER7=0 means that all high bits disable the corresponding interrupts (if they are set). For example, LDA #7F/ STA E84D disables all seven interrupts. On the other hand, IER7=1 means that high bits enable interrupts, so LDA #C0/ STA E84D enables T1's interrupt. An IRQ will now be generated when T1 times out. (i) Single-step (e.g. Supermon's) enables T2's interrupt, turns off the screen interrupt (by DEC E813), alters IRQ's vector, and loads T2 with 46 decimal. This value is calculated to time out just as the next machine-code instruction starts. When it does, the interrupt awaits completion of the command, then jumps to the new IRQ address, which first calls a tape routine to reset the timers and screen interrupt to normal, before disassembling the instruction. We can change the rate of keyboard scan, the internal clock, and the cursor flash rate in a similar way, using T1 to generate regular interrupts, if the interrupt processing sequence is moved to RAM and references to E813 deleted (otherwise the screen interrupt also runs). The timer must either be in free-running mode, or restarted with each interrupt.

CHAPTER 15: INDEX TO CBM BASIC ROMS AND RAM STORAGE

PET/CBM MEMORY MAP
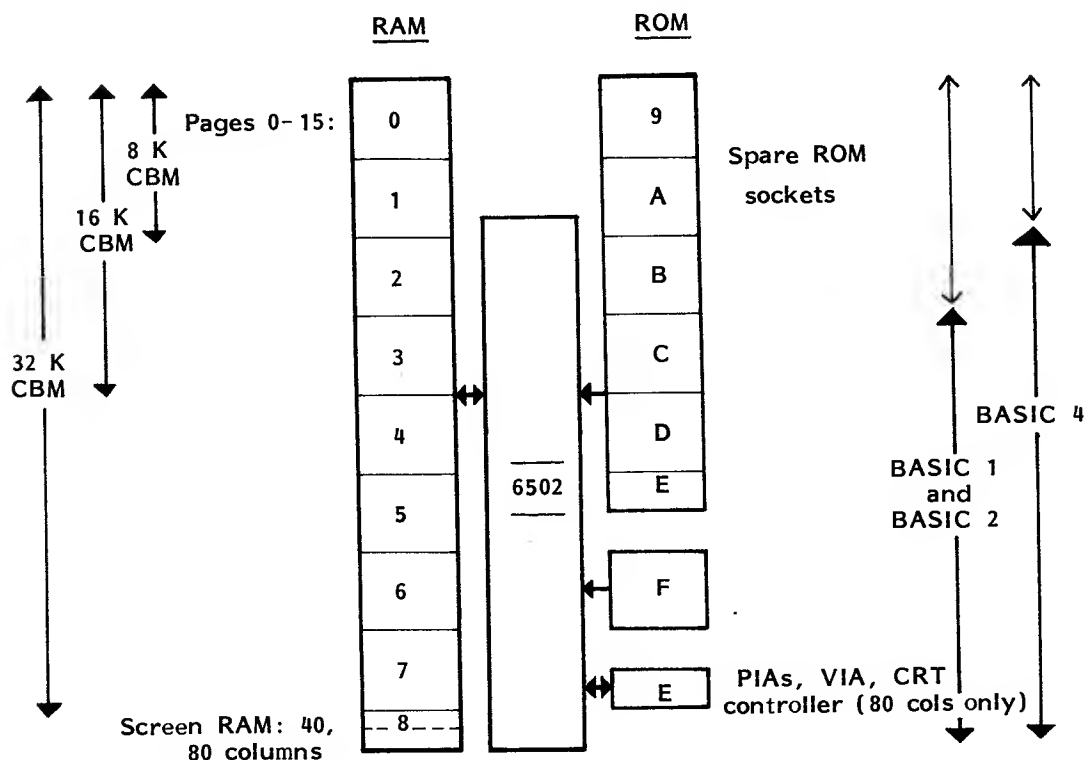
The diagram shows how the CBM's addressable memory of 64K bytes is partitioned between RAM and ROM and hardware input/output. Each full-sized block corresponds to 4K (4096) bytes; the blocks are therefore 0000 - 0FFF, 1000-1FFF, and so on up to F000 - FFFF. Machines with less than 32 K have RAM space available, although increasing the RAM may not be possible or may require the same RAM slots to take larger capacity chips, with hardware modifications (i.e. 'surgery' on the address lines). Spare ROM space is however indicated by spare sockets. These of course are often occupied by 'toolkit'-type ROMs or EPROMs, stored software such as word processors, industrial software in EPROM, and non-CBM devices of various types - disks, video boards.

A description of the BASIC ROMs would be incomplete without mention of the storage areas, buffers, flags and routines which BASIC inevitably needs during its operation. With CBM equipment, this means pages zero to three, stretching from the important zero page to the start of BASIC storage in RAM. The ROMs are very similar to each other in many respects, of which absolute addresses of ROM routines is the major exception. The sequence on the following pages is based on BASIC 4. The guide (next page) showing how the working storage and ROMs are laid out should make the location of most routines fairly easy.

PET/CBM MEMORY MAP: THE FIRST FOUR PAGES

	BASIC 4 (80 COL)	BASIC 2	BASIC 1
0000	USR jump address		
	String pointers		
	Numeral pointers		Input buffer for BASIC lines, direct commands, INPUT, GET
	BASIC pointers		
	Calculation work area	As BASIC 4	
0080	GETCHAR routine		BASIC flags
	RND/ TI/ int. vectors/ ST		BASIC pointers
	Keyboard/ screen/ IEEE		Variables processing
			Calculation work area
	Tape parameters and flags		CHRGET routine
	Window parameters	Screen line table	Screen and file data
0100	ASCII conversion area		
	Tape correction/ check area		
0180	Stack	As BASIC 4	
0200	Input buffer for BASIC lines, direct commands, INPUT, GET.		Clock/ keyboard buffer/ screen/ int. vectors
			Screen line table
	Tables: logical file #/	As BASIC 4	Tables: logical file#/
	device #/ sec. addr./ keybd.		device #/ sec. addr.
0280	buffer.		Tape flags.
	Cassette buffer #1		
		Cassette buffer #1	
0300			
	DOS command string buffer and Cassette buffer #2		
0380		Cassette buffer #2	
0400			

Most of BASIC 4 is identical to BASIC 2; BASIC 2 is fairly similar to BASIC 1, except for the input buffer's move from zero-page to $0200, with the consequent changes in most pointers. Major differences between BASIC 4 and BASIC 2 are (i) the cassette buffer for tape #2 is no longer used solely by tape operations, but by the new disk commands too, and (ii) 80 column BASIC 4 replaces the table of screen line pointers with screen and keyboard parameters. BASIC 4 (40-col) and the 80-column version differ in ROM E000 - E7FF, dealing with screen and keyboard processing.

BASIC1   BASIC2 (&4)   BASIC4

PAGE 0: RAM $0000 - $00FF

BASIC1		BASIC2 (&4)		Description
0-2		0-2		USR jump instruction (default prints 'illegal quantity error'). 0 holds #4C=JMP; ($01)=jump address
$5A	90	3		Offset pointer when scanning for end of statement or line
$5B	91	4		Quotes marker. Is zero when not in quotes.
$5C	92	5		Input buffer pointer/ number of subscripts of an array
$5D	93	6		Default DIM flag/ array name initial/ AND, OR flag
$5E	94	7		Type of variable: #FF=string, #00=numeric
$5F	95	8		Type of numeric variable: #80=integer, #0=floating point
$60	96	9		Flag used in DATA/ LIST/ garbage collect/ memory
$61	97	$0A	10	Flag used for subscripts/ FN DEFinitions
$62	98	$0B	11	Flag with INPUT=#0, GET=#40, READ=#98
$63	99	$0C	12	ATN sign/ comparison evaluation flag
		$0D	13	DS$ length in BASIC 4 only
		($0E)	14-15	DS$ pointers in BASIC 4 only
$64	100	$0D	13	Flag to suppress PRINT or PRINT# when negative
$03	3	$0E	14	$10 16 File number of current I/O device (when non-zero suppresses INPUT prompt etc)
$06	6	$0F	15	Terminal width (unused - carried over from teletype)
$07	7	$10	16	Width of source (unused - carried over from teletype)
($08)	8-9	($11)	17-18	2-byte integer address computed for GOTO, SYS, GOSUB
$65	101	$13	19	Index to next string pointer
($66)	102-103	($14)	20-21	Pointer to descriptor stack for string processing
$68-$70		$16-$1E		Descriptor stack of three temporary string pointers of
	104-112		22-30	the form length then 2-byte pointer
($71)	113-114	($1F)	31-32	Pointer e.g. for memory-move/ for string in memory
($73)	115-116	($21)	33-34	Pointer e.g. for number movements
$75-$79		$23-$27		Intermediate product area for calculation
	117-121		35-39	
($7A)	122-123	($28)	40-41	**Pointer to start of program (usually $0401 = 1025)**
($7C)	124-125	($2A)	42-43	**Pointer to start of variables/ end of program**
($7E)	126-127	($2C)	44-45	**Pointer to start of arrays/ end of variables**
($80)	128-129	($2E)	46-47	**Pointer to start of free RAM/ end of arrays**
($82)	130-131	($30)	48-49	**Pointer to present lower limit of dynamic string storage**
($84)	132-133	($32)	50-51	**Utility string pointer to reserve space for new string**
($86)	134-135	($34)	52-53	**Top-of-memory pointer (e.g. $8000 on power-on with 32K)**
($88)	136-137	($36)	54-55	**Current linenumber/ highbyte=#FF means direct mode**
($8A)	138-139	($38)	56-57	Previous linenumber
($8C)	140-141	($3A)	58-59	Pointer to statement for CONT
($8E)	142-143	($3C)	60-61	Linenumber of current DATA line
($90)	144-145	($3E)	62-63	Pointer to current DATA value (starts at $0400)
($92)	146-147	($40)	64-65	INPUT, READ, and GET vector to save CHRGET
($94)	148-149	($42)	66-67	Current variable name, first character first
($96)	150-151	($44)	68-69	Pointer to variable in RAM; points just after name
($98)	152-153	($46)	70-71	Holds variable name for FOR...NEXT/ WAIT parameters &c
($9A)	154-155	($48)	72-73	Save Y-register/ new operator/ operator pointer etc
$9C	156	$4A	74	Comparison symbol check ; bits 0,1,2 are <,=,>
($9D)	157-158	($4B)	75-76	Pointer to temporary storage in RAM for FN DEF, TAN,&c
$9F-$A2		$4D-$50		Pointer to string, length, and garbage collect constant
	159-162		77-80	
$A3-$A5		$51-$53		Jump vector for function evaluations, consisting of #4C
	163-165		81-83	(=JMP) followed by arithmetic function address
$A6-$AB		$54-$59		Temporary pointers (e.g. in memory move) plus numeral
	166-171		84-89	storage of intermediate results ('Floating point Acc'r #3')
($AC)	172-173	($5A)	90-91	Numeric pointer e.g. in ASCII conversion, series eval'n
($AE)	174-175	($5C)	92-93	Pointers e.g. in LET, search for linenumber
$B0-$B5		$5E-$63		**Floating-point accumulator #1 (most results of evaluations**
	176-181		94-99	**are left here). Exponent, 4 mantissas, and sign bytes**
$B6	182	$64	100	Series evaluation counter of number of items in series
$B7	183	$65	101	Overflow byte on normalizing floating-point accumulator #1
$B8-$BD		$66-$6B		**Floating-point accumulator #2 (used with FPAcc.#1 in eval-**
	184-189		102-107	**uation of products, sums, differences, etc. EMMMMS**

BASIC1		BASIC2 (&4)		BASIC 4
$BE	190	$6C	108	Sign comparison between FPAccs: #0=equal, #FF opp.
$BF	191	$6D	109	Rounding byte for floating-point accumulator #1
($C0)	192-193	($6E)	110-111	Cassette buffer length/ series eval'n/ VAL etc. pointer
$C2-$D9	194-217	$70-$87	112-135	BASIC's CHRGET routine which loads A with the next BASIC character (not space) and sets flags: C clear if ASCII numeral 0-9; Z set if end-of-line or :
$C8	200	$76	118	CHRGOT entry point loads A with current BASIC character and sets flags as CHRGET does.
$DA-$DE	218-222	$88-$8C	136-140	RND number seed and subsequent values; always the previous random number generated
$0200-$0202	512-514	$8D-$8F	141-143	3-byte jiffy clock arranged most significant through least significant bytes
($0219)	537-538	($90)	144-145	IRQ RAM vector, usually E685/ E62E/ E455
($021B)	539-540	($92)	146-147	BRK RAM vector, usually 0000/ FD17/ D478
none		($94)	148-149	NMI RAM vector, usually --/C389/B3FF to print 'ready'
$020C	524	$96	150	Status byte ST, from which ST is computed
$0203	515	$97	151	Which key pressed? (Interpretation may vary with keyboard decoding). #FF=no key
$0204	516	$98	152	Shift key pressed? #0 if no, #1 if yes
($205)	517-518	($99)	153-154	Low and high bytes of 'correction clock' (slows TI)
$0209	521	$9B	155	Contents of E812 for testing Stop key etc
$020A	522	$9C	156	Tape timing constant
$020B	523	$9D	157	Flag for LOAD or VERIFY: #0=LOAD, #1=VERIFY
$020D	525	$9E	158	No. of characters currently stored in keyboard buffer
$020E	526	$9F.	159	Screen reverse flag: #0=normal, #12=reversed
$021D	541	$A0	160	IEEE output flag: #FF=character awaiting output
$021E	542	$A1	161	Count of characters of line input from screen
$021F	543	$A2	162	Not used
$0220	544	$A3	163	Cursor row [also $F5/ $D8]
$0221	545	$A4	164	Cursor column [also $E2/ $C6]
$0222	546	$A5	165	IEEE byte buffer for output (#FF means no character)
$0223	547	$A6	166	Copy of keypress checked by interrupt so that a constant keypress registers once only. #FF=no key
$0224	548	$A7	167	Cursor on/off flag: #0=on, other value = off
$0225	549	$A8	168	Countdown each interrupt for cursor flash
$0226	550	$A9	169	True character at cursor's position
$0227	551	$AA	170	Cursor in blink phase=#1; otherwise =#0
$0228	552	$AB	171	End of tape input flag
$0260	608	$AC	172	Input from screen (#3) or from keyboard (#0) flag
$0261	609	$AD	173	X-register save in tape handlimg (saves cassette #)
$0262	610	$AE	174	Total number of open files (max. 10)
$0263	611	$AF	175	Input device (default = #0, keyboard)
$0264	612	$B0	176	Output device (default = #3, screen)
$0265	613	$B1	177	Tape character parity
$0266	614	$B2	178	Byte received flag
		$B3	179	Temporary save e.g. by DOS wedge
$E9	233	$B4	180	Tape buffer leading chr. (e.g. #5=end of tape)/ MLM
$0268	616	$B5	181	MLM flag, counter/ (B4) points to file name for SAVE
		$B6	182	
$026C	620	$B7	183	Serial bit counter
		$B8	184	
$026F	623	$B9	185	Cycle counter
$0270	624	$BA	186	Tape write countdown
($0271)	625-626	($BB)	187-188	Pointer (0-192 decimal) for use with tape operations
$0273	627	$BD	189	Counter for tape writing and reading
$0274	628	$BE	190	Write byte/ error flag on tape read
$0275	629	$BF	191	Write start bit/ read bit sequence errors
$0276-$0277	630-631	$C0-$C1	192-193	Pass 1 read errors/ pass 2 read errors
$0278	632	$C2	194	Cassette read flags: 0=scan/ 1-15=count/ #40=LOAD/ #80=end of tape marker
$0279	633	$C3	195	Counter of seconds before tape write/ checksum

BASIC1		BASIC2 (&4)		BASIC4
($E0)	224-225	($C4)	196-197	Pointer to screen RAM position of start of current line
$E2	226	$C6	198	Position of cursor along line
($E3)	227-228	($C7)	199-200	Start address for tape LOAD/ utility pointer
($E5)	229-230	($C9)	201-202	End address for tape LOAD
$E7-$E8		$CB-$CC		Constants for tape timing
	231-232		203-204	
$E9	233	$CD	205	Quote flag: #0=direct cursor, else control chrs. printed
$EA	234	$CE	206	Tape read timer flag
$EB	235	$CF	207	End of tape read
$EC	236	$D0	208	Read character error
$ED	237	$D1	209	Length of file name; 0=no name
$EF	239	$D2	210	Current file number
$F0	240	$D3	211	Current secondary address OR'd e.g. with #60
$F1	241	$D4	212	Current device number: #0=keyboard, #1-2=tape, #3= screen, #4 typically printer, #8 typically disk drives
$F2	242	$D5	213	Right-hand of window (BASIC 4)/ length of current line (39 or 79) (BASIC<4 or 40-col. BASIC 4)
($F3)	243-244	($D6)	214-215	Pointer to start of tape buffer #1 or #2
$F5	245	$D8	216	Screen line of cursor
$F6	246	$D9	217	Last key input/ buffer checksum/ temporary I/O store
($F9)	249-250	($DA)	218-219	Pointer to start of file name
$FB	251	$DC	220	Number of keyboard inserts outstanding
$FC	252	$DD	221	Write shift word/ read character in
$FD	253	$DE	222	Number of blocks remaining to read/ write
$FE	254	$DF	223	Serial word buffer
$0229-$0241		$E0-$F8		40 column machines: Table of 25 high bytes of the
	553-577		224-248	RAM addresses of the start of screen lines. (A ROM table holds the corresponding low bytes). Lines which wrap around (i.e. are double length) are flagged.

80 column machines:

$E0-$E2	224-226	Top, bottom, left margins of window
$E3	227	Maximum length of keyboard buffer
$E4	228	Repeat flag: #0=on, #40=off
$E5	229	Repeat countdown
$E6	230	New key marker
$E7	231	Bell timing: #0=off
$E8	232	Counter for two [HOME] keys
($E9)	233-234	Screen input indirect vector ($E11D)
($EB)	235-236	Screen output indirect vector ($E20C)
$ED-$F7	237-247	Unused
$F8	248	Counter to speed TI by 6/5

BASIC1		BASIC2 (&4)		BASIC4
$0207-$0208		$F9-$FA		Cassette flags for #1 and #2
	519-520		249-250	
($F7)	247-248	($FB)	251-252	Pointer for MLM, start of tape address with .S
---		($FD)	253-254	Pointer for MLM, others

PAGE 1 (THE STACK): RAM $0100-$01FF

$00FF-$010F	$00FF-$01FF	Area for conversion of numerals into ASCII string format for printing
$0100-$013E	$0100-$013E	Tape read error log
$0140-$01FF	$0140-$01FF	Stack as used by BASIC

PAGE 2: RAM $0200-$02FF

---	$0200-$0208	MLM area: holds, in sequence, stored values of the program counter high and low, the processor status flags, A,X,Y, the stack pointer, and the IRQ vector.
$0A-$5A	$0200-$0250	Input buffer. Length is 80 characters maximum (plus null byte to terminate string)
$0242-$024B	$0251-$025A	Table of up to 10 file numbers
$024C-$0255	$025B-$0264	Table of up to 10 corresponding device numbers
$0256-$0261	$0265-$026E	Table of up to 10 corresponding secondary addresses

BASIC1	BASIC2 (&4)	BASIC4
$020F-$0218 527-536	$026F-$0278 623-632	$026F- Keyboard input buffer (interrupt driven); 623- length is variable in BASIC 4
$027A-$0339 634-825	$027A-$0339 634-825	Input and output buffer for cassette tape #1
$027A	$027A	Type of tape file
($027B)	($027B)	Start address for load
($027D)	($027D)	End address for load

PAGE 3: RAM $0300-03FF

$033A-$03F9 826-1017	$033A-$03F9 826-1017	Input and output buffer for cassette tape #2
$033A	$033A	Type of tape file
($033B)	($033B)	Start address for load
($033D)	($033D)	End address for load

$033A	826	DOS byte parameter in RECORD
$033B	827	DOS drive number
$033C	828	DOS drive number
$033D	829	DOS length/ write flag
$033E	830	8-bit syntax checking flag
$033F-$0340	831-832	Diskette ID
$0341	833	Length of DOS command string
$0342-$0352	834-850	Buffer for filename
$0353-$0380	851-896	Full DOS command string buffer

40-column BASIC 4 only:-

$03E9	1001	Repeat key countdown
$03EA	1002	Delay between repeats
$03EB	1003	Maximum size of keyboard buffer
$03EC	1004	Bell timing: #0=off
$03ED	1005	Counter to speed TI by 6/5
$03EE	1006	Repeat flag: #0=on, #40=off
$03F0-$03F9	1008-1017	Table of 80 bits to set tabs

80-column BASIC 4 only:-
$03EE-$03F7 1006-1015 Table of 80 bits to set tabs

($03FA) 1018-1019 USRCMD extension vector from MLM; set on power on to print .? in monitor.

$03FC	1020	IEEE 'timeout defeat': when poked negative, ST is no longer set for timeout after .065 second's delay.

PAGE 4: RAM $0400-$04FF

$0400	1024	$0400	1024	Null byte at start of BASIC
$0401	1025	$0401	1025	Start of BASIC storage (unless pointers changed from $0401). Sequence is 2 byte link address; 2-byte line-number; tokenised BASIC terminated by a null byte; and so on, until the end is marked by three consecutive null bytes.

PET/CBM MEMORY MAP: GUIDE TO ROMS

VERSION OF BASIC and ROM starting address			APPROXIMATE CONTENTS
BASIC 1	BASIC 2	BASIC 4	
C000	C000	B000	Keywords and operators with their addresses, and a table of error messages. Stack handling is here and includes routines to check space left on the stack and to search for tokens of GOSUB and FOR. The direct mode processing of commands and of BASIC lines, plus the routines to clear variables and run programs, are all in this ROM: NEW, CLR, RUN. END, STOP, and CONT occur here, with with other system-like commands including LIST, RESTORE, GOSUB, GOTO, RETURN, IF, ON and LET, and the input/output commands PRINT, GET, INPUT, CMD, READ. Also LET is part of this ROM; it is a default keyword. It checks variable types and evaluates expressions.
D000	D000	C000	This ROM performs most of the complex processing required by string and numeric variables. It includes arrays ('subscripted variables'), the garbage collection routine, and string-numeral interconversion routines like STR$. It processes all floating point accumulators and interconverts ASCII with numerals, integers and so on. Most mathematical functions are calculated from here, including PEEK, POKE, WAIT, SGN, ABS, INT, SQR, EXP,RND, COS, SIN.
		D000	BASIC 4 only: processes the additional BASIC commands used by CBM disks: DOPEN, APPEND, HEADER, and so on. Processing for DS and DS$ is partly here, and partly in earlier ROMs. Actually, only about a half of this ROM processes disk commands: the rest is taken from the previous ROM of older versions and from E000ff. of the older ROMs, and includes the MLM monitor.
E000	E000	E000	The first half of this slot (E000 – E7FF) is occupied by ROM; the remainder by a few I/O chips. Print routines, screen processing routines, keyboard and cursor control and similar functions are carried out here. Note that 40-column BASIC 4 and 80-column differ in this part of ROM. Reset and IRQ also come here.
F000	F000	F000	All the tape processing – loading, saving, writing, and reading – is controlled by this ROM. The tape operating system is similar in all the ROMs, apart from corrections made to remove bugs from BASIC 1. Tape is not an IEEE device. Also the input/ ouput for IEEE is carried out from here; this includes OPEN, CLOSE, VERIFY, LOAD and SAVE and also error messages. BASIC 1 has no monitor, but it does have diagnostic routines (which only work if the user port is specially wired up). BASIC 2 has most of its MLM (monitor) here. The 'kernel' jump addresses are here, in the top of memory near the 6502's NMI, Reset, and IRQ vectors.

BASIC1 BASIC2 BASIC4

$C000	$C000	$B000	**5 tables of addresses, keywords, and error messages.** These are (i) Addresses-1 of principal keywords, i.e. those which start a BASIC statement. The addresses are pushed on the stack and RTS executed (via GETCHR) to jump to them; hence the displacement by 1 from the true entry
$C046	$C046	$B066	point. (ii) True addresses of numeric and string functions.
$C074	$C074	$B094	(iii) Addresses - 1 of operators, with a byte to assign their hierarchy: the table corresponds to add, subtract, multiply, divide, power, and, or, negative, not, and comparison. The lattermost function evaluates <, =, and >. Their hierarchy values in hex are: 79,79,7B,7B,7F,50,46,7D,5A and 64.
$C092	$C092	$B0B2	(iv) Keywords with the final character stored with bit 7 high. These include +,-,*,/ etc. which are converted to tokens as
$C190	$C192	$B20D	well as END, FOR, NEXT,... (v) Error messages, stored with the final byte zero as a terminator.

The keywords in each ROM are different; BASIC 2 has GO, which is not present in BASIC1; BASIC 4 additionally has 15 disk commands (including DIRECTORY). A list of each appears in a table in Chapter 2.   The error messages are identical, except that BASIC 1's BAD DATA becomes FILE DATA in subsequent ROMs.

$C2AC	$C2AA	$B322	**Check stack for 'FOR'.** Called by NEXT and RETURN. If Z flag=0 on return from this, FOR has not been found and ?NEXT WITHOUT FOR results. Otherwise the loop variable is checked. Also eliminates FOR when GOSUB token is expected in the stack by RETURN.
$C2DA	$C2D8	$B350	**Open up space in memory.** This routine enables BASIC lines to be merged into BASIC. After checking that there is sufficient RAM, a memory move takes place up RAM.
$C2E1	$C2DF	$B357	In BASIC 2/4:($55)=Top of area to be moved to + 1 ($57)=Top of area to be moved + 1 ($5C)=Bottom of area to be moved $1F =Temporary parameter. In BASIC 1 the parameters are: ($A7), ($A9), ($AE) and $71. On exit, all the pointers are changed.
$C31D	$C31B	$B393	**Check space within stack.** Tests whether twice the byte in the accumulator will fit the stack; if not, ?OUT OF MEMORY is printed. The bottom of the stack allows 62 bytes for other purposes. (I.e. the whole stack is not used as a stack; some is treated as ordinary RAM). So, to fit 10 bytes on the stack, LDA #5 then JSR to this routine tests the space.
$C32A	$C328	$B3A0	**Check for overlap of BASIC strings and variables in RAM.** On input, A and Y hold the address high byte and low byte. If, on comparison with the string pointer there isn't sufficient room in RAM, the intermediate calculation is stored and garbage is collected. If there still isn't room, ?OUT OF MEMORY ERROR is printed.
$C357	$C355	$B3CD	Print 'OUT OF MEMORY ERROR' to the screen - or:
$C359	$C357	$B3CF	Print the error message offset by X from the start of the error message table. Then:
$C37C	$C37A	$B3F0	Restore keyboard input and screen output, reset stack and flags, print "ERROR", and if in program mode, "IN" with the linenumber. Then:
$C38B	$C389	$B3FF	Prints [Return] READY. [Return] and await BASIC line or direct command.

BASIC 1   BASIC 2   BASIC 4

$C394	$C392	$B406	**Await direct or program line from the keyboard.** This calls C468/C46F/B8F6 which puts one line into the input buffer, and on [return] puts a zero terminating byte at the end of its input. After this, the initial character is read by CHR-GET. If this returns carry clear, the initial character was numeric, and the following routine is branched to; otherwise it's treated as a direct mode command, and is tokenised and run by C48D then C6E9/C495 then C6F7/B4FB then B77C.
$C3AC $C3FD	$C3AB $C3FD	$B41F $B470	**Tokenise BASIC program line.** If the linenumber exists, replace it; if it is new, insert the line into BASIC in RAM. Note that the length of the line is stored in $5C/$05/$05. If the line exists, it is erased by a memory move routine at C3EF/C3EE/B462 before dropping through to the line insertion routine. All the variables are erased by CLR and BASIC is rechained (so variable values are lost on editing). Then the previous major routine is called again.
$C430	$C439	$B4AD	**Reset BASIC execution to start; clear; and chain.** This is also called by LOAD when not in program mode.
$C433	$C442	$B4B6	**Rechain BASIC program in memory.** This searches for 0 bytes marking end-of-line, then recalculates the link addresses. Lines longer than 255 cause this routine to hang. BASIC 1 has a different implementation from the other BASICs and is used by the keyboard entry routine.
$C468	$C46F	$B4E2	**Input keyboard line into buffer.** BASIC 1's input buffer starts at $0A; subsequent BASICs start at $0200. Single characters are input fron a 'device' which is usually the keyboard, and stored in consecutive locations in the buffer, until [return] is pressed. Then, a null terminating byte is put into the end of the string and RTS is called. BASIC 4 tests whether the line exceeds 80 characters, and stops with ?string too long error if so. Earlier BASICS use a little routine to fetch a character, based on FFCF, which appears to suppress output if CHR$(15) is read in. This is dropped in BASIC 4.
$C479	$C481		Single character input routine.
$C48D	$C495	$B4FB	**Tokenise the input buffer.** The buffer is processed until a zero byte is found, each recognised keyword being converted into a single byte (with bit 7 set high). ? and " are checked. BASIC 4 uses ($1F) as a pointer to the table of keywords; this table is now too long to be spanned by a single register's offset. This is the routine which can be fooled by eN, fO, nE and so on.
$C522	$C52C	$B5A3	**Search BASIC for a linenumber.** In BASIC 4, ($11) holds the linenumber, low byte first as usual. On exit, carry bit clear means that the line was not found. If it exists, ($5C) points to it. The location pointed to is the start of the link address, i.e. one byte beyond the 0 end-of-line marker. BASIC 2 is identical; BASIC 1 uses ($08) and ($AE).
$C551	$C55B	$B5D2	**Perform NEW.** This has a syntax check to disallow NEW followed by anything other than : or zero byte. It relies on the start-of-BASIC pointers; putting zero bytes into the start of BASIC, and the next byte, then storing start + 2 into end of BASIC. Next GETCHR is loaded with start-of-BASIC - 1. Then the following is executed:
$C56A	$C577	$B5EE	**Perform CLR.** Like NEW, CLR has a syntax check. Its action essentially is to set all the variables' pointers to coincide with the pointer to the end of BASIC, so they are effectively erased. The stack is also reset. And I/O activity is aborted.

BASIC1 BASIC2 BASIC4

BASIC1	BASIC2	BASIC4	
$C59A	$C5A7	$B622	Reset GETCHR to start of program. Adds #FFFF to ($28) and stores the result in ($77). (Compare this with RESTORE to see different programming styles).
$C5A8	$C5B5	$B630	Perform LIST. Full check for parameters, including -.
$C5C9	$C5D6	$B651	List program with no parameter checks. ($11) holds the high line number and defaults to #FFFF with LIST or LISTn-. ($5C) points to the low linenumber; #0401 is its lowest value.
$C5D5	$C5E2	$B65D	List one line of BASIC, i.e. number then text.
$C62B	$C63A	$B6B5	Converts a token in A (i.e. #$80+) into keyword.
$C649	$C658	$B6DE	Perform FOR. This sets up a block of data on the stack. It assigns the loop variable value, then checks the stack for FOR and for 18 bytes of space. Its scans for the end of the FOR statement, and pushes 18 bytes onto the stack:
$C664	$C673	$B6F9	(i) Pointer to following statement, (ii) Current linenumber, (iii) Floating-point value of higher limit, (iv) Value of STEP plus its sign byte, (v) loop variable name, (vi) FOR token.
$C692	$C6A1	$B727	This routine processes STEP by assuming 1 and overwriting this with the true value if a STEP token (#A9) is found.
$C6B5	$C6C4	$B74A	BASIC warm start. This is the controlling loop which runs BASIC statements. It tests the Stop key, updates the CONT pointer (unless in direct mode) and tests for colons or for end-of-line null bytes between statements.
$C6CF	$C6D4	$B75F	This routine exits if an end-of-program 0 is found (so that END isn't needed) and otherwise processes a new line, by incrementing the CHRGET address to point to the start of the next line.
$C6E9	$C6F7	$B77C	BASIC start with CHRGET pointing to BASIC text (not link address). The above routine drops through to here, where GETCHR gets the next BASIC character, the start of a statement, into A, then executes it with the following sub-routine, and loops back to the warm start entry point where CHRGET points to a link address.
$C6F2	$C700	$B785	Perform a BASIC keyword. This routine (i) Returns with nothing done if a colon is found; (ii) Assumes 'LET' by default if a token is not the first character found; (iii) Checks that tokens (i.e. byte with high bit set, therefore with value #$80 +) are within the range of the token table. (If BASIC 4 disk commands are 'run' on BASIC 2, for example, the tokens will be unrecognised). (iv) Lastly, the keyword's address is pushed on the stack. These two bytes are taken from the table at the start of BASIC; they're found by doubling the value (token - #$80) and using this as an offset. Note that BASIC 2 has a patch to test for GO. It checks that GO is followed by TO, then performs GOTO. The actual execution is performed by jumping to GETCHR, so that the accumulator holds the next character of BASIC *and also* the address of FOR or RESTORE or LET or which-ever it may be is made the destination when RTS is reached at the end of GETCHR.
$C70D	$C730	$B7B7	Perform RESTORE. Sets the DATA pointer to start-of-BASIC, as this appears in the pointers, minus 1. In BASIC4 ($3E) holds ($28) - 1; in BASIC 1, ($90) becomes ($7A) - 1.
$C71C	$C73F	$B7C6	Perform STOP, END and break in program. If the carry flag is set (for example, when $FFE1 tests the stop key and finds it pressed) STOP is performed, if Z is also set. If carry is clear, END is performed. Both routines save information for CONT (pointer to BASIC, linenumber) unless in direct mode; STOP prints BREAK IN n, while END skips this to print only READY. The stop key performs STOP, and an end-of-program terminating zero calls END.

BASIC1 BASIC2 BASIC4

$C745	$C76B	$B7EE	Perform CONT. This (i) Rejects CONT if it is followed by something other than an end of statement indication, (ii) Prints ?CAN'T CONTINUE ERROR if the highbyte of the pointer is #0; it's set to this on a syntax error. And (iii) The pointer into BASIC and the then-current BASIC linenumber are restored, and the program continues.
$C775	$C785	$B808	Perform RUN. This has two branches, RUN and RUN n, where n is a linenumber.
$C567	$C572	$B5E9	RUN resets CHRGET to the start of BASIC, then CLRs variables and stack and runs.
$C77A	$C78A	$B80D	RUN n CLRs variables and stack then calls 'GOTO'.
$C780	$C790	$B813	Perform GOSUB. This tests the stack for space to push 6 bytes; if this doesn't elicit ?OUT OF MEMORY ERROR, the following 5 bytes are pushed on the stack: (i) Contents of CHRGET, (ii) Current linenumber, (iii) GOSUB token (#8D). Then it calls GOTO, which changes CHRGET according to the location of GOSUB's linenumber, and finally warm starts BASIC from its new position. The data on the stack is used by RETURN.
$C79D	$C7AD	$B830	Perform GOTO. There are three parts to this routine: the first fetches the linenumber following GOTO, and stored it in ($11) or ($08) in BASIC 1. Then, this linenumber is sought in the program: to save time with long programs, the following linenumber is compared with the sought one and the starting point of the search depends on the result of comparing the high bytes of these lines. Finally, the routine to search BASIC for a linenumber looks for the line and if it's found loads the pointer (less 1) into CHRGET. To clarify the operation, consider this program line: 10000 GOTO 12000, which may be part of a very long program. 10000=39*256 + 16, so the current linenumber is stored as the 2 bytes 0A and 27. 12000=46*256 + 224, which is stored as E0 and 2E. The highbytes are compared, and since 2E exceeds 27 only lines after 10000 are serached. On the other hand, 10000 GOTO 10001 searches BASIC from the start.
$C7CA	$C7DA	$B85D	Perform RETURN. This checks for GOSUB on the stack and recovers the subroutine's details, or prints ?RETURN WITHOUT GOSUB ERROR. (i) The syntax is checked, (ii) The stack is searched (bypassing FOR's variable pointer processing), (iii) If A doesn't hold the GOSUB token (#8D) the error message is printed, (iv) The original BASIC linenumber and pointer are reconstructed, (v) The next statement is found, as an offset in Y, (vi) CHRGET's address is set, so the next statement will execute.
$C7D8	$C7E8	$B86B	Prints ?RETURN WITHOUT GOSUB ERROR.
$C7DB	$C7EB	$B86E	Prints ?UNDEF'D STATEMENT ERROR.
$C79A	$C7F0	$BF00	Prints ?SYNTAX ERROR.
$C7F0	$C800	$B883	Performs DATA. This routine is shared with the end of GOSUB: it's the part which looks for and continues with the next statement, so that DATA 1,2,3: PRINT X ignores the DATA, but carries on at the print statement.
$C7FE	$C80E	$B891	Search for next BASIC statement. Looks for : or null byte.
$C801	$C811	$B894	Search for next BASIC line. Looks for null byte, marking the end of the line. In either case, on return the Y register holds the displacement from CHRGET'S address, which is ($77) or ($C9) in BASIC 1. An interchange routine is used to ensure that a colon within quotes is not regarded as an end-of-statement indication.

BASIC1 BASIC2 BASIC4

$C820	$C830	$B8B3	Perform **IF**. This routine evaluates the expression following IF, and checks that the expression is followed either by GOTO (#89) or THEN (#A7). Assuming this to be OK, the next step is to load the accumulator with the exponent of floating point accumulator #1, in which the result of the evaluation was deposited. When a result is evaluated as zero, the exponent is set 0, and because of the importance of this special case, it is sufficient to test this single byte when finding if a result was 0. At this point the routine branches: a result of 0 means 'false'. In this case the next line is found, and an uncon- ditional branch rejoins DATA at the point where CHRGET is incremented by the offset to the next line. This is the
$C833	$C843	$B8C6	same routine used to perform REM.
			If the result was non-zero, this is regarded as 'true', and the next statement is executed using the keyword processing routine – unless a numeral follows, when 'GOTO' is called.
$C843	$C853	$B8D6	Perform **ON**. (i) Checks variable type and evaluates it, (ii) tests for either GOSUB or GOTO, (iii) repetitively decrements the variable value in $12 (or $B4 in BASIC 1), and works through the list of commas, until the value is reduced to zero. When this finally happens the token is recovered from the stack and the appropriate command carried out by entry into the routine which executes BASIC statements.(If the location never becomes zero, the next statement is performed by default).
$C863	$C873	$B8F6	**Fetch integer (usually linenumber) from BASIC.** This uses shifts, rotations, and adds to multiply consecutive ASCII digits by 10, add the next, and so on until a non-numeric character is encountered. On entry, A holds the value read by GETCHR. If it isn't numeric, there is an immediate return and the number is 0. Note that validation is not complete; this is why 'GOTO 100xxx' is syntactically OK. To use this routine, point ($77) in GETCHR to the start of the number. Then JSR 0070/ JSR B8F6 reads the number into ($11) and leaves ($77) pointing at the first non-numeric character. BASIC 1's GETCHR is ($C9), and numeral ($08).
$C89D	$C8AD	$B930	Perform **LET**. There are three parts to this routine: (i) The variable (X say in X=5) is searched for in RAM, and set up if it doesn't yet exist. (ii)'=' is checked for (its token is #B2) and the following expression or string evaluated. (iii) Floating-point accumulator is moved into RAM *or* pointers are set to the string, depending on the type of variable. This completes the assigning process.
$C8B2	$C8C2	$B945	These are the entry points for the assignment; see VARPTR for an illustration of assignment.
$C8BC	$C8CC	$B94F	Assigns floating-point numbers
$C8CE	$C8DE	$B961	Assigns integers
$C92B	$C937	$B9BA	Assigns strings, except:
$C8DC	$C8EF	$B972	Assigns TI$ (e.g. TI$="123456")
$C91C	$C928	$B9AB	Adds ASCII digit, pointed to by ($1F),Y, to the present contents of floating-point accumulator #1. (Used by the previous routine with TI$).
$C97F	$C98B	$BA88	Perform **PRINT#**. This routine has two opcodes only; the first calls CMD, which is why the syntax of CMD and that of PRINT# are identical. The second jumps to the end of the routine which performs INPUT#. The part of this routine which it executes aborts the file used by CMD and sets the 'current device' to zero. That is,locations #10, #0E, and #03 in BASICs 4,2, and 1 respectively are made zero.

BASIC1 BASIC2 BASIC4

$C965	$C991	$BA8E	Perform CMD. CMD is identical to PRINT#, except that the output device is left as an output device, not cancelled. Thus, future output, even with PRINT, goes to the same device. This is roughly what happens, at any rate. CMD evaluates its parameter (single byte only) and stores it in $62. ($B4 with BASIC 1). The comma is checked for if the statement hasn't ended. The output device is set by FFC9 and PRINT performed.
$C999	$C9A5	$BAA2	Part of a loop which PRINT uses to print a string from memory, then continue with punctuation of PRINT...
$C99F	$C9AB	$BAA8	Perform PRINT. This is the main entry point to PRINT from BASIC. The flowchart of PRINT in Chapter 5 shows what it does. On exit the buffer is reset: $0200 holds #0, X holds #FF, Y holds #1. (BASIC 1's buffer is different - starts at $0A. Note that BASIC 1processing is rather different from later BASICs. BASIC 4 is closely similar to BASIC 2, except that the CMD file is $10, not $0E, and linefeed is not automatic after carriage return).
$C9AB	$C9B8	$BAB5	Test for comma, branch if found.
$C9AF	$C9BC	$BAB9	Test for semi-colon, branch if found.
$C9A3	$C9AF	$BAAC	Test for TAB(, branch if found.
$C9A7	$C9B3	$BAB0	Test for SPC(, branch if found.
$C9BA	$C9C7	$BAC4	Print numeral (after converting to ASCII string).
$C9D8	$C9E2	$BADF	Print CRLF or CR.
$C999	$C9A5	$BAA2	Print string.
$CA27	$CA1C	$BB1D	Print string from memory. From this entry point, if the accumulator A holds the low byte and Y holds the high byte of an address, this routine prints consecutive characters from that location upward until a zero terminator is found. BASIC 4 is reported to insert zero bytes; it may be necessary to write a routine with FFD2 on the lines of this next routine:
$CA44	$CA39	$BB3A	Print a screen format character. BASIC 1 prints cursor right; the others print *either* cursor right (to screen) or space (when some output file exists). BIT is used to separate the alternatives.
---	$CA3D	$BB3E	Print space
$CA44	$CA40	$BB41	Print cursor right
$CA47	$CA43	$BB44	Print ? for error messages - also slipped in.
$CA77	$CA4F	$BB4C	Print error messages for GET, INPUT, and READ. On entry to this routine, a zero page flag (location $0B or, in BASIC1, $62) holds #0 to denote INPUT, #$40 for GET, and #$98 for READ. The routine separates these out; READ and GET both generate ?SYNTAX ERROR and exit to direct mode. INPUT splits according to whether a file is open or not; if not, ?REDO FROM START is printed and GETCHR loaded with the previous linenumber's pointer again. If a file is open, ?FILE DATA ERROR (or BAD DATA in BASIC 1) terminates the program.
$CA9F	$CA7D	$BB7A	Perform GET and GET#. GET is based around FFE4, as might be expected. Its additions include: (i) Testing for direct mode, (ii) where '#' exists, inputting the file number, checking the comma and setting the device for input, (iii) setting the input buffer for one character only with null bytes, (iv) GETting the character and assigning it to its variable, and finally, where an input file was used, restoring the default devices of screen and keyboard. Note that A is loaded with #40 before the GET/INPUT/READ routine processes the single input character. Stored in $0B, this keeps the three processes distinct when necessary.

BASIC1 BASIC2 BASIC4

$CAC6	$CAA7	$BBA4	**Perform INPUT#.** INPUT# relies heavily on INPUT; it adds only input of the file number and a check for the presence of a comma, plus the turning on and turning off of the device on either side of INPUT.
$CAE0	$CAC1	$BBBE	**Perform INPUT.** If a quotation mark is found after INPUT, by CMP #$22, the string within quotes is pointed to and printed - usually to the screen. Direct mode INPUTs are rejected. Now the following routine is called, which, on carriage return, completes input of a line to the buffer, using in fact the same subroutine as BASIC in direct mode. There is a test for ST. If this is 3 (BASIC 4) or 2 in the others, the command is aborted and the next BASIC statement carried out. The scanning and assignment of the parsed input buffer is carried out in the GET/INPUT/READ routines, where INPUT is signalled by #0 in $0B. Note that the INPUT crash, on carriage return, is deliberately programmed in to go to END.
$CB17	$CAFA	$BBF5	**Print ? prompt and put input into buffer.** This is the routine which INPUT uses to get data to the buffer. All data is transferred on carriage return, including commas and colons, which are only distinguished as separators by the parsing routine after this one. User-defined INPUTs can use this routine, omitting the query if preferred, to input and format data in other ways than CBM's.
$CB24	$CA4F	$BC02	**Perform READ.** GET and INPUT share this routine, but are distinguished when necessary by the flag in $0B, which contains #98 with READ. The object of these routines is to scan the input buffer or DATA statements, assigning variables to each syntactically correct chunk of data, and signalling mismatches and other errors.
$CB29	$CB0E	$BC09	INPUT entry point,
$CB2A	$CB10	$BC0B	GET entry point (preceded by LDA #$40).
$CB88	$CB72	$BC6D	Assign string to string variable,
$CBA0	$CB8A	$BC85	Assign numeral to numeric variable.
$CBCF	$CBB9	$BCB4	Scan program for DATA statements; used by READ.
$CBF5	$CBDF	$BCDA	Checks whether pointer is at end of buffer, i.e. for zero byte. If this isn't found it prints ?EXTRA IGNORED - unless there is an active file, in which case no warning is printed.
$CC12	$CBFC	$BCF7	?EXTRA IGNORED crlf and ?REDO FROM START crlf text messages (with null byte terminator).
$CC36	$CC20	$BD19	**Perform NEXT.** NEXT carries out this sequence of operations: (i) If NEXT is alone, ($46) becomes #0000; if not, the variable following NEXT is sought in memory, and A returns set to the low byte of its pointer, Y to the high byte; these are put in ($46). The stack is searched; no FOR, or no matching FOR, gives ?NEXT WITHOUT FOR ERROR. (ii) The current value of the loop variable is added to the step, and the result moved up within RAM. This requires several pointers to be set, e.g. into variable storage in RAM. (iii) The comparison routine is called, which sets A depending on the result. (iv) If the loop is now
$CC92	$CC79	$BD72	finished, another routine deletes the stack entry and checks for a comma. If one is found, NEXT is entered again. (v) If the loop isn't finished, CHRGET and the previously current linenumber are loaded (as they are with RETURN) and the BASIC warm start routine continues the program.

BASIC1 BASIC2 BASIC4

$CCA4	$CC8B	$BD84	Input and evaluate a numeric expression with check for type mismatch. This calls a subroutine - a little further in ROM - which evaluates any BASIC expression, whether string or numeric. Numerals are left in floating-point accumulator #1, and $07 is loaded with a flag: #FF for a string expression, #0 if it was numeric. (The flag is $5E for BASIC 1). Checking for type mismatch is done by the next routine:
$CCA7	$CC8E	$BD87	Checks numeral was input.
$CCA9	$CC90	$BD89	Checks string was input. These routines interlock; the carry flag determines which category of variable will cause ?TYPE MISMATCH ERROR. Note the use of BIT $38 to give an entry point which sets carry. (SEC = #$38).
$CCB8	$CC9F	$BD98	Input and evaluate any expression. This elaborate routine (500 bytes or so excluding other subroutines) parses any string or numeric expression, checking for syntax errors, and on exit leaves the type of expression flag ($07, or $5E in BASIC 1) set to #FF for a string, #0 for numeral. If numeric, the result is left in floating point accumulator #1. From here it can be processed further; JSR CF93 converts it to an ASCII string at the low end of the stack, and JSR BB1D prints this out, for example. (These are BASIC 4 ROM addresses). If the result is a string, on exit from this routine A holds the length, as do $5E and $C8; and the pointer is stored in ($60) and ($C8). BASIC 1's locations are $CC and ($CD9, and $B0 and ($B1) respectively. Note that this routine is *not* an INPUT routine, but takes a BASIC expression from RAM; before calling it, CHRGET must point to its starting byte. It enables complex expressions like 24+VAL("1.23"+X$)*5*(A=NOT B) to be evaluated. In the process, a lot of the stack and many zero-page flags are used. All Microsoft BASICs have a routine of this type. Parsing is by operator precedence in the case of numeric expressions; as the expression is scanned, an operator of greater hierarchical value is pushed on the stack, with the evaluated result from accumulator #1. An operator lower in the hierarchy pops the stack result into accumulator #2, which is then combined with accumulator #1. The routine is recursive. Unexpected ?OUT OF MEMORY ERROR messages may appear with rather complicated expressions, because of the intermediate results on the stack; simplifying into short sub-expressions may cure this.
$CCC3	$CCAA	$BDA3	Push accumulator onto stack and recursively run routine.
$CCD2	$CCB9	$BDB2	Test for >=< and store their combined code in $4A ($9C with BASIC 1).
$CCF1	$CCD8	$BDD1	Process other operators
$CD3A	$CD21	$BE1A	Puts FPAcc. #1 on the stack and performs mathematical operation determined by offset Y and table of addresses.
$CD72	$CD59	$BE56	Pop stack into FPAcc. #2; loads A with exponent.
$CD9D	$CD84	$BE81	Evaluation routine. This looks for ASCII numeral strings, e.g. 123, variables, pi, . - + ", NOT, FN, arithmetic functions, e.g. SGN, ABS, and expressions in parentheses.
$CDBC	$CDA3	$BEA0	Pi as 5 byte floating point number.
$CE05	$CDEC	$BEE9	Check parentheses and evaluate expression within them.
$CE0B	$CDF2	$BEEF	?SYNTAX ERROR if CHRGET doesn't point to).
$CE0E	$CDF5	$BEF2	?SYNTAX ERROR if CHRGET doesn't point to (.
$CE11	$CDF8	$BEF5	?SYNTAX ERROR if CHRGET doesn't point to ,.
$CE1C	$CE03	$BF00	?SYNTAX ERROR and return to READY.
$CE11	$CDFA	$BEF7	?SYNTAX ERROR if CHRGET doesn't point to a byte identical to that in A. If it does, A returns with the next character.

BASIC1 BASIC2 BASIC4

$CE28	$CE0F	$BF0C	Evaluate a variable. This first uses the routine to search for a variable in RAM, returning with A holding the low byte and Y the high byte of its pointer. Strings are not processed, except for TI$ and DS$ in BASIC 4, but all numeric variables - integer, floating-point, TI, ST and, in BASIC 4, DS - are evaluated and the result is stored in floating-point accumulator #1.
$CE3D	$CE2E	$BFAD	Read clock (TI$) and set up string holding result.
		$BFC9	Read DS$ and set up string holding result.
$CE58	$CE43	$BFD8	Evaluate integer variable. Result in FPAcc.#1.
$CE65	$CE82	$C040	Evaluate floating-point variable, not TI, ST, or DS. Note that (i) BASIC 1 uses a set of patches in E19B-E1DF which are moved to be in line with the main code in BASIC 2. (ii) BASIC 4 has a slightly different arrangement, due to the introduction of DS and DS$.
$CE6D	$CE60	$BFF3	Evaluate TI. Result in FPAcc.#1.
$CE8A	$CE7D	$C017	Evaluate ST. Result in FPAcc.#1.
		$C024	Evaluate DS. Result in FPAcc.#1.
$CE79	$CE89	$C047	Process arithmetic functions.
$CED6	$CEC8	$C086	Perform OR.
$CED9	$CECB	$C089	Perform AND. These two binary operations are written as one routine; a flag holds #FF for OR, #0 for AND. See Chapter 5 for the rationale. The flag is location $05, or, in BASIC 1, $5C. Each of the two arguments is converted from floating-point to integer form, with an error message if the range is wrong. Intermediate results are stored in the zero page. The result is left in FPAcc. #1.
$CF06	$CEF8	$C0B6	Perform comparisons. This routine begins by testing that the two items do in fact match in type. It separates into two branches depending on whether numerals or strings are to be compared.
$CF0B	$CEFD	$C0BB	Numeric comparison, and:-
$CF1E	$CF10	$C0CE	String comparison. Numbers are compared with another subroutine (DB2D/ DB67/CD91) after first modifying FPacc. #1 to include the sign bit in the mantissa. The string comparison function works like this:- The first string's parameters are $5E=length, ($5F)=pointer; the second string has its length put in A, and its pointers in ($69). (BASIC 1 is different - B0 and (B1) and (BB) are its equivalents). The X register holds one of three values on exit: X=0 means the strings are equal, X=1 means the first is 'greater than' the second, and X=255 means the second is 'greater than' the first. The accumulator holds only #0 or #255 on exit; this varies with the contents of the comparison evaluation flag.
$CF71	$CF63	$C121	Perform DIM. This routine calls the next routine, which searches for a variable in memory and sets it up if it isn't found. So for example DIM A$(44) sets up the variable A$(44) in memory; and in the process it generates the entire array from elements A$(0) through to A$(44). If the statement has not ended, the routine loops repetitively, checking for a comma, and setting up the next array.
$CF7B	$CF6D	$C12B	Search for variable and set it up if not found. The first half of this routine validates the variable's name: the leading character must be alphabetic, the next may be that or numeric; a loop rejects further alphanumerics; and the variable type flag in $07 is set to #FF if '$' is found, and #0 otherwise; and the numeral flag in $08 becomes #80 if a '%' is found. $0A indicates a function. A '(' causes another

BASIC1	BASIC2	BASIC4	
			ROM routine, 'Find or create array', to be called. Finally, the name is stored in ($42), with its initial character in $42. BASIC 1, naturally, is different - it uses ($94). These are stored with their high bits set according to the type of variable: see Chapter 2 for the four types. So much for the first part of this routine. The second actually looks for the name in RAM. All the variables are stored together after BASIC and before the arrays; moreover they each occupy a total of 7 bytes. So a loop simply compares consecutive
$CFD7	$CFC9	$C187	variables until the sought one is found. If in fact it doesn't exist, the ROM routine 'Create a new BASIC variable' is branched to.
$D005	$CFF7	$C1B6	**Check A holds alphabetic ASCII character.** The carry flag is set to 1 if A holds ASCII A-Z.
$D00F	$D001	$C1C0	**Create a new BASIC variable.** Sets up a new simple (not array) variable in RAM after the present variables. If any arrays are present, they have to be moved 7 bytes up in RAM to accommodate the variable. The array pointers need to be changed, and BASIC 4 string-into-pointers also need to be updated. This can take a second or two. On exit, ($5C) points to the start of the variable, i.e. the first character of its name; ($44) points two bytes forward of this, to the variable's value or pointers if it's a string. All its bytes are set to zero. TI and ST, and DS with BASIC 4, are checked for and give ?SYNTAX ERROR if they've been used on the left of an expression. The same is true of DS$. TI$ returns with a dummy value (null string).
$D088	$D078	$C2C8	**Allocates space for array pointers.** This adds #5 to twice the number of dimensions of an array, and in turn adds this result to a pointer. This makes room for the housekeeping of an array, not for the actual data.
$D099	$D089	$C2D9	Holds -32768.0005 as a 5 byte floating-point numeral.
$D09D	$D08D	$C2DD	**Input and evaluate expression as a positive integer.** This is not part of INPUT; it takes an expression from BASIC, such as PEEK(123)+99, evaluates it, and, if the result is positive and less than 32768, it is converted into a fixed point number held in the two bytes ($61) within FPAcc. #1.
$D0B9	$D0AC	$C2FC	**Find array element or create new array in RAM.** This is rather similar to the routine which searches for simple variables. However, the details of the array are held on the stack, so the more dimensions an array has, the larger is the space used on the stack. And this routine is much
$D100	$D0F3	$C343	longer and more complex. A loop checks for the existence of the subscripted variable; it has two exits, one taken when the variable is not found, and the other taken when it is.
$D149	$D13C	$C38C	Array variable not found; set it up. (DIM=10).
$D135	$D128	$C378	Array variable found.
$D12D	$D120	$C370	?BAD SUBSCRIPT ERROR then READY.
$D130	$D123	$C373	?ILLEGAL QUANTITY ERROR then READY.
$D1F4	$D1E7	$C436	?OUT OF MEMORY ERROR used by next routine:
$D233	$D228	$C477	**Compute size of array subscript.** This loops 16 times, and returns with X and Y holding the size required, X the low and Y the high bytes.
$D135	$D128	$C378	?REDIM'D ARRAY ERROR if the DIM flag ($06 or $5D in BASIC 1) is non-zero.
$D264	$D259	$C4A8	**Perform FRE.** If string mode is on, temporary strings are cleared and 'garbage collect' performed. After this, the pointer to the lowest string minus the end-of-arrays pointer is stored in A and Y, and put into FPAcc. #1:-

BASIC1 BASIC2 BASIC4

$D278	$D26D	$C4BC	Convert 2-byte integer into floating-point. On entry, Y holds the low byte and A the high byte of an integer in the range 0-65535. This routine converts it into floating-point form, leaving it in FPAcc. #1 (i.e. $5E holds the exponent, $5F-$62 hold the mantissa, and $63 is the sign.)
$D285	$D27A	$C4C9	Perform POS. Calls the previous routine, ignoring whatever dummy variable appeared in POS(x). It loads Y with the position of the cursor on its line (from $C6 or $05 with BASIC 1) and the high byte A with #0, then calls the last routine, overwriting the contents of FPAcc. #1 with the
$D287	$D27C	$C4CB	value of POS. **Put contents of Y into FPAcc. #1.**
$D285	$D280	$C4CF	Check for program mode. If the high byte of the current linenumber is #FF, this is a code used to signal that a command was entered in direct mode (i.e. from the keyboard without a linenumber).
$D290	$D285	$C4D4	?ILLEGAL DIRECT ERROR then READY.
	$D288	$C4D7	?TYPE MISMATCH ERROR then READY.
$D295	$D28D	$C4DC	Perform DEF (function definition). Some of the syntax checking is carried out by the next routine. This one tests for direct mode and the presence of a '(', then searches for and/or sets up its dependent variable, checks for ')=', and pushes 5 bytes on the stack. The first byte is the first character of the function definition, perhaps a variable's initial character or a token for LOG or SQR. Then the dependent variable's address and the current pointer into BASIC are stored; when a FN is addressed, the expression to be evaluated is calculated by temorarily restoring CHRGET to its present value, pointing to the start of the expression. Finally, the next statement is scanned for, and the bytes are all popped and loaded into the function definition in RAM.
$D2C3	$D2BB	$C50A	Check some of DEF FN's syntax. This (i) Checks for a FN token (#A5), (ii) Sets the function flag, ORing the initial of the function's name with #80, (iii) Searches for this function, setting it up if it doesn't yet exist, (iv) Checking that the type is numeric. (String functions aren't allowed).
$D2D6	$D2CE	$C51D	Evaluate FN. This routine (i) Checks FN with the last subroutine, (ii) Evaluates the expression in parentheses, and checks that it's numeric, leaving the answer in FPAcc#1, *without* changing the value of the dependent variable, (iii) Recovers the five values stored by DEF FN; (iv) Stores the current variables on the stack, (v) Puts the five floating-point bytes directly into FN's area in memory, (vi) performs the evaluation, leaving the result in FPAcc. #1, and (vii) Pops and replaces the FN DEF data in RAM.
$D349	$D33F	$C58E	Perform STR$. This apparently short routine in fact calls a rather longer routine, which is an important one in string handling. STR$ first checks that the argument evaluates to a number; it converts the contents of FPAcc. #1 into a string starting at $0100 ($0200 in BASIC 1), in the usual Microsoft form, e.g. with numbers smaller than .01 expressed in scientific notation, like 5E-03. It throws away a return address (popping 2 bytes from the stack) and sets pointers to the buffer holding the string; now, it's ready to convert the pointers into a standard zero-page pointer (not A and Y any more) and to measure the length parameter, which is determined by the first null byte encountered.

BASIC1 BASIC2 BASIC4

$D359	$D34F	$C59E	Allocate pointers and length to new string. Because *lengths* of strings are not dimensioned, each new string has to have its pointers and length recalculated. Thus, X$="ONE": X$="TWO" requires two calculations solely for this purpose. This routine requires that A on entry holds the length of the string; on exit, $5E holds the length and ($5F) points to the RAM area allocated for the string. The routine also transfers a temporary address. CHR$, LEFT$, STR$ and so on all use part of this routine. BASIC 1 uses $B0 and ($B1).
$D36B	$D361	$C5B0	Set up string in memory. This routine is used by INPUT, READ, STR$, and other functions to generate space for a string in the high end of RAM, put the string there, and set the pointers for (say) X$ to point to it. Two flags, $03 and $04 (or $5A and $5B in BASIC 1) are used for test locations here; they contain either quotes or, with a later entry point, : and , respectively. The quotes of course are redundant, except for the first, but they make the same routine usable for different purposes. On entry to this routine, A holds the low byte, Y the high byte, of the pointer to the start of string-1. The string may end with a zero terminator, or with " , or : depending on the type of string being processed. $5E holds the length, and ($5F) the pointer, on exit; many other temporary pointers are used. (BASIC 1: $B0 and ($B1)).
$D3AA	$D3A4	$C5F3	Sets string pointers – entry point from CHR$, '+', etc.
$D3B0	$D3AA	$C5F9	?FORMULA TOO COMPLEX ERROR then READY.
$D3D2	$D3CE	$C61D	Allocate space for string. On entry, A holds the length of a string; this is the amount by which the current string pointer is decremented (using a 2's complement method). This is ($30) in BASIC>1 and ($82) in BASIC 1. The same result is put into the adjacent locations which hold a 'utility string pointer'. If the end-of-arrays pointer overlaps the lowered string pointer the next subroutine is called:
$D3F4	$D3F0	$C65B	Garbage collects or prints ?OUT OF MEMORY and exits. After a garbage collection, the previous routine is re-entered. A flag in $09 ($60 in BASIC 1) ensures this process isn't endless by being set on exit form this subroutine with bit 7 high, then tested on re-entry. So garbage collection is done once only. *
$D404	$D400	$C66A	Garbage collection. This is a long routine which tidies the strings in the high end of RAM, and their pointers. To watch this in action, see Chapter 2 for programs. BASICs prior to 4 are notorious for the slowness of their garbage collection, *if* a large number of strings have been defined in the high area of RAM (i.e. not null strings or strings whose pointers point back within a program). In practice, this means string arrays. (Numeric arrays don't need garbage collection). This formula: Time in seconds=.00008*(n+11)^2 gives an accurate approximation for the time taken by n strings to free memory. BASIC 4 has a shorter and far faster routine. This operates on the pointers ($4B) and ($5C). Several subroutines subtract A from these in the course of memory freeing. In earlier BASICs the following routines have been identified :
$D497	$D497		Check for most eligible string collection.
$D4A1	$D4A1		Collect a string.

*A bug has been reported in which BASIC 4 prints ?OUT OF MEMORY instead of garbage collecting when three strings are concatenated.

BASIC1	BASIC2	BASIC4	
$D515	$D517	$C74F	**Perform string concatenation.** This routine apparently works by adding the two strings' lengths,allocating that amount of space in memory, then putting each string into the space side by side and finally setting up the name and pointers for the entire new string. ($61) is assumed to point to the first string; the second is input and evaluated by this routine.
$D530	$D532	$C76A	?STRING TOO LONG ERROR then READY.
$D552	$D554	$C78C	**Store string in high end of RAM.** Comparatively simple routine which uses A to hold the string length, and ($1F) as a temporary ponter to the start of the string. It also uses the utility pointer ($32) to transfer the data; on exit this points to the end of the string. BASIC 1 equivalents are ($71) and ($84). Several entry points are used:-
$D552	$D554	$C78C	($6C) points to the byte just after the variable's name in RAM. (In BASIC 1, ($BE)).
$D560	$D562	$C79A	A holds length, X and Y point to the byte immediately following the variable's name.
$D564	$D566	$C79E	A holds length and ($1F) the temporary pointer.
$D57B	$D57D	$C7B5	**Discard temporary string.** This routine begins by loading ($1F) with a pointer to the strings parameters; on exit, the same pointer ($1F) points to the actual string, and the bottom-of-string pointers are moved up by the length of the string, so that it will be overwritten by the next string to be defined. (This is only done if the string was the very last to be defined). BASIC 1 uses ($71).
$D5B3	$D5B5	$C811	**Clean the descriptor stack.** A holds the low byte, Y the high byte of a string vector; if these match the temporary store in ($14), Y is loaded with #0 (and the Z flag set) and $13 and $14 are loaded with A and A-3. The purpose of this is mysterious to me. This is used by the previous routine.
$D5C4	$D5C6	$C822	**Perform CHR$.** All strings of the type CHR$(n) have length 1; this routine simply inputs the parameter, ensures that a 1-byte space is available, puts the character in it and sets up the string details.
$D5D8	$D5DA	$C836	**Perform LEFT$.**
$D604	$D606	$C862	**Perform RIGHT$.**
$D60F	$D611	$C86D	**Perform MID$.** Each of these routines uses the second part of LEFT$ (from D5E6/D5E6/C843) to allocate space for the string and set it up in memory with its pointers. The length of the new substring and its starting point within the string, which were earlier pushed on the stack, are later popped and used to construct the substring. The rest of the routines, notably MID$ which has two valid syntaxes, deal mostly with syntax checking and with processing parameters. For example, RIGHT$ pulls the parameters (see next routine) including n in RIGHT$(X$,n) which is held in A and X. The length of X$ is reduced by n before entering LEFT$'s routine; so RIGHT$ picks a substring starting within X$. Note that some validation takes place to guarantee that the string doesn't reach beyond the end of its parent string. This is done by comparing the string's length with with the parameters and selecting whichever is less with the help of the carry flag.

BASIC1 BASIC2 BASIC4

$D637	$D63B	$C897	Pull String function parameters from stack. The diagram illustrates the working of this subroutine, which is called by each of the string functions. It first checks for the existence of a right parenthesis, ')', then pops ($4E), the pointer to the string, and its length, from the stack. On exit, both A and X hold the length of the string; Y is set to zero. BASIC 1 has a slightly different routine; and its pointer address is different. It uses ($9F).

The stack before:

$4E
$4D
X & A
$52
Y

SP:→

and after:

$52
Y

SP:→
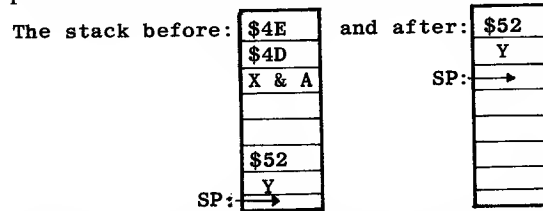
$D654	$D656	$C8B2	Perform LEN. Calls the next subroutine, from which it returns will the string's length in Y. Then it jumps into POS, where Y only is placed in FPAcc. #1 in floating-point.
$D65A	$D65C	$C8B8	Load length/ move into numeric mode from a string. VAL, ASC, and LEN each call this . It checks for a string and points to it (with 'discard a string'), loading the accumulator in the process with its length. It sets the mode flag to numeric ($07, or $5E in BASIC 1, is #0). Finally, TAY puts the length into the Y register in addition to A.
$D663	$D665	$C8C1	Perform ASC. This calls the last routine – and rejects a string of length zero – then just loads A from the temporary pointer which it set up, i.e. ($1F). This is put into floating point form by entry into POS. Note that only the initial of the string is dealt with.
$D130	$D672	$C8CE	Jump to print error message ('illegal quantity').
$D673	$D675	$C8D1	Evaluate and input a 1-byte parameter (0-255). GETCHR must point to the expression, which is evaluated and checked for range and type and also rounded down. The result is left in $62 and X; and A holds the character at the end of the numeric expression. With BASIC 1, the parameter is returned in $B4. Functions which interact with machine code need some such routine as this one; for example, POKE.
$D685	$D687	$C8E3	Perform VAL. VAL operates by treating its string as a buffer, and scanning it with the routine which converts a string into floating-point in FPAcc. #1. The exception is a zero-length string, which returns VAL =0. Several parameters are stored and later retrieved, when the conversion process is over.
$D6C4	$D6C6	$C921	Evaluate and input parameters for POKE and WAIT. Typically POKE 12345,6 and WAIT 23456,7 are the statements which this routine inputs and checks; firstly, a numeric expression is evaluated, then converted (see next routine) into a 2-byte integer. The comma is then checked for and the next parameter calculated and put into X (see last-but-one routine).
$D6CA	$D6CC	$C927	WAIT 123,45,6 is checked by re-entering the routine. This has to be left until the first 1-byte parameter has been dealt with, of course. The larger parameter is deposited in ($61) and in ($11), where it is less transient. The other location, in FPAcc. #1, is liable to be overwritten. BASIC 1 uses ($09).

BASIC1	BASIC2	BASIC4	
$D6DA	$D6D2	$C92D	Convert Floating-point accumulator #1 into 2-byte integer. This checks that the number is positive and within the range 0-65535; then calls the conversion routine in DB6D/DBA7/ CDD1; and finally stores the result in ($11), or, with BASIC 1, ($08). This is used by AND, OR, WAIT and some other BASIC commands where a 16 bit number is wanted.
$D6E6	$D6E8	$C943	Perform PEEK. On entry, floating-point accumulator #1 (i.e. $5E-$63) holds the address to be peeked in floating-point form. On exit, Y holds the peeked value, and it's reconverted to floating-point format. This is done partly by the last routine, which puts the address into a convenient form to access memory. Note that BASIC 1 tests the PEEK address to reject some values and return zero. And its floating-point accumulator occupies different memory slots, from $B0-$B5.
$D6F9	$D707	$C95A	Perform POKE. Gets two parameters, puts the second into A, and stores A into RAM where the 2-byte parameter points.
$D702	$D710	$C963	Perform WAIT. Gets two parameters, and an optional third, which otherwise is made zero. The address parameter is put into ($11), the first byte parameter into $46, and the other, optional, parameter into $47. Now WAIT is performed, a
$D713	$D721	$C974	loop which continues until the address, exclusive-ORed with the third and ANDed with the second bytes, is not zero. BASIC 2 (not 1 or 4) has Microsoft's joke here. See WAIT in Chapter 5. Or try, say, POKE 70,n:SYS 55121.
$D71e	$D72C	$C97F	Add .5 to contents of FPAcc. #1. A (low byte) and Y (high byte) point to .5 in floating-point form in ROM, then the addition routine is entered. Used when rounding.
$D275	$D733	$C986	Perform subtraction. Replaces FPAcc. #1 by FPAcc. #2 minus FPAcc. #1. On entry at this entry point, A must hold the low byte and Y the highbyte of a pointer to a 5-byte floating-point constant, which will be loaded into FPAcc. #2.
$D728	$D736	$C989	On entry here, however, both floating-point accumulators are assumed to be loaded, and their contents will be subtracted as I 've indicated.
$D73C	$D773	$C99D	Perform addition. Replaces FPAcc. #1 by FPAcc. #1 plus FPAcc. #2. On entry here, A must hold the low byte and Y the high byte of a pointer to a 5-byte floating-point value in ROM or RAM. This will be loaded into FPAcc. #2, then added to FPAcc. #1. The result is in floating-point form. If the value to be added is zero, the routine jumps to simply copy FPAcc. #2 into FPAcc. #1 without any further
$D73F	$D776	$C9A0	calculations. This entry point makes this test-it assumes that A holds the exponent of floating-point accumulator #2's contents, i.e. the contents of location $66 or $B8 in BASIC 1. If this is so, the test will speed up additions of zero.
$D744	$D77B	$C9A5	Finally, this entry point adds the two numbers without any special test.
$D778	$D7AF	$C9D9	Add two numbers which have equal exponents. (In other cases one of the numbers is modified until both have equal exponents
$D81C	$D853	$CA7D	Replace FPAcc. #1 by its 2's complement. (I.e. all the bits of the accumulator are flipped; then 1 is added).
$D853	$D88A	$CAB4	?OVERFLOW ERROR and READY.
$D858	$D88F	$CAB9	Multiply a byte subroutine.

BASIC1 BASIC2 BASIC4

$D891	$D8C8	$CAF2	Table of constants: 1, then constants for LOG (byte of 3 which is a counter for the series calculation, then .4342559, .5765845, .9618007, 2.88539), and $\frac{1}{2}$ SQR(2), SQR (2), -.5, and $\log_e 2$.
$D8BF	$D8F6	$CB20	Perform LOG to base e. See Chapter 5 on LOG for an explanation of this function and its operation.
$D8FD	$D934	$CB5E	Perform multiplication. Multiplies the contents of FPAcc. #1 by the contents of FPAcc. #2, leaving the result in FPAcc.#1. This first entry point assumes that pointers are set to the second value in memory, held in 5-byte floating-point format. These pointers are: A to the low byte, and Y to the high byte, of the start of the value. As an illustration, note that D8F9/D930/CB5A point to $\log_e 2$ and then drop through into this routine, thus multiplying floating-point accumulator #1 by $\log_e 2$. (This of course is part of the previous function). Note that the routine which loads the second floating-point accumulator also loads A with the exponent of the first floating-point number; in this way, if the first number is zero, nothing more need be done.
$D902	$D93C	$CB66	Multiplies the two floating-point accumulators without loading either of them afresh. The result is left in FPAcc.#1.
$D92B	$D965	$CB8F	Multiply a byte and store the result in the product area. (This is a temporary accumulator in locations $23-$27. In BASIC 1, $75-$79).
$D95E	$D998	$CBC2	Load Floating-point accumulator #2 from memory. This routine takes the value held as a 5-byte floating-point number and puts it into floating-point accumulator #2. In the process it unpacks the sign byte and stores this separately. These locations are used: $66 (exponent), $67-6A (mantissa), and $6B (sign). On exit from this routine, A holds the sign of the number in floating-point accumulator #1 (not #2). Pointers: A low, Y high.
$D989	$D9C3	$CBED	Multiplication subroutine to check both accumulators. This checks various conditions; if FPAcc.#2 is zero, then FPAcc.#1 is made zero; if the exponents together are too large or too small, ?OVERFLOW ERROR or zeroisation of the result respectively take place.
$D9B4	$D9EE	$CC18	Multiply Floating-point accumulator #1 by 10. This short routine doesn't use a value of 10 in ROM; instead it multiplies accumulator #1 by 4, adds this result to itself, and doubles the result. At each stage it tests for overflow.
$D9CB	$DA05	$CC2F	Constant: 10 in 5 bytes of floating-point. (84,20,0,0,0).
$D9D0	$DA0A	$CC34	Divide contents of floating-point accumulator #1 by 10. This moves accumulator #1 into accumulator #2, then sets pointers to 10 and performs division.
$D9D9	$DA13	$CC3D	Perform division into floating-point accumulator #2. On entry, A, Y, and X hold the low and high pointers to a 5-byte value and the sign comparison byte, in that order. Then FPAcc.#1 is loaded- leaving FPAcc.#2 unchanged- and the result of FPAcc.#2 / FPAcc.#1 calculated and left in FPAcc.#1.
$D9E1	$DA1B	$CC45	Perform division: FPAcc.#2 / FPAcc.#1 into FPAcc.#1. This entry point loads accumulator #2 before the division, using the routine at D95E/D998/CBC2, so the pointers A and Y must be arranged beforehand. The following entry points:
$D9E6	$DA20	$CC4A	Divide the present accumulators without changing either of them.
$DA5C	$DA96	$CCC0	?DIVISION BY ZERO ERROR then READY.

BASIC1   BASIC2   BASIC4

$DA74	$DAAE	$CCD8	**Load Floating-point accumulator from** memory. This routine takes a value held as a 5-byte floating-point number and puts it into floating-point accumulator #1. In the process it unpacks the sign byte and stores this separately. The bytes are taken from the memory locations pointed at by A (low) and Y (high). See, for an example, the routine starting at D9D0/DA0A/CC34 which loads 10 into floating-point accumulator #1 then divides this into FPAcc.#2. The locations used are $5E (exponent), $5F-$62 (mantissa), and $63 (sign). Note that zero sign bit means plus, #FF means minus.
$DA99	$DAD3	$CCFD	**Store Floating-point accumulator #1 into** memory. This packs the sign byte and rounds the accumulator, so that it fills the standard 5 bytes of a numeric variable. It is stored into 5 bytes starting with the address pointed to by X (low byte) and Y (high byte). There are four entry points: two of these point to special zero-page locations in which TAN and series expansions are worked out. (I.e. $59-$5D and $54-$58).
$DAA2	$DADC	$CD06	The third entry point stores the value in the location which ($46) points to (or ($98) in BASIC 1). This is used by LET and by the FOR-NEXT loop to store an evaluated quantity in a variable's storage after a BASIC program.
$DAA6	$DAE0	$CD0A	Finally, this entry point is the one to select when the X and Y values have to be set explicitly and don't correspond to those cast in the silicon of ROM.
$DACE	$DB08	$CD32	**Copy accumulator #2 into accumulator #1.** This moves the sign and five data bytes from one accumulator to the other; both now hold the same value. The rounding byte is made zero.
$DADE	$DB18	$CD42	**Round and copy accumulator #1 into accumulator #2.** Calls the following routine, moves 6 bytes of the accumulator (more elegantly than the last routine!), and zeroises the rounding byte. Each of these short pieces of code therefore loses a little information.
$DAED	$DB27	$CD51	**Round accumulator #1.** The rounding routine doubles the rounding byte and exits without action if the result has the carry bit clear, showing that it was less than 128. It also exits with zero (this is always signalled by the exponent's value being zero). However, if the carry bit is set, a single bit is added to the floating-point value; this process can be traced in ROM. Each byte is incremented until the result of the increment is not zero (which of course is usual). If the addition propagates through the accumulator, a routine is called which adds one to the exponent and also rotates all the bytes right. In this case, the rounding bit is lost .
$DAFD	$DB37	$CD61	**Find sign of accumulator #1.** On exit, these values apply: A=0 means value is 0. A=1 means value is positive. A=#FF means value is negative.
$DB0B	$DB45	$CD6F	**Perform SGN.** Because BASIC function arguments are put in floating-point accumulator #1 after evaluation, SGN calls the previous subroutine to compute its sign. This is placed in floating-point accumulator #1 as shown here:

$5E	$5F	$60	$61	$62	$63		$6D
#$88	sign	0	0	0	0		0

and a subroutine in the 'addition' routines is called to convert 0,1, or #FF into their floating-point form, normalised and with the sign byte set. See the next subroutines.

BASIC1	BASIC2	BASIC4	
$DB0E	$DB48	$CD72	Store contents of accumulator only in accumulator #1. Example: LDA #A0 / JSR CD72 puts the value 160 (decimal) in floating-point form in FPAcc.#1.
$DB16	$DB50	$CD7A	Evaluates a double-byte integer and converts the result into floating-point form (0-65535). On entry here, X must hold #$90, $5F the high byte, and $60 the low byte, like this:

$5E	$5F	$60	$61	$62	$63		$6D
#$90	sign	0	0	0	0		0

Note that the carry bit indicates the sign in all these routines. If it is set, the number is treated as positive and vice versa.

NOTE: The values #88, #90 (136, 144 decimal) are exponents indicating the size to which the number is to be normalized. Numerals of three or four bytes can be evaluated by an extension of this calculation routine. See INT.

BASIC1	BASIC2	BASIC4	
$DB2A	$DB64	$CD8E	**Perform ABS.** See Chapter 5.
$DB2D	$DB67	$CD91	Compare Floating-point accumulator #1 with 5-byte floating-point number. A (low byte) and Y (high byte) point to the 5-byte value in memory. On exit, the accumulator (A, not one of the floating-point variety) indicates the relative sizes: A=0 means the values are equal. A=1 means that accumulator #1 > memory. A=#FF means that accumulator #1 < memory.
$DB6D	$DBA7	$CDD1	**Convert Floating-point #1 into integer, within FPAcc.#1.** This routine is called by D6DA/D6D2/C92D which, however, also treats the fixed-point number as an address, which it stores in ($11), or, with BASIC 1, in ($08).

$5E	$5F	$60	$61	$62	$63
	HI++	HI+	HI	LO	

BASIC1	BASIC2	BASIC4	
$DB9E	$DBD8	$CE02	**Perform INT.** Acts on floating-point accumulator #1, rounding it down to the nearest integer, but leaving the result in floating-point form.
$DBBB	$DBF5	$CE1F	Used when zeroising all of accumulator #1 when the exponent has been found to be zero.
$DBC5	$DBFF	$CE29	**Convert an ASCII string into a numeral in FPAcc.#1.** VAL and other routines use this to evaluate a numeral which is in string form. GETCHR should point to this string before entering this routine; then JSR 0070/ JSR CE29 (or whatever other values apply for BASICs 1 and 2) scans the string and puts the result in floating-point accumulator #1. E . + - and leading and other spaces are specially checked; the routine to multiply by 10 adds together consecutive digits as they are encountered.
$DC3C	$DC76	$CEA0	Add new ASCII numeral to the mantissa.
$DC50	$DC8A	$CEB4	Add contents of A only to floating-point accumulator #1. Example: LDA #$0F/JSR CEB4 adds 15 to the accumulator.
$DC85	$DCBF	$CEE9	String conversion constants. There are three of these: 99 999 999.9, 999 999 999.75 and 1 000 000 000.
$DC94	$DCCE	$CF78	**Print IN followed by linenumber.** IN is a message from the standard table. The linenumber is printed by loading A and X with the high and low bytes respectively which are stored in ($36), or ($B1) in BASIC 1. This is the current linenumber, which is stored by RUN as BASIC is executed.
$DC9F	$DCD9	$CF83	This prints 256*A + X on the following line.

BASIC1 BASIC2 BASIC4

$DCAF	$DCE9	$CF93

Convert contents of **Floating-point accumulator #1** into **ASCII string** starting at $0100. On exit, A and Y hold #0 and #1, pointing to $0100, so that the print routine CA27/CA1C/BB1D can print the result as a string. Note that the buffer is at the lowest end of the stack, inaccessible to BASIC. Chapter 2 has a table showing how this formatting process works in practice. Note the zero terminating byte, and the special-case processing for zero. PRINT USING, in Chapter 5, demonstrates how this routine may be adapted to get other output formats.
Note that FPAcc.#1 is changed when this routine has been run. The tables of constants following this routine are used in the comparison/ conversion process. The later ones deal with TI\$. And the three values tabled before this routine are used to decide when scientific format should be used.

$DD3A	$DD74	$D01E

Convert TI\$ from three bytes into the corresponding string.

$DDE3	$DE1D	$D0C7

String conversion and TI\$ constants: .5 for SQR and rounding, then 15 4-byte constants, -100 000 000, 10 000 000, -1 000 000, 100 000, -10 000, 1 000, -100, 10, -1 and -2 160 001, 216 000 (=1 hour), -36 000, 3600 (=1 minute), -600, 60 (=1 second).

$DE24	$DE5E	$D108

Perform **SQR**. This puts FPAcc.#1 into FPAcc.#2, loads FPAcc.#1 with .5 and performs the next routine:

$DE2E	$DE68	$D112

Perform **power calculation (↑)**. Calculates FPAcc.#2 to the power FPAcc.#1. Note that FPAcc.#1 may be loaded from memory by setting A and Y pointers and entering one instruction earlier. FPAcc.#2 must be loaded before running this routine. Both numbers are tested for equality with zero and if zero is found, set the result in FPAcc.#1 to 0 or 1 according as FPAcc.#1 or FPAcc.#2 is zero. The function is evaluated by saving FPAcc.#1 in the zero page, then multiplying the logarithm of FPAcc.#2 by FPAcc.#1, and finding the exponent of the result.

$DE67	$DEA1	$D14B

Negate contents of floating-point accumulator #1. Changes the sign byte with EOR #FF, so 0 becomes #FF and vice versa. FPAcc.#1 is unchanged if it equals zero.

$DE72	$DEAC	$D156

Table of constants: $1/\log_2 e$ and 8 constants for EXP's series evaluation: byte of 7 then 2.149876 E-5, 1.435231 E-4, 1.342263 E-3, 9.614017 E-3, 5.550513 E-2, 2.402263 E-1, 6.931471 E-1, 1. The series in fact calculates 2^n.

$DEA0	$DEDA	$D184

Perform **EXP**. The value e↑FPAcc.#1 is computed and left in FPAcc.#1. For notes on the method and on the series used, see Chapter 16.

$DEF3	$DF2D	$D1D7

Function evaluation routine: this calls the next routine. It evaluates more complex expressions of the type q*fn(q*x), where fn(x) is evaluated by the series expansion formula embodied in the next piece of code:-

$DF09	$DF43	$D1ED

Main series evaluation routine. All the mathematical functions (LOG,SIN,COS, etc.) are evaluated by transforming the argument into a suitable range (e.g. 0-1), calculating the result and finally, where necessary, modifying the result- perhaps by chenging the sign or altering the exponent.
This subroutine must be entered with the pointers ($6E), or ($C0) in BASIC 1, looking at a single byte, which will be read as the number of values in the table. Then by a repetitive process the tabled values are added and multiplied to FPAcc#1 so the table byte=3/5/3/1/10 (for example) finds the value of $10 + 3x + 5x^2$. See Chapter 16 for more on this subject.

BASIC1 BASIC2 BASIC4

$DF3D	$DF77	$D221	RND - multiplicative constant. (=11 879 546.4)
$DF41	$DF7B	$D225	RND - additive constant. (=3.927 677 78 E-8)

$DF45	$DF7F	$D229	**Perform RND.** The first three instructions of this function compute the sign of accumulator #1 (and hence of the argument of RND) and branch to three sections of the routine according to this sign - counting negative, zero and positive as different 'signs'. All three branches meet and exit from the routine together. Briefly, what happens is this:
$DF4C	$DF86	$D230	Zero argument. The four bytes $5F-$62 in FPAcc.#1 are each loaded from the VIA timers; 2 of these change with every clock cycle, so there's some justification for calling this 'random'. (BASIC 1 uses the wrong ROM addresses here, probably because the final positioning of the chips wasn't settled when RND was written). Then jumps to common exit.

$5E	$5F	$60	$61	$62	$63		$6D
Exp.	M	M ←→ M		M	Sign		Round

$DF63	$DF9D	$D247	Positive argument. Multiplies the stored random number by the first constant at the top of the page, then adds the second. Then continues with :
$DF78	$DFB2	$D25C	Negative argument. Interchanges bytes as marked.
$DF88	$DFC2	$D26C	Common exit routine. This puts: #0 into $63 (i.e. positive), Exponent into rounding byte, and #$80 into Exponent. The latter forces the result into the range 0-1, the former perhaps is intended to ensure that the exact value 0 does not occur. Finally, FPAcc.#1 is stored into the random number work area, ready for the next positive argument in RND.

$DF9E	$DFD8	$D282	**Perform COS.** Puts pi/2 into FPAcc.#2 and adds; then:
$DFA5	$DFDF	$D289	**Perform SIN.** Evaluates SIN of FPAcc.#1 and leaves the result in FPAcc.#1. The argument is in radians. See Chapters 5 and 16 for more information.
$DFEE	$E028	$D2D2	**Perform TAN.** Evaluates TAN of FPAcc.#1, by dividing the sine of that value by its cosine. As the argument approaches 90° and other values (pi/2) the calculation will inevitably lose precision.
$E01A	$E054	$D2FE	Table of constants: pi/2, 2*pi and .25. Then there's a byte which acts as a counter; it is 5, and the constants following (6 of them!) are -14.38139, 42.007797, -76.70417, 81.605223, -41.3417021, and 2*pi again. These are used by SIN. BASIC 2 has also !TFOSORCIM in encoded form.
$E048	$E08C	$D32C	**Perform ATN.** The arctangent is left in FPAcc.#1 after evaluation; it's in radians. It is calculated with the aid of a series with 12 terms; this is the longest series used, but it also happens to be based on the simplest, and is optimised for the range 0-1. (The basic series is $x - x^3/3 + x^5/5 - \ldots$)
$E078	$E0BC	$D35C	Counter and table of 12 constants for ATN evaluation. These are: -6.84793912 E-4, 4.85094216 E-3, -.0161117018, .034209638, -.0542791328, .0724571965, -.0898023954, .110932413, -.142839808, .19999912, .333333316, and 1.
$E0B5	$E0F9	$D399	**CHRGET routine and RND seed for relocation into RAM.** BASIC on reset moves both these tables into RAM, starting at $70 (or $C2 in BASIC 1) where they are positioned consecutively. In fact only 4 bytes of RND are transferred, so its value, theoretically .811635157, could presumably vary within the range .811635137 - .81165196.
$E0C6	$E10A	$D3AA	Entry to ROM CHRGET where the fixed address is unimportant, at SEC/SBC #30 etc. Can be used to save zero page bytes. If the program has no spaces an earlier entry point may be used.

BASIC1 BASIC2 BASIC4

$E0D2	$E116	$D3B6	**Test RAM and initialise BASIC.** This routine is the final call made by the system on reset, including switch on, when the reset vector at (FFFC) is called. This first sets input/output values before jumping to this routine. Note that there is an alternative path on reset, used by Jim Butterield's technique for resetting BASIC 2 and 4 machines, which enters the monitor without disturbing BASIC. In BASIC 1 this leads to a diagnostic program, not a monitor.

A SYS call to the appropriate address above, or an indirect jump to (FFFC) with any ROM, erases RAM memory above $0400 and resets all the BASIC pointers, and so is a reliable way to return memory to the cold-start situation which obtains when the machine is turned on. (Note that an indirect jump is represented by opcode $6C=108 decimal. So POKE 108 and 252 and 255 into consecutive locations and SYS the first of these to reset any ROM). See Chapter 13 for an account of the events of Reset. Some major locations are:

$E0E5	$E12F	$D3C7	**Move CHRGET subroutine and RND seed to zero page.**
$E10C	$E152	$D3EA	**Start of BASIC becomes $400; RAM test and exit.** Note that A holds #0 on entry. From $0400, #$55 (i.e. %01010101) is written to RAM and read back; then #$AA (%10101010) is written and read.*This is a standard type of chip test. When the read-back is not equal, or $8000 has been reached, the next routine is dropped into:
$E131	$E174	$D417	**Set BASIC string and variable pointers to their start values.**
$E167	$E196	$D42A	**Print *** COMMODORE BASIC *** (BASIC1),**
			### COMMODORE BASIC ### (BASIC2), or
			*** COMMODORE BASIC 4.0 *** (BASIC 4).
$E15A	$E19D	$D431	**Calculate and print bytes free.**
$E174	$E1B7	$D44B	Tables holding the messages for bytes free and for Commodore BASIC. The second table for BASIC 4 seems to be an
		and $DEA4	afterthought. It includes '4.0'.
		$D3F9	Prints 44030 bytes free. (Joke?).

MACHINE-LANGUAGE MONITOR (MLM).

$FD11	$D472	**'Call' entry to monitor.** This prints C* followed by details like this:

```
     PC   IRQ  SR AC XR YR SP
  .;  B780 E455 32 38 2C 34 FA
```

PC points to BASIC; IRQ and SP are taken from the stack and are reliable. The 'registers' are garbage from the input buffer.

$FD17	$D478	**'Break' entry to monitor.** This prints B* and pulls the stack to determine the contents of the program counter and registers: it assumes an entry by SYS 1024 or SYS 4 from BASIC (or any location hoding a zero byte), or by a machine-code routine entering a BRK instruction. SYS to this address will remove data from the stack. Note that BASIC 4 differs from BASIC 2 in restoring normal devices on BRK, so the monitor always prints to the screen. In order to dump monitor information, use the call entry: OPEN 128,4: CMD 128,"MONITOR": SYS 54386 which directs output to the printer. PRINT#128: CLOSE 128 unlistens the printer. (These figures are for a printer with automatic line feed on).
$FD56	$D4BA	**Start point: waits for command after . and executes it.** For example, . M 1000 1010 is processed from here: M is searched in a table, analogous to the BASIC keyword table, and its address-1 pushed on the stack. RTS then jumps.

---

*BASIC 1 has a less thorough test, using #$92 and #$24 (%10010010 and %00100100). The intention, to test all the bits, is the same.

BASIC2 BASIC4

$FD70 $D4D4   **Search table of commands.** Compares 8 single byte commands with that input. (: ; R M G X L S). If not found, jumps to the address USRCMD, ($03FA), which is set up to point at a routine to print ? crlf, then jumps to START. This RAM address may be altered to include one's own monitor commands such as those used by Extramon.

$FD93 $D4F7   **Display memory.** On entry, A holds the number of bytes to be displayed, and ($FB) holds the first address.

$FDA7 $D50B   **Read a byte and store in RAM.** Reads a byte into A and stores it in ($FB), exiting with ? if the readback doesn't equal the byte. Increments the pointer ($FB).

$FDBF $D523   Sets ($FB) ready to $0202. On exit, A holds #5.
$FDCA $D52E   Print two spaces.
$FDCD $D531   Print one space.
$FDD0 $D534   Print one carriage return + line feed.
$FDD5 $D539   Increment temporary pointer locations ($FB).

$FDE0 $D544   **Three tables for MLM.** These are:
(i) ASCII values of commands : ; R M G X L S,
(ii) Address high then address low bytes corresponding to the address of each command less 1,
(iii) Text storage of [Rtn]   PC   IRQ   SR   AC   XR   YR   SP.

$FE23 $D587   **R (Display registers).** Prints 29 characters of the text table, followed by the program counter and IRQ as 'words' and 5 other bytes, all from the input buffer. These record the situation as it was at BRK, i.e. SYS 1024 etc.

$FE58 $D5BC   **M (Display memory).** Most of this routine is validation and housekeeping. Sets of 8 bytes are displayed using 'Display memory' (above) with A=#8. ($FC) holds the upper limit beyond which memory won't be displayed, except as part of the last 8-byte block of data.

$FE97 $D5FB   **; (Modify registers).** This inputs the new program counter, storing it in 0200 & 0201; IRQ similarly is put in 0207 & 0208. Finally, 5 bytes are read and stored in $0202 ff. The layout within the input buffer is this:

$0200	$0201	$0202	$0203	$0204	$0205	$0206	$0207	$0208
PC hi	PC lo	PSR	ACC'R	XR	YR	SP	IRQ hi	IRQ lo

Note that the registers are *not* modified by R; only the input buffer stores these values, which are loaded by G, the 'go run' command.

$FEB9 $D61D   **: (Modify memory).** Reads the memory address into ($FB), then reads-and-stores 8 bytes into RAM. This routine is used by the latter routine also; in its case ($FB) points to $0202 and only 5 bytes are stored. This routine stops, printing a query, if on readback the byte doesn't have the write value. This happens on trying to write to ROM for instance.

$FECF $D633   **G (Go, Go run).** This command has two formats. G alone fetches all the registers from $0200-$0208 and loads them, so its destination is determined by the program counter stored in $0200 and $0201. G ABCD overwrites the program counter store with $ABCD, but loads the other registers just as G does. The effect is that any routine can be called, with any values of A,X,Y, processor stack, and IRQ.

$FF07 $D66B   **X (Exit to BASIC).** Sets the stack pointer to its entry value and jumps to BASIC warm start (C389/B3FF) where READY is printed and a direct commend awaited. The program and its variables are all preserved intact The input buffer reverts to its BASIC input buffer role.

BASIC1 BASIC2 BASIC4

$FF11 $D675   **L / S (Load and Save machine-code routines).** These are mixed together because of the similarity in syntax. The main locations used are these:

$B4=index of command in table (i.e. L=6, S=7).
$96=ST byte.
$9D=LOAD/ VERIFY select flag - Load=0, Verify=1.
$D1=length of string (i.e. device number + name, or, in the case of defaults to tape, name only).
$D4=device number (8=normal CBM disk, 1=cassette #1, etc.).
($FB) and ($C9) store the low address and high address for Save. (Load needs only the low address).
($DA) points the start of the string or filename.

The syntax is shown by these examples:
```
.L "M/C SORT",01         : REM LOADS M/C SORT FROM TAPE #1
.S "1:OLD.033A",08,033A,0381: REM SAVES 033A-0380 ON DISK
.L                       : REM LOADS FIRST FILE ON TAPE #1
.S  "0:RAM DUMP",0D,0000,0100:REM SAVE TO DEVICE #13
```

When the above pointers have been set, the routine at F43E/F322/F356 performs LOAD without requesting parameters, and the routine F6B1/F6A4/F6E3 performs SAVE in the same way.*These routines can be called from BASIC and represent the only feasible way of loading and saving chunks of machine-code from BASIC.

Subroutines used by MLM. The names are Commodore's.

$E76A $D717   **WROA. Ouput hex digits.** Prints contents of ($FB) as 4 hex digits, for example 4CD3.

$E775 $D722   **WROB. Output single byte.** Prints the contents of the accumulator as 2 hex digits, for example F3.

$E784 $D731   **WRTWO. Output two characters.** X contains the first, Y the second, character; in the monitor, these are set, by the next routine, to be 48-57 or 65-70, i.e. ASCII 0-9 or A-F.

$E78D $D73A   **ASC. Convert 0-15 into ASCII character.** This takes the contents of A and converts to ASCII - see previous routine.

$E797 $D744   **T2T2. Exchange contents of ($FB) with ($FD).**

$E7A7 $D754   **RDOA. Input full hex address.** This sets the flashing cursor and awaits input of a 16-bit value, e.g. ABD8. The result is placed in ($FB). Carry is cleared if there are spaces only.

$E7B6 $D763   **RDOB. Input one hex byte.** The cursor flashes and a single hex byte (e.g. AB) is input to the accumulator, which holds the same value(e.g. AB!). Carry clear means nothing was input.

$E7E0 $D78D   **HEXIT. Convert ASCII numeral to HEX.** Accumulator values of #30-#39 and #41-#46, which print as 0-9 and A-F, are converted to 0-#F in the accumulator.

$E7EB $D798   **RDOC. Input character/ await return.** Flashes cursor and inputs a single character. If this is carriage return, the subroutine return is stopped, and the routine exits to check the command letter or punctuation symbol presumably present at the start of the line.

$E7F7 $D7A4   **ERROPR. Print ?.** Then go to input the next line.

---

*BASIC 1, though without a machine-code monitor - unless TIM ('tiny monitor') or a Supermon-style monitor is loaded in - nevertheless has LOAD and SAVE as BASIC commands which are usable from BASIC or machine-code. The locations are different: There is no index; ST is $020C; $020B is LOAD/VERIFY; $EE is length; $F1 is device#; low and high addresses for SAVE are ($F7) and ($E5); and ($F9) points to the start of the filename.

DISK COMMANDS - BASIC 4 ONLY.

$D7AF   **Perform RECORD.** This routine validates RECORD#file number,record number [,optional byte number]. The byte parameter is tested to ensure it's within the range 1-254; and it defaults to 1 if not explicitly mentioned. The logical file number is tested to ensure it is not zero; and the record number can take any 2-byte value. It may be written as an expression, but if it is, it must be within brackets unless it starts with a number. Thus, these are valid: RECORD#2*2,(Q),1 and RECORD#2,145,5. Commodore has introduced some new rules for validation into its disk commands, which are not quite the same as in BASIC itself. RECORD jumps to $DA31 to send its message to disk.

$D804   **4 disk BASIC parameter checking routines.** These print ?SYNTAX ERROR if the bits set in $033E don't match a bit pattern that is looked for, and so indicate that a wrong parameter has been entered, or a correct one omitted. $D82E for example checks that A has bits 0 and 2, at least, on.

$D838   **Dummy disk control messages.** This table holds commands corresponding to 10 instructions. The tables are used to construct full messages in the disk command buffer. A simple example: BACKUP has 44 D2 3D D1. 44 is ASCll V and 3D is ASCII =. D2 and D1 are not ASCII values, but a code showing that destination and source drives are to be substituted. The resulting string has the same effect as D1=0 which duplicates disk 0 onto disk 1. The word 'BACKUP' is not used by the disk unit.

$D839   DIRECTORY or CATALOG ($ D1)
$D83B   DOPEN etc. (D1 :F1,E1,E0)
$D842   APPEND (D1 :F1,A)
$D847   HEADER (N D1: F1) or (N D1:F1,D0)
$D84D   COLLECT (V D1)
$D84F   BACKUP (D D2=D1)
$D853   COPY (C D2: F2=D1: F1)
$D85B   CONCAT (C D2: F2=D2: F2, D1: F1)
$D867   RENAME (R D1: F2=D1: F1)
$D86F   SCRATCH (S D1: F1)

    Some of these commands have alternative forms: HEADER may have length 4 or 6 in its string, COLLECT 1or 2, CONCAT 8 or 12, depending (for example) whether HEADER's ID is given or not. The following table shows how the dummy values (which are detected by bit 7 being high) are understood : D0 = DOS disk ID (2 bytes)
    D1 = source drive number
    D2 = destination drive number
    E0 = read or write
    E1 = parameter length (relative files) or S (sequential files)
    F1 = source file name
    F2 = destination file name

$D873   **Perform CATALOG or DIRECTORY.** Both of these commands jump to this address; they are identical. Ths syntax checking test that $033E has bits 1,2,3,5,6, and 7 all off; only a drive number and string are permitted. The validation is performed by the routine $DC68; to save space I shall not mention this with each instruction, although every disk command except RECORD uses it. DIRECTORY works like, and closely resembles, the DOS wedge program; it 'lists' the directory, not in RAM, but by looking for end-of-line zero bytes, throwing away the link address, printing the 'linenumber' which is the filelength, and printing each character of the name. Thus the listing takes place without disturbing RAM. Nevertheless, the directory is still stored, as in DOS 1, in program form.

$D8A3,** **$D8A5 Throw away 4 (later 2) bytes then print 'linenumber'
$D905   End-of-line and possible end-of-program subroutine.
$D911   Exit if ST <> 0
$D91A   **Output a character.** I.e. set device/ output/ set default devices. Enables the directory to be output to printers etc.

BASIC 4

$D92F　**Find next available secondary address.** Sets $D3 (=secondary address) to #62 + by searching all the open files until an unused secondary address is discovered. At each loop the trial value is incremented. This saves the user the effort of thinking up yet another meaningless secondary eddress.

$D942　**Perform　DOPEN.** Tests for DOPEN# filenumber,"name" and also the options for drive number, relative record length, unit number (i.e. ON U9 or ,U9 with unit #9), and sequential read/ write. It sets up a command string and jumps to the normal OPEN routine.

$D977　**Perform APPEND.** This command is not in some disk manuals. The command string puts ,A after the file name and this automatically performs OPEN *and* sets the pointers to write sequentially on ot the end of the file.

$D991　**Get disk status string DS$.**
$D995　**Get DS$ (jump table entry).** These routines set up a string in memory with length held in $0D and pointer to start of ($0E). The value of DS can be tested with this machine-code:

```
LDY #0
LDA (OE),Y
CMP #32
BCS ERROR   ; VALUE IS NEITHER 0 NOR 1.
```

$D9D2　**Perform　HEADER.** This has two forms, with and without a disk ID. As well as the usual validation, this command uses the ARE YOU SURE? prompt. On exit, DS id checked and ?BAD DISK ERROR appears if DS > 1.

$DA07　**Perform DCLOSE.** The syntax check permits either DCLOSE or DCLOSE# file number [ON U8], or other device number. $DA1B closes a numbered file; when no file number is given, all open files of the correct device number are sought and closed by the routine at $DA1B.

$DA31　Set up disk record pointers. This is called from RECORD. It sends a five byte string to the disk which contains:

ASCII for p	Secondary address.	Rec.no.low	Rec.no.high	Byte
$0353	$0354	$0355	$0356	$0357

The default value for byte is 1. Byte is checked to ensure that only values from 1-254 are accepted.

$DAC5　**Perform COLLECT.** COLLECT (in BASIC<4, VALIDATE corresponded to this) has two forms: one has one parameter in the command string, the other two, depending on whether a drive is specified or the default is used.

$DA7E　**Perform BACKUP.** This checks that two drives are specified and an optional device number. It sets Y=#$16 and A=#$4 and enters the next routine, which is also used by all the other disk commands except RECORD:-

$DA98　**Send DOS command string from buffer to disk.** On entry, Y holds the offset from D839, the table of dummy commands, and A holds the length of the *dummy* command: the true length of the command, after the details have been inserted by DBFA, naturally varies with (for example) the length of a program's name.

$DAA7　**Perform COPY.** COPY sends a disk command string with 8 components, irrespective of its syntax (there are several valid versions).

$DAC7　**Perform CONCAT.** Like COPY, CONCAT sends a command string with a fixed number of variables. In the case of CONCAT this means 12 variables. These are arranged (see D838 ff.) in a string like this:

```
C D2 : F2 = D2 : F2 , D1 : F1  where D and F are drive & file numbers.
```

Note that this string,, and the others like it, are sometimes called the 'DOS interface' in Commodore documentation, referring to the fact that the data which is sent to the disk has to be in one of the standard forms to be processed corretly.

BASIC 4

$DAD4 **Put source file name into DOS command string.** This routine is called within DBFA when F1 is encountered in the dummy disk command table; it places details, including the file's name, into the command buffer, which starts at $0353, and it sets pointers, i.e. ($FD), to this address.

$DAFD Store 2 parameters in adjacent addresses in the command string buffer.

$DB0D **Perform DSAVE.** This uses three parameters, sharing those of DOPEN at D83B. The filename and disk drive only are sent in the command string. Note that the filename may be preceded by '@' if save-with-replace is required; then the file is saved without the necessity to avoid ?FILE EXISTS ERROR by first scratching the file. (However, '@' is reputedly not bug-free, and is to be avoided by the cautious user).

$DB3A **Perform DLOAD.** DLOAD uses similar output parameters to DSAVE. The flag in $9D is set zero for LOAD, not VERIFY. (This suggests that a disk verify command, say DVERIFY, could be written, identical to DLOAD but storing #1 in the verify flag).

$DB55 **Perform RENAME.** DOS interface is R D1:F2=D1:F1

$DB66 **Perform SCRATCH.** The DOS interface is S drive no. : filename (the filename may include * and/or ?). This combination of parameters is checked by the parsing routine. To make erroneous deleting of files less easy, a subroutine which prints ARE YOU SURE? (at DB9E) waits for 'yes' or 'y'. On exit, DS$ is read and - if it's been set - printed to the screen if the mode is direct.

$DB99 Check command is direct mode entry. If it is, the equals flag (Z) is set.
$DB9E Print 'are you sure?' and await reply. Only unshifted y or yes set the flags: C is returned clear if y or yes in entered. Otherwise BCS may be used to exit or jump past the unwanted code.
$DBD7 Print ?BAD DISK ERROR if in direct mode. DBDC prints it in any mode.

$DBE1 **Clear DS$ and ST.** This routine leaves A,X, and Y unchanged, and sets ST and the length of the string DS$ to zero. Both are effectively zeroised. If a DS$ string existed already, its pointers in RAM are set to $FF28.

$DBFA **Expands dummy variables to fill DOS command string in buffer $0353 ff.** On entry, Y holds the offset of the start of the command string from $D839. A holds the length of the *dummy string*. Example: APPEND sets Y=#9,A=#5. This corresponds to the data indicated for $D842; q.v. Each dummy value, for instance D1 or E0, is filled from the storage details in $033A ff.

$DC4C Set file name length to value in X register; set pointer ($DA)=$0353.
$DC57 Process L,S, and W flags.

$DC68 **Parse disk BASIC command and store parameters.** It is this routine which permits disk parameters to be entered in any order. A large loop processes the string, looking for: # W L R D ON token U I " or (. Anything else gives ?SYNTAX ERROR. $033E stores, bitwise, the parameters as they are processed. So, if (say) DLOAD#3#4 is entered, which of course is wrong, the file number flag will be set on the second look at # and this will cause the ?SYNTAX ERROR message. The 7 bits of $033E have these meanings:

MEANING OF BITS SET IN $033E

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
1: NO @	WRITE	DEST.DRIVE.	SOURCE DRIVE	DEVICE #	FILE #	DEST.FILE	FILE
0: @	READ	NO DEST. DR.	NO SOURCE DR.	NO DEV.#	NO LFN	NO DEST.	NO FILE

$DE49 Get file name. On entry, ($1F) points to the start of the string. If its length exceeds 16, or begins with '@' and exceeds 17, the routine prints ?SYNTAX ERROR. On exit, A holds length, X and Y also hold ($1F) pointers.

$DE87 Get parameter in range 0-255. The value is returned in X and in $62. The parameter is taken from BASIC (by CHRGET) and evaluated; if it does not begin with 0-9, it must be within parentheses; so all these expressions are accepted if the range is right: (X+Y) and 12 and (12+VAL(J$)) and 4+X. Parameters such as the logical file number are input using this.

$DE20 ?SYNTAX ERROR; $DE27 ?ILLEGAL QUANTITY ERROR; $DE74 ?STRING TOO LONG ERROR; $DB27 ?BAD DISK ERROR.

SCREEN, KEYBOARD, AND INTERRUPT PROCESSING ($E000 ff).

BASIC1	BASIC2	BASIC4	
		$E000	JUMP TABLE FOR 80-COLUMN CBM ONLY.
		$E000	JMP E04B: Bell + home cursor + initialise input/ output.
		$E003	JMP E0A7: Input from keyboard buffer.
		$E006	JMP E116: Input from screen or keyboard via ($E9).
		$E009	JMP E202: Output a character via ($EB).
		$E00C	JMP E442: IRQ servicing: BRK to monitor, hardware interrupt
		$E00F	JMP E455: Clock, cursor, keyboard, tape servicing.
		$E012	JMP E600: Exit from interrupt.
		$E015	JMP E051: Clear screen within window.
		$E018	JMP E07A: Set CRT controller chip to lower/ upper case.
		$E01B	JMP E082: Set CRT controller chip to upper case/ graphics.
		$E01E	JMP E088: Other CRT controller chip settings.
		$E021	JMP E3C8: Screen scroll down.
		$E024	JMP E3E8: Screen scroll up.
		$E027	JMP E4BE: Finds key from keyboard decoding table.
		$E02A	JMP E6A7: Rings bell one chime.
		$E02D	JMP E036: Store accumulator A in repeat flag.
		$E030	JMP E1E1: Set top left of scrolling window.
		$E033	JMP E1DC: Set top right of scrolling window.
		$E036	Called from E02D.

BASIC1	BASIC2	BASIC4 40-col.	BASIC4 80-col.	
$E1E1	$E1DE	$E000	$E60F	Initialise input/ output locations in V1A, P1As, set clock to zero, set cursor, etc.
$E236	$E229	$E04B	$E051	Clear the screen (within window in 8032).
$E269	$E257	$E257	$E05F	Home cursor.
$E5DB	$E25D	$E07F	$E06F	Position cursor anywhere on screen − $C6 holds horizontal, $D8 vertical, positions. (In BASIC 1, $E2 and $F5).
$E27D	$E285	$E087	$E0A7	Get character from keyboard buffer. On exit, the character is in A. The number of characters in the buffer is held in $9E ($020D in BASIC 1), and is assumed to be at least 1.
$E294	$E29A	$E0BC	$E0BC	Input from keyboard. Gets character(s) from the keyboard buffer, echoing them to the screen and handling the cursor position, finishing with carriage
$E2B7	$E2B8	$E0DA	$E0DA	return. If shift-Stop = Run is pressed, the keyboard buffer is replaced by dL"*[Return]run[Return] in BASIC 4, LOAD[Return]RUN[Return] in BASIC<4.
$E2FA	$E2F4	$E116	$E116	Input from screen or keyboard. This routine is used by the INPUT routine whenever input is to be made from non-tape and non-1EEE devices, i.e. CBM's internal screen or keyboard. The X and Y registers are preserved. When input is from screen, quotes and reverse flags are tested for, and the cursor is updated.
$E349	$E33F	$E167	$E16A	Switch quote flag (0 to 1 or 1 to 0) if quote found.
$E356	$E34C	$E174	$E177	Print screen character; update cursor.
$E397	$E38D	$E1B3		Set 80-character line indicator.
$E3A4	$E396	$E1BE		Convert 40-column line to 80-character line.
$E3C4	$E3B4	$E1DE		Back to previous line (when actioning [DEL], [LEFT])
			$E1AA	Advance cursor; next line if end of window.
			$E1C1	Clear line to end of window.
			$E1D2	Set window to fullest size.

BASIC1	BASIC2	BASIC4 40-col.	BASIC4 80-col.	
$E3EA	$E3D8	$E202	$E202	Print CBM ASCII character to the screen. This routine deals with all the cursor control and screen editing characters. It takes care of cursor processing and automatic screen scrolling. On entry, A holds the character to be printed; note that both X and Y registers' contents are preserved. BASIC 4, but *only the 80-column version,* has an indirect jump enabling users' routines to intercept the output; it is via ($EB).
				There is different processing for direct mode and program mode.
$E3FF	$E3EC	$E216	$E224	**UNSHIFTED CHARACTERS:**
$E3FF	$E3EC	$E216	$E224	Carriage return, CHR$(13)
$E40A	$E3F7	$E221	$E22F	Ordinary ASCII character, CHR$(32) – CHR$(127)
$E41D	$E40A	$E234	$E242	Delete, CHR$(20)
$E444	$E431	$E25B	$E26D	Reverse, CHR$(18)
$E44B	$E437	$E261	$E273	Home, CHR$(19)
$E452	$E43E	$E268	$E287	Cursor right, CHR$(29)
$E468	$E454	$E27E	$E296	Cursor down, CHR$(17)
			$E2A0	Tab, CHR$(9)
			$E2D4	Erase beginning of line, CHR$(22)
			$E2E7	Delete line, CHR$(21)
			$E595	Scroll down, CHR$(25)
			$E59F	Set top of window, CHR$(15)
			$E5AE	Text mode, CHR$(14)
			$E5B7	Bell, CHR$(7)
$E48F	$E47A	$E2A4	$E2F4	**SHIFTED CHARACTERS:**
$E482	$E48D	$E2B7	$E307	Shift-return, CHR$(141)
$E49B	$E34C	$E174	$E177	Shifted ordinary ASCII chr., CHR$(160)-CHR$(255)
$E4AD	$E498	$E2C2	$E312	Insert, CHR$(148)
$E512	$E4FC	$E326	$E35C	Reverse off, CHR$(146)
$E52A	$E513	$E33D	$E377	Clear, CHR$(147)
$E51B	$E504	$E32E	$E364	Cursor left, CHR$(157)
$E4E2	$E4CD	$E2F7	$E34B	Cursor up, CHR$(145)
			$E380	Tab set, CHR$(137)
			$E393	Erase to end of line, CHR$(150)
			$E5C0	Insert line, CHR$(149)
			$E5D6	Scroll up, CHR$(153)
			$E5E2	Set bottom of window, CHR$(143)
			$E5F1	Graphics mode, CHR$(142)
			$E5B7	Shift-bell (=bell), CHR$(135)
			$E3BD	Escape and shift-escape, CHR$(27) and CHR$(155)
$E530	$E519	$E343	$E3A3	Cursor down.
$E548	$E52F	$E359	$E3B6	Process Return.
$E559	$E53F	$E369	$E3C8	Scroll screen up. BASIC 1 and 2 are identical; and BASIC 4 (40 column) almost identical to these. But BASIC 4 (80 column) is rewritten (i) To allow the screen to scroll up; (ii) To include a pause feature which stops screen scroll; (iii) To prevent the IEEE 'EOI' character being sent, which all other BASICs do when the screen scrolls (because one of the VIA timers, E811/ E812, is used to time the delay loop when RVS is pressed!) Note that BASIC 4 with 40 columns still has this bug in the IEEE. (The remedy, of course, is simply to avoid scrolling the screen).

BASIC1	BASIC2	BASIC4 40-col.	BASIC4 80-col.	
			$E3E8	Scroll screen down.
$E5C1	$E5A1	$E3C9	$E40B	Check for [Reverse]; if pressed, .5 sec. delay.
$E5C8	$E5A8	$E3D0	$E412	Half-second delay.
$E5DB				Start new screen line (called by E269).
$E605				Action 'Insert'.
$E617	$E5CC	$E3E2		Open a space in a line with 'Insert'
$E66B	$E61B	$E442	$E442	**MAIN INTERRUPT ENTRY POINT FROM IRQ.**

Save A,X, and Y. Then test for BRK or hardware interrupt. An indirect jump is performed according to the result of this test.
BRK. ($92) holds the vector; on setting up BASIC this is pointed to the Break entry-point of the monitor. (BASIC 1: ($021B) points to $0000, giving ?ILLEGAL QUANTITY ERROR unless a BASIC USR function is operative).
Hardware. ($90) holds the vector; on setting up BASIC this points to the IRQ servicing routine. It is this indirection which makes possible user interception of the 60-per-second interrupts.* In BASIC 1 the vector is in ($0219).

$E685	$E62E	$E455	$E455	IRQ servicing routine. Unless the interrupt is masked by SEI, or the vector is altered not to point here on an interrupt, or the interrupt is programmed not to take place, this routine is performed sixty times each second.*
$E685	$E62E	$E455	$E455	Update clock. A single JSR call updates the clock see $FFEA). 8032 BASIC has a loop to add 1 jiffy in every 7. This also checks for the Stop key, so pointing ($90) to the following routine - ($0219) in BASIC 1 - disables the Stop key (and stops the clock).
$E688	$E631	$E458	$E458	Cursor flash. Several flags are used: $A7: If non-zero, the cursor won't flash. $A8: Counts to zero; then reverses the cursor. $A9: Holds the actual character, not its reverse. $AA: Flag = 0 or 1 to indicate flash/ not flash. $E4: Repeat flag (8032 only). When>127 countdown constant = #2 so flashing is much faster.
$E6B0	$E64D	$E474	$E47A	Prepare for keyboard scan. This sets 'key image' to 'no key', 'shift key image' to off, clears the four bits 0-3 in E810, performs I/O functions (e.g. turns off the cassette motors) - details vary with ROM - and loads X with #$50, (80 decimal), ready to scan the 80 characters in the 10 by 8 decode table.
$E6F7	$E68E	$E4B5	$E4CD	Loop which scans keyboard.[2] Each key sets 1 bit low in $E812, which therefore holds only #FF, #FE, #FD, #FB, #F7, #EF, #DF, #BF, or #7F. However, it is low only when $E810 holds the correct 'row' - a value from 0-9. Thus 80 characters are possible, most of which are used. In addition, the shift key may be pressed, approximately doubling the number of keyboard characters available. $E810 is incremented during this loop; on exit it holds 9 in its right four bits, and this default value is in force when $E812 is loaded into A; this is why characters like =,<, space in non-8032 machines, and :,9,6, etc. in 8032, are often used in non-ASCII ways.

---

*12" screen CBM's interrupts occur 50 times per second.
[2] A debounce routine (a small loop) is included in the keyboard scanning routine.

BASIC1	BASIC2	BASIC4 40-col.	BASIC4 80-col.	
$E714	$E6C2	$E4E9	$E504	Process new key. The new 'key image', in $A6, is compared with the previous key, in $97. If they are the same (both may be #FF, signalling no key, or both may hold 1-80, corresponding to a character in the table).
			$E546	Routine to erase graphics characters by unsetting the high bit which shift may have set.
$E72C	$E6D6	$E4FD	$E563	Put new character into the keyboard buffer. All BASICs except the 8032 BASIC 4 delete the keyboard buffer if more than 10 characters are now present; this includes BASIC 4 in the 4016 and 4032. The 8032, however, preserves the current buffer, and in addition has a variable length buffer, where the maximum number of stored characters is PEEK(227)+1.
$E67E	$E6E4	$E600	$E600	Return from interrupt; recover A,X, and Y.
$E7AC	$E6EA	$E606	$E606	Poke contents of A into screen. Example: when A contains #2, b or B depending on the ROM mode is printed on the screen. On exit, Y holds the cursor position on its line. Note that BASICs 1 and 2 have a loop which awaits the retrace interrupt before printing. This prevents 'snow' (with the old PETs) and also slows the print. (Some other machines, e.g. Sharp MZ-80K, share this old PET feature).
			$E6A4	Ring bell twice.
			$E6A7	Ring bell once. If $E7 holds #0, this is turned off; otherwise, it is a delay constant. A table of 7 values plays the chime; these are #0E,#1E,#3E,#7E,#3E,#1E, and #0E.

KEYBOARD DECODING TABLES.

$E75C	$E6F8	$E60B	Table of 80 ASCII characters for BASIC 1, BASIC 2, and 4016/ 4032.

CONTENTS OF:

$E810 (59408)	$E812 (=59410)								
	#$7F	#$BF	#$DF	#$EF	#$F7	#$FB	#$FD	#$FE	#$FF
$-9	=	.	N/A	STOP	<	SPACE	[	RVS	N/A
$-8	-	0	RIGHT SHIFT	>	N/A	]	@	LEFT SHIFT	N/A
$-7	+	2	N/A	?	,	n	v	x	N/A
$-6	3	1	RTN	;	m	b	c	z	N/A
$-5	*	5	N/A	:	k	h	f	s	N/A
$-4	6	4	N/A	l	j	g	d	a	N/A
$-3	/	8	N/A	p	i	y	r	w	N/A
$-2	9	7	↑	o	u	t	e	q	N/A
$-1	DEL	DOWN	N/A	)	\	'	$	"	N/A
$-0	RIGHT	HOME	←	(	&	%	#	!	N/A

NOTES: i. The shift keys are detected separately and are labelled 'right' and 'left' here.

      ii. It can be seen from the table that WAIT 59410,4,4 pauses until space or shift-space is pressed, WAIT 59410,1,1 waits until RVS or RVSOFF is pressed, WAIT 59410,5,255 waits for either space or reverse, and so on.

      iii. The order of characters is the same as in the ROM table.

BASIC1 BASIC2 BASIC4 BASIC4
                40-col.   80-col.

$E6D1   Table of characters for the 8032. Note that tabled values with the high bit set have no shifted equivalents; they correspond to keys like @ and] which are marked with a single, non-alphabetic, symbol.

CONTENTS OF:

$E810 (59408)	$E812 (=59410)								
	#$7F	#$BF	#$DF	#$EF	#$F7	#$FB	#$FD	#$FE	#$FF
$-9	[20]	[4]	:	STOP	9	6	3	←*	N/A
$-8	1*	/	[21]	HOME	m	SPACE	x	RVS	N/A
$-7	2*	[16]	[15]	0*	,	n	v	z	N/A
$-6	3*	RIGHT SHIFT	[25]	.*	.	b	c	LEFT SHIFT	N/A
$-5	4*	[*	o	DOWN	u	t	e	q	N/A
$-4	DEL	p	i	\ *	y	r	w	TAB	N/A
$-3	6*	@	1	RTN	j	g	d	a	N/A
$-2	5*	;	k	]*	h	f	s	ESC*	N/A
$-1	9*	[6]	↑*	7*	0*	7	4	1	N/A
$-0	[5]	[14]	RIGHT	8*	-	8	5	2	N/A

NOTES: i.   * beside a character means that it has no shifted equivalent. Hence some characters, e.g. all the numerals, appear twice.

        ii.   Note that the contents of E812 (=59410) when E810 holds -9 are not the same as those for the earlier ROMs and 40-column BASIC 4. This is the reason for the use of different sets of keys when slowing (and pausing) screen scroll.

       iii.   The quantities in square brackets appear to be unused ASCII values.

			$E72A	Two tables of 18 constants each for CRT controller.
			$E73C	Lower case mode (switch-on) and upper case. See Chapter 9 on this chip.
$E7BC	$E748	$E65B	$E755	Table of 25 low bytes which mark the end of each screen line.
			$E76E	Table of 25 high bytes marking the start of each screen line. These are held in RAM in 40-column machines to allow alterations for double-length lines.
$E7D4	$E761	$E674	$E721	Message table. LOAD [Return] RUN [Return] or dL"* [Return] run [Return].

SYSTEM INPUT/ OUTPUT MEMORY MAP.

$E810	$E810	$E810	$E810	PIA (Peripheral interface adapter) #1.
$E820	$E820	$E820	$E820	PIA #2.
$E840	$E840	$E840	$E840	VIA (Versatile interface adapter).
			$E880	CRT controller. 8032 only.

All these addresses are incompletely decoded.

BASIC1 BASIC2 BASIC4

$F000	$F000	$F000	**Table of messages for file handling.** These are: /too many files/file open/file not open/file not found/[Rtn]searching/ for /[Rtn]press play /& record/on tape #/[Rtn]load/[Rtn] writing/[Rtn]verify/device not present/not input file/not output file/[Rtn]found /[Rtn]ok[Rtn]/[Rtn]ready.[Rtn]/. BASIC 4 has these two messages in addition: /[Rtn]are you sure ?/[Rtn]? bad disk/.
$F0B6	$F0B6	$F0D2	**Send 'Talk' on IEEE-488 bus.**
$F0BA	$F0BA	$F0D5	**Send 'Listen'.**
$F0BC	$F0BC	$F0D7	**Send 'Untalk' or 'Unlisten'.** This routine, with three entry points, handles handshaking on the bus before falling through to the next routine, where its prepared character is sent on the bus and causes the device to talk or listen or otherwise respond. What happens is as follows: the current device number (e.g. 8 for a disk) is ORed with the value that A held on entry to this routine, and which was pushed on the stack. There are four possible values: A=#40 means 'Talk', A=#20 means 'Listen', and A=#3F and #5F mean 'Unlisten' and 'Untalk'. Of these, two are set on entering the routine at the appropriate entry points; while 'Unlisten' for example requires LDA#5F/ JSR F0D7. A is ORed with the device number and placed into the IEEE buffer in $A5; it is sent from here by the next routine. Just before entering it, ATN (attention) is set low, i.e. true, so that the byte is understood as a command. Consequently, after this routine, ATN must be set high again. Note that a character in the IEEE buffer which has not yet been sent is taken care of by the present routine: $A0, the output flag, is non-zero if a character is waiting in the buffer $A5, and if this situation applies, the character will be sent before processing the IEEE command. BASIC 1's IEEE buffer is location $0222.
$F0E4	$F0E2	$F0FD	Puts A into the buffer, sets ATN true, and sends the byte.
$F0F1	$F0EE	$F109	**Send one character on IEEE-488 bus.** The character which is sent is the one previously stored in the buffer. The sequence of events is this: (i) Sets Data Valid out false; (ii) Tests for activity on the bus ; if none is found,ST is set to #80, to signal a device not present error. (iii) Loads the byte from the buffer, reverses it, because the IEEE convention is the reverse of ASCII, and stores the result in $E822, the output register. (iv) Loops while NRFD (not ready for data) is true; then sets DAV (data valid) true. (v) Sets the VIA timer and loops as
$F111	$F10D	$F128	long as NDAC (not data accepted) is true - i.e. while the byte has not been accepted by the device. If the timer reaches 65 milliseconds, ST is set to #1. This is a 'Write time out' error if it occurs. BASIC 4 has an optional override to cancel this mechanism. (v) Data valid is set false; the output register is loaded with #FF, the IEEE equivalent of a null byte.
$F12C	$F128	$F143	**Send one character and clear ATN.** This is typically used to send IEEE commands (such as the secondary address, #$60 + 0-15) when ATN is true and one command only is wanted. It is used by loading A with the character, then calling this subroutine, which stores A in the IEEE buffer, calls the routine immediately before this one, then sets attention high (false).
$F132	$F12D	$F148	Set ATN high (false).
		$F151	Optional timeout override (with Stop key test).
$F13B	$F136	$F165	**Flag errors into ST.** ST=1 (write time out), ST=-128 (device not present), and ST=2 (read time out) are processed here in three routines.
$F14B	$F146	$F175	Clear IEEE control lines.

BASIC1   BASIC2   BASIC4

BASIC1	BASIC2	BASIC4	
$E7DE	$F156	$F185	Print message from table starting $F000. This is always called from F579/ F56E/ F5AD which aborts files, prints Return and a query, and follows the message with 'error' and an optional linenumber if it's in a program. The Y offset controls the actual message.
$F15B	$F164	$F193	Send byte; then set NDAC (not data accepted) true.
$F167	$F16F	$F19E	Send IEEE character. *If* the buffer contains a character at present, that character is output, and the contents of A put in the buffer. Otherwise, the contents of A are put into the buffer, and the output flag reset from #0 to #FF. In either case, on exit flag $A0 holds #FF, and buffer $A5 holds A. (BASIC 1: $021D and $0222 respectively).
$F17A	$F17F	$F1AE	Send 'Untalk'. BASIC 4, unlike BASIC<4, sets ATN true before entering F0BC/F0BC/F0D7. This corrects a bug; see Chapter 14.
$F17E	$F183	$F1B9	Send 'Unlisten'. All ROMs function identically.
$F187	$F18C	$F1C0	Get one character from the IEEE-488 bus. The byte is returned in A. This routine uses the identical timing subroutine used to output a character; BASIC 4 again has the option of overriding the time out. ST=2 if this is not done and the device fails to return a byte within 65 milliseconds. The sequence of events is: (i) Sets NDAC (not data accepted) true, and NRFD (not ready for data) false. (ii) Waits until DAV (data valid) has been set true. ST is set =2 if the wait exceeds 65 milliseconds (but the timer can be overridden in BASIC 4, by poking $03FC (1020 decimal) with a 'negative' number). (iii) Sets NRFD true, (iv) Checks EOI; if found, ST is set to #40 (64 decimal) to indicate end-of-file. (v) Takes the byte, reverses it, and saves this value on the stack. (vi) Sends NDAC false, to indicate that the data was accepted, then waits for DAV to become true; finally, NDAC is set true again, and the byte is recovered from the stack and placed into A.
$F1CC	$F1D1	$F205	GET a byte. The jump table entry for GET – which gets a character into the accumulator without assigning it to a name– jumps here. The operation of this routine depends on the contents of $AF ($0263 in BASIC 1) which holds the input device number, for example 1 for cassette #1, 3 for the screen and 8 for a disk unit. If the device number is 4 or more, then input from the IEEE bus is assumed, and the previous routine is used. Otherwise there are three other possibilities.
$F1D6	$F1D9	$F20D	GET from the keyboard buffer.
$F1DF	$F1E1	$F215	INPUT a byte. The jump table entry for INPUT is here. Most of the logic is identical to GET – hence its position here amid GET. The difference is that input from device 0 is taken from the screen.
$F1F5	$F1F4	$F228	GET from the screen.
$F202	$F1FF	$F233	GET from cassette #1 or cassette #2. This routine is in two parts: the first reads a byte, and the next byte, so that ST may be set to #40 (64 dec.) on end of file, simultaneously with returning the last byte. The other routine is a subroutine which is called by the first routine of the two. It advances the buffer pointer and, if necessary, loads another buffer of data.
$F227	$F228	$F25C	GET from an IEEE device. This calls F187/F18C/F1C0, but only if ST=0. If the status byte holds any non-zero value, the IEEE routine is not called; instead, the carriage return character (#0D) is put into the accumulator. (BASIC 1 lacks this feature. It returns the value of ST instead).

BASIC1   BASIC2   BASIC4

$F230	$F232	$F266	**Print one character to any device.** The kernel jump table command $FFD2 jumps to this address. Like the previous GET/ INPUT routine, its operation depends on a single byte which tells it which device is to receive output. This location is the current output device number, held in $B0, or $0264 in BASIC 1. The accumulator contains the character to be output. So if $B0 holds 3, LDA #$93/ JSR $FFD2 clears the screen.
$F23D	$F239	$F26D	PRINT to screen.
$F243	$F23F	$F273	PRINT to device #>3. Uses IEEE output buffer routine.
$F247	$F243	$F277	PRINT to device #<3. This writes to tape. The tape buffer is one byte, location $B4 ($E9 in BASIC 1), from whence it is moved to the buffer appropriate to the cassette#, and, when this buffer is full, as measured by the pointer in $D4 ($F1 in BASIC 1), the buffer is written to tape. Note that the line-feed character, with ASCII value #0A (10 decimal) is trapped by this routine and cannot be written as data to tape by this PRINT routine.
$F236			?NOT OUTPUT FILE ERROR if 'output device' is #0 (i.e. the keyboard).
$F2A4	$F26E	$F2A2	**Abort all files and I/O activity.** This routine (i) Sets the number of open files flag to zero (i.e. $AE or $0262 in BASIC 1). (ii) If the output device number exceeds 3, 'Unlisten' is sent; and if the input device number exceeds 3, 'Untalk'. The files are not CLOSEd, so files being written to may be incompletely processed, and there is some risk of later corruption with disk files. The routine now performs:-
$F299	$F284	$F2B8	Restore default input and output device numbers. This simply puts #3 into the output file flag and #0 into the input device number flag. ($B0 and $AF respectively - or $0263 and $0264 in BASIC 1).
$F2AB	$F28D	$F2C1	**Search table for logical file number.** A holds the logical file number on entering this routine. If the file number exists in the table, the 'equals zero' Z flag is set, and X holds the displacement from the start of the table.
$F2B8	$F299	$F2CD	**Set file data from position in table.** On entry, X is the offset from the start of each table - as found by the previous routine. The logical file number, device number, and secondary address are all taken from their respective tables and put into $D2,$D4, and $D3 which are the current values. (In BASIC 1 these locations are $EF, $F1, and $F0.
$F2C8	$F2A9	$F2DD	**Perform CLOSE.** $FFC3 in the 'kernel' jump address table comes here. The start of this routine fetches the parameters used with CLOSE and stores the logical file number, device number and secondary address in $D2-$D4. It uses the previous routines for this.
$F2D5	$F2B6	$F2EA	$D2-$D4 are assumed set up; X holds the position of the file data in the three tables. Now the routine branches:
$F2E1	$F2C2	$F2F6	CLOSE devices #1 and #2 (i.e. cassettes). This involves writing a zero byte on the tape, and optionally an end-of-tape 'header' holding the marker value #5.
$F307	$F2E1	$F315	CLOSE devices #4 and greater (i.e. all IEEE devices). This 'Unlistens' the device; then executes the following:-
$F30A	$F2E4	$F318	CLOSE devices #0 and #3; and remove Xth item from all three file tables. This is carried out by decrementing the flag holding the number of open files; then transferring the previous last file details into the Xth position, effectively deleting the file records.

BASIC1   BASIC2   BASIC4

$F339	$F30F	$F343	**Test 'Stop' key.** $FFE1 in the 'kernel' jump table comes here. This calls an immediately preceding subroutine, then jumps to the start of the BASIC STOP and END routine. If the zero flag was set, a break occurs; otherwise, Stop wasn't pressed and BASIC continues normally. Note that all ROMs test for #EF in $E812; Stop is one of the few keys decoded in the same way by BASICs 1-4. It also follows that SYS 62275 can be used to test for Stop even if that key is otherwise disabled by a change in the interrupt vector.
$F33F	$F315	$F349	**Send file message from $F000ff if in direct mode.** BASIC 1 has an apparently unreliable test for direct mode; BASIC>1 uses a short subroutine. The message printed depends on the value in Y, which is treated as an offset. (E.g. when Y=#E, the message is FILE OPEN).
$F362	$F322	$F356	**Load a BASIC program or other RAM image.** This is *not* the BASIC entry point; this routine is called after the parameters have been input, and before the pointers are set after the load. It handles the process of fetching data into memory. (In BASIC 1 it is not fully separate from LOAD, but later BASICs have it as a separate subroutine). The device number as input with the parameters (e.g. LOAD "HELLO",2 sets the device number parameter to 2, i.e. cassette #2) determines the course of this routine:-
$F366	$F326	$F35A	?SYNTAX ERROR if device is #0 or #3.
$F36F	$F32F	$F363	**Load from any IEEE device.** A program name is assumed; its length is stored in $D1 and ($DA) points to its start. If $D1 holds zero, this routine prints ?SYNTAX ERROR. Several messages follow, each using a test for direct mode (see last-but-one routine) so the screen layout is retained with a load from within a program. The IEEE is 'Talk'ed and the secondary address sent; now the actual loading begins:
$F37E	$F348	$F37C	**Fetch data from device.** Note that BASIC 1 always sets the starting address to $0400. BASIC>1 uses the first two bytes from the bus to set the low and high bytes respectively of the starting address. In addition, BASIC 4 has a read time out defeat at this point, presumably to allow for disk read time.
$F37B	$F352	$F38C	**Print LOADING or VERIFYING if in direct mode.** If the load flag ($D4) =0, load is signalled; 1 is used for verification.
$F37E	$F355	$F38F	**Loop which loads data into RAM or verifies data already in RAM.** Firstly, an inner loop handles the input of 1 byte; it tests for Stop and repeatedly loops until no time out on read error is shown in ST. Secondly, the routine branches, depending on whether LOAD or VERIFY is being performed:
$F38E	$F36D	$F3A7	**VERIFY.** Compare byte with memory; set ST=#$10 (16 dec) if the two don't match. Then continue.
$F39A	$F378	$F3B3	**LOAD.** Store the byte in RAM. Then continue:
$F39C	$F37A	$F3B5	**Increment load address ($FB) or ($F7) in BASIC 1.** Check bit 6 of ST (EOI) and continue with loop if this is 0.
	$F387	$F3C6	**Sets the end address when LOAD or VERIFY is finished.** I.e. transfers the incremented ($FB) contents into ($C9). Also Untalks and clears channel.
$F3A5	$F395	$F3D4	**Load from cassette.** This routine is in three parts: the first sets pointers to one of the cassette buffers (which one is indicated by the device number), and prints various messages, if the mode is direct, and also waits for the cassette key to be pressed. The second part finds the header: this is simply a buffer which contains the program or file name and some other data. The third part is the actual loading / verifying into RAM. BASIC 1, again, is more confusingly written than later ROM revisions.

BASIC1   BASIC2   BASIC4

$F346	$F3C2	$F401	Perform **LOAD**. $F3C2 from the 'kernel' jump table comes here. This puts 0 into the load/verify flag.
$F34B	$F3C6	$F405	Entry point from VERIFY; flag is loaded with 1. (Note: this value must be 1; it cannot simply be a non-zero quantity, as it is used during the processing). Now, three fairly distinct operations are carried out. First, the parameters are fetched from BASIC and the current BASIC pointers are saved. The routine waits until no key on the keyboard is pressed. Secondly, the previous routine is called to LOAD or VERIFY the program. Thirdly, on return, there may be a ?LOAD ERROR, a READY. message, or, if LOAD or VERIFY took place from a program, BASIC is warm started, retaining the previous variables (up to a point-see Chapter 5 on LOAD).
$F3FF	$F40A	$F449	Print **SEARCHING** if in direct mode.
$F408	$F414	$F453	If length of string is non-zero print FOR and
$F415	$F421	$F460	print name string.
$F422	$F42E	$F46D	Print **LOADING** or **VERIFYING** if in direct mode. The actual message depends on the LOAD / VERIFY flag.
$F433	$F43E	$F47D	Fetch parameters for **LOAD, SAVE, or VERIFY**. The parameters are taken from BASIC or from the input buffer, and stored. In BASIC>1 they are: $D1=length of string and ($DA)=pointer to start of string. $D3=secondary address. $D4=device number. In all BASICs these default to 0 length, 0 secondary address, and device #1 (cassette #1).
$F45C	$F460	$F49F	Check for comma and evaluate parameter 0-255. The result is returned in the X register.
$F462	$F466	$F4A5	Send name string to IEEE-488 bus. This assumes that the secondary address and length have been put in $D3 and $D1. The device is sent 'Listen' and (if it responds) the string.
$F47D	$F47C	$F4BB	Print ?DEVICE NOT PRESENT ERROR.
$F482	$F483	$F4C0	Send name string (if it exists) and close IEEE channel.
$F495	$F494	$F4D3	Search for a named tape header block. This calls the routine to find *any* header, i.e. the next header on tape. When a header is found, its name (which starts at position 5 in the buffer) is compared with the stored name at ($DA). This process continues until end-of-tape, or until the tape runs out, or a match is found, in which case A holds the length of the name in the header (which may be shorter than the name searched for).
$F4BB	$F4B7	$F4F6	Perform **VERIFY**. $FFDB in the 'kernel' jump table comes here.
$F4C3	$F4BB	$F4FD	Check bit 5 of ST.
$F4CA	$F4C4	$F503	Print ?VERIFY ERROR and exit.
$F4CF	$F4C9	$F508	Print OK. (not from within a program).
$F433	$F4CE	$F50D	Fetch parameters for **OPEN or CLOSE**. This routine fetches the parameters corresponding to this schema: OPEN arithmetic expression [, arith. exp. [, arith. exp. [, string exp.]]]. The first of these, which is the logical file number, is compulsory; the rest are optional. In BASIC>1, these are the locations which are set: $D2=logical file number. $D1=length of string, ($DA) its pointer. $D1 defaults to 0. $D4=device number. Default = 1. $D3=secondary address. Default=0, or #FF with IEEE device. BASIC 1 equivalents are: $EF,$EE and (($F9),$F1 and $F0.

BASIC1	BASIC2	BASIC4	
$F515	$F50E	$F54D	Exit from parameter-fetching subroutine if end-of-statement.
$F51D	$F516	$F555	Check the existence of a comma followed by any character except colon or end-of-line.
$F52A	$F521	$F560	Perform **OPEN**. $FFC0 in the 'kernel' jump table comes here. This routine first uses the subroutine at F433/F4CE/F50D to fetch and store the parameters. If the file number is zero,
$F531	$F526	$F5AF	?SYNTAX ERROR is printed. If the file already exists,
$F539	$F52D	$F56C	?FILE OPEN ERROR is printed. ST is made equal to zero.
	$F537	$F576	If there are already 10 open files, the routine prints ?TOO MANY FILES and exits. (BASIC 1 has a bug at this point which causes an infinite loop - see F53B to F547).
$F549	$F539	$F578	Store the new logical file number, secondary address, and device number to the tables at 0251-025A, 0265-026E, and 025B-0264.
$F556	$F549	$F588	If device number is 0 or 3, RTS - i.e. nothing more with screen file or keyboard file.
$F563	$F556	$F595	IEEE device: send program name or string to IEEE bus.
$F566	$F559	$F598	Cassette #1 or #2. The processing here depends on the secondary address. If it is the default value of 0, tape is read; otherwise it is written to.
$F574	$F569	$F5A8	OPEN to read named file. (After WAIT: PRESS PLAY... and SEARCHING ...).
$F579	$F56E	$F5AD	Print ?FILE NOT FOUND ERROR IN ... and exit. Aborts files.
$F58B	$F583	$F5C2	OPEN to read unnamed cassette file (i.e. the next on tape).
$F592	$F58A	$F5C9	OPEN for write. This writes a header onto the tape; the header type character is #4. Also the secondary address, or #BF where this is 0, is stored for reference when CLOSEing the file - it indicates whether an end-of-tape block is to be written or not.
$F5AE	$F5A6	$F5E5	**Load next tape header.** This saves the load/verify flag on the stack, reads a block, then continues to read blocks unless the first character in the buffer is 1,4, or 5. These signal program or RAM image header, data header, and, lastly, when #5, an end-of-tape header. In either of the first two cases, FOUND with 16 characters maximum of the name is printed, if load is in direct mode. On exit, A holds #0 if an end-of-tape header was read; otherwise, A holds #1.
$F5AE	$F5DA	$F619	**Write tape header.** On entry, A holds the type-of-header byte (see previous routine's notes). Most of this routine is then occupied with putting data into the buffer. ($D6) points to the start of the cassette buffer, which is filled with spaces by a
$F5E3	$F5EB	$F62A	short loop. The following bytes are now stored in the buffer:

TYPE FLAG	LOW THEN HIGH	LOW THEN HIGH	PROGRAM NAME:
(#1,#4 OR #5)	BYTES OF (FB)	BYTES OF (C9)	LENGTH=UP TO $D1

BASIC1	BASIC2	BASIC4	
$F632	$F625	$F664	Sets the buffer start and end address and writes to tape. (FB) and (C9) point to the low byte and high byte of RAM area to be written. $C3 holds a timing value, #69, controlling the amount of tape to which a tone is written before the header proper is written.
$F64D	$F63C	$F67B	**Tape address subroutines.** When reading a tape, this first subroutine takes the start and end addresses from the bytes loaded from the header. They are put into (FB) and (C9), the start and end addresses respectively. BASIC 1: (F7) & (E5).
$F667	$F656	$F695	Sets pointer (D6) = 027A or 033A for device #1 or #2.
$F67D	$F66C	$F6AB	Sets (FB) and (C9) from (D6) - set by the previous routine - to (D6) and (D6) + #$C0 (192 decimal).
	$F68D	$F6CC	Sets (FB) and (C9) to start and end of BASIC. Used in LOAD.

BASIC1 BASIC2 BASIC4

BASIC1	BASIC2	BASIC4	
$F695	$F684	$F6C3	**Perform SYS.** $FFDE from CBM's 'kernel' jumps here. The routine evaluates any arithmetic expression, rounds it down and loads the result into ($11), if its value is within the acceptable range for SYS (0-65535). It performs an indirect jump to ($11).
$F69E	$F69E	$F6DD	**Perform SAVE.** $FFD8 from CBM's 'kernel' jumps here. This routine first inputs the parameters from BASIC by calling F433/F43E/F47D. This stores the string and its pointers, the device number, and the secondary address. Then BASIC's start address and end address pointers are transferred to the SAVE start and end pointer addresses. All the parameters are now set up for SAVEing.
$F6B1	$F6A4	$F6E0	This is the next entry point, which is used by the monitor and may be part of a user routine: all that's required is the parameters for device number, bottom and top address, pointer to name, length of name, and secondary address to be set, as though the previous two subroutines had been called. Note that the topmost address is not saved; with BASIC this doesn't matter, but machine-code dumps from RAM may very well crash if they're truncated by a byte.
$F6B5	$F6A8	$F6E7	?DEVICE NOT PRESENT if device number is #0 or #3.
$F6C0	$F6B3	$F6F2	SAVE to IEEE device. This performs the following steps: (i) Secondary address is made 1. (ii) ?SYNTAX ERROR if program name has length zero. (iii) Send name to IEEE and secondary address. (iv) Make (C7) and (C9) the start and end addresses of the part or RAM to be SAVEd. (v) Send the contents of C7 and C8. On LOAD, these are used to determine the address from which bytes will be stored in RAM again. (vi) Characters are sent one by one until the lower pointer has been incremented to equal the higher pointer. Also Stop is tested; so SAVE may be aborted by the Stop key. (vii) The channel is cleared and the device Unlistened.
$F6F6	$F703	$F742	SAVE to cassette #1 or #2. This performs the following steps: (i) Set buffer pointer to 027A or 033A depending on device #. (ii) Print PRESS PLAY AND RECORD ON TAPE #1 or 2. Wait for cassette keypress – hopefully of the correct keys. (iii) In direct mode, print WRITING and name. (iv) Write the header with type character = 1.
$F70D	$F71B	$F75A	(v) Write the contents of RAM from the lower to the higher address. (vi) If the secondary address has bit 1 on, i.e. secondary address was 2, write another header with type character = 5 to signify an end-of-tape marker.
$F736	$F729	$F768	**Update the clock / Save Stop or Reverse key.** $FFEA in the 'kernel' jump table calls this routine. In addition to adding 1 to the clock (usually), this saves the contents of $E812 in a special location, $9B or $0209 in BASIC 1. This is used as a test for the stop key; it becomes #EF if Stop is pressed. Some other keys are also detected; see the keyboard decode tables at about E600 for this. Only Stop is constant between the 8032 and other PET/CBM machines, which is why the screen scrolling is slowed by Reverse in non-8032 machines, and by the back arrow in the 8032.
$F736	$F729	$F768	Increment the correction clock; if it is #026F, reset to zero and skip the jiffy clock increment.
$F74E	$F73B	$F77A	Increment the jiffy clock. This is in $8D-$8F ($0200-$0202 in BASIC 1) with the most significant byte first.

BASIC1	BASIC2	BASIC4	
$F75B	$F745	$F784	Compare the jiffy clock's value with the constant held in the table below. If identical, reset the jiffy clock to zero by putting 0 into each byte.
$F774	$F75C	$F79B	Reset correction clock to zero.
$F77C	$F762	$F7A1	Get keyboard PIA value (with debounce) and store it: i.e. fetches contents of $E812 and puts result in $9B or $0209.
$F788	$F76D	$F7AC	Table of three bytes. they are #4F, #1A, #01. This is the constant for 24 hours. 79*256² + 26*256 + 1 = 5 184 001, which is 24 hours of 1/60th sec + 1 extra 60th second.
$F78B	$F770	$F7AF	**Set Input device.** $FFC6 of the 'kernel' jump table comes here. On entry, the X register holds the logical file number of an open file. This makes an IEEE device a Talker. See the description under FFC6 in the table of common kernel routines for details. BASIC 4 is slightly different from BASIC<4, since it clears DS$ in addition to ST. The DOS wedge program's source listings refer to this routine as 'Check out'; three errors can occur, causing ?FILE NOT OPEN, ?NOT INPUT FILE, or ?DEVICE NOT PRESENT error messages to appear.
$F7DC	$F7BC	$F7FE	**Set Output device.** $FFC9 of the 'kernel' jump table comes here. On entry, X holds a logical file number. The IEEE device corresponding to this file becomes a Listener, provided no error is detected. This routine, like the previous one, can print one of three possible errors, which are ?FILE NOT OPEN, ?NOT OUTPUT FILE, and ?DEVICE NOT PRESENT. See the description under FFC9 in the table of CBM kernel routines for more details. Note that this latter pair of routines, 'Check in' and 'Check out', only actually influence the peripheral device when this is on the IEEE bus, i.e. with device number greater than 3. Tape, screen, and keyboard files are merely validated, and the current input device number or output device number is set so that prompting messages and so forth are not printed where this would be superfluous. The input device number is stored in $AF or $0263, and the output device number in $B0 or $0264 (BASIC>1 and BASIC 1 respectively).

CASSETTE TAPE OPERATING SYSTEM

BASIC1	BASIC2	BASIC4	
$F82D	$F806	$F84B	**Increment tape buffer pointer.** This short subroutine (i) sets ($D6) to 027A or 033A, depending on the device number in $D4; (ii) increments either $BB or $BC, again depending on $D4; (iii) compares the result with #C0 (192), so that if Z is set on return from the subroutine, the pointer points to the end of the buffer.
$F83B	$F812	$F857	**Test cassette keypress.** If a cassette key is pressed when this routine is entered, nothing further is done; exit immediately takes place. Otherwise, PRESS PLAY ON TAPE #1 or 2 is printed to the screen. Now a loop is entered; this repeatedly
$F851	$F828	$F86D	tests the Stop key and the cassette. A cassette keypress prints OK and the routine is finished; the Stop key of course aborts the tape read.
$F85E	$F835	$F87A	**Test cassette keys subroutine.** If the Z flag is set, a key is pressed; if not, not. So for example LABEL JSR F87A/ BNE LABEL loops indefinitely until a key is pressed. $E810 is tested for bit 4 (tape #1) or bit 5 (tape #2) high.
$F871	$F874	$F88C	**PRESS PLAY AND RECORD...** This is identical to the routine which tests the cassette for a keypress, following this with PRESS PLAY ON TAPE ... except that an additional message, AND RECORD, is interpolated before entering the earlier routine.

BASIC1   BASIC2   BASIC4

$F87F	$F855	$F89A	**Read Tape.** On entry, (FB) and (C9) point to the low and high locations in RAM into which tape data is to be loaded. (F7) and (E5) are BASIC 1's equivalents. This subroutine can be used to read any blocks from a CBM tape.
$F87F	$F855	$F89A	Sets ST=0, sets load/verify flag to load, and sets (FB) and (C9) according to device number 1 or 2 in $D4 or $F1 with BASIC 1, to 027A-033A or 033A-03FA. So exactly one block will be read into one or other cassette buffer. This routine loads a header for F5AE/F5A6/F5E5; the block can be identified as a header by its first byte, #1,#4, or #5.
$F88A	$F85E	$F8A3	This routine skips the call which sets (FB) and (C9) to point to a cassette buffer. It loads data direct into RAM from tape. It does not read blocks of data, but consecutive bytes written one after another. If there is a checksum or other error the second copy of the program on tape is likely to be able to correct the load; the probability of an unrecoverable error increases with program length.
$F890	$F864	$F8A9	Initialise for tape read. The interrupt is masked and these locations zeroised: $C0,$C1,$C2,$CB,$CE and $B2
$F8A8	$F873	$F8ED	With cassette #1, CA1, and with cassette #2, CA2 is enabled.
$F8A3	$F882	$F8C7	X is loaded with #0E and the tape read/write routine FD1B/F89B/F8E0 entered. #0E corresponds to the fourth interrupt vector from the table of tape interrupts.
$F8B9	$F886	$F8CB	**Write Tape.** This first entry point is used to write data files to tape; it sets addresses (FB) and (C9) from the cassette device number, then:
$F8BC	$F889	$F8CE	Set inter-block time counter, $C3, to 20 decimal, then:
$F83B	$F890	$F8D5	Write consecutive bytes from RAM to tape. This entry-point is used to write headers; however, a larger value, 105 decimal, in the counter $C3 ensures a longer delay than obtains with a block of data. Timer #2 in the VIA is interrupt-enabled; this is used to control the timing of writing onto tape. X is loaded with #8 (corresponding to the first interrupt vector from the table of tape interrupts) and the following is performed:
$FD1B	$F89B	$F8E0	**Tape read/write subroutine.** This is shared by both of the other routines on this page. It sets a new IRQ; with Read, * the vector is F95F/F931/F976 (when offset X=#0E), and with * Write FCCF/FC54/FC99 (when X=#08). This interrupt will be made active when the interrupt mask is cleared with CLI. $E813 is decremented; this, like POKE 59411, disables the normal retrace interrupt; only the cassette interrupts are now enabled. Also, several counters and flags are initialised.
$F8DC	$F8A8	$F8ED	Now the cassette motor is switched on. This involves setting 1 bit low; the process is the reverse of FFED/FCA6/FCEB, which turns both motors off.
	$F8C0	$F905	1/3rd second delay for motor to pick up speed. (Omitted in BASIC 1, which helps explain its greater unreliability).
$F8F6	$F8CA	$F90F	Sets timer #2, clears the interrupt disable mask, and waits for the IRQ vector to be reset to normal. It tests the Stop key and also updates the clock when the retrace interrupt flag is high, so the clock (and the Stop key test) are both updated in the normal way. The processing is done during the interrupts, at the addresses listed above; see locations marked *.

BASIC1	BASIC2	BASIC4	
$F913	$F8E6	$F92B	Await IRQ's return to normal; test Stop key and action it if it's pressed. This tests Stop with the usual routine, but mod-
$F91E	$F8F0	$F935	ified because the tape and interrupt vector need to be aborted. In addition the IRQ vector is examined - in fact its high byte only, which is enough - and the 'equals zero' flag is set true when this happens.
$F92E	$F900	$F945	Set VIA timer #1 to new value, synchronized with timer #2 becoming zero. Timer #1 is set to a multiple of the contents of $CB 0r $E7 with BASIC 1, and the entry value of X. The timing is uesd by the tape read routine.
$F95F	$F931	$F976	**Read bits from tape: interrupt entry when table offset = #$0E.** After a timed delay, this routine sets timer #2 to a 65 milli- second cycle and and reads the tape. $DF builds up a byte by rotating each bit in consecutively. When 8 bits have been input, the 'byte received' flag in $B2 is set, and the byte stored in RAM.
$FAA3	$FA57	$FA9C	**Store bytes in RAM.** This stores characters from (FB) up to (C9). It also performs the tape error checking :
$FAC5	$FA76	$FABB	Flag 'Long Block' error into ST, setting bit 2 of ST (=8 dec.).
$FB11	$FABB	$FB00	Flag 'Short Block' error into ST, setting bit 3 of ST (=4 dec.).
$FB23	$FACF	$FB12	Comparison and error-storing routine, which puts errors into the low end of the stack ($0100 ff.) and compares, where necessary, on the second read.
$FB8B	$FB2B	$FB70	Flag 'Unrecoverable Read Error' into ST, setting bit 4 of ST (=16 decimal).
$FBDC	$FB76	$FBBB	Puts (FB) into (C7) - header pointer back to start of buffer.
$FBE5	$FB7F	$FBC4	Flag contents of A into ST byte.
$FBEC	$FB84	$FBC9	Reset flags for new byte. Sets various flags to 0, and the bit counter $B7 - 026C in BASIC 1 - to #8.
$FC00	$FB93	$FBD8	**Write a tone to tape.** Timer #2 is loaded with a value, and the cassette output bit is reversed by EOR. This is used to write a bit to tape. The timer is loaded with #0060, #00B0, and (next routine) #0110, giving different frequencies. Within this routine, the timer's value depends on the contents of $DD ($FC in BASIC 1). If this is even, #60 is loaded; if it is odd, #B0.
$FC21	$FBB4	$FBF9	**Write bits to tape: interrupt entry when table offset = #$0A.** This uses the previous routine to write a tone onto tape. The actual frequencies used for bits 0 and 1, and details of the parity bit and inter-byte marker, are quoted in 'The PET Revealed' on pp. 136-7.
$FCCF	$FC54	$FC99	**Write a block to tape.** This is called from 'Write header'. It starts by loading the timer with #$78 and writing the corre- sponding frequency to tape. It resets the flags and delays, for a length of time corresponding to the counter in $C3, which is decremented with every interrupt. Finally, the IRQ vector is replaced by that corresponding to an offset of #0A from the table of IRQ vectors, at FC21/FBB4/FBF9. When the interrupt disable flag is set to 0, this routine performs writing during interrupts; meanwhile, $DE counts the blocks remain- ing to be written.
$FCFB	$FC7B	$FCC0	Turn off motor/ turn off abnormal interrupts / restore IRQ. VIA interrupt enables are returned to normal, i.e. produced by the retrace interrupt.
$FFED	$FCA6	$FCEB	Turn off motors. (In BASIC 1 this runs into the NMI vector).
$FD7C	$FCB4	$FCF9	Perform checksum and increment pointer. Checksum is EOR of all bytes and is stored in $C3 (($0279 in BASIC 1).
$FD90	$FCC6	$FD0B	Check whether low address has been incremented as far as the high address; if so, the 'equals zero' flag, Z, is set true.

BASIC1   BASIC2   BASIC4

$FD38	$FCD1	$FD16	**Power-on Reset.** (FFFC) in the 6502's hardware vectors points to this address; this is the first software activity which occurs on switching on the macine. It may also be called during the course of a program; for example, SYS 64824/64721/64790 from BASIC clears RAM and resets all major pointers (it does, however, also leave the cassette buffers and some other RAM untouched). There is an optional branch when this routine is called: if bit 7 of $E810 is high (this line is called the 'diagnostic sense line' because of its behaviour in BASIC 1), the monitor is called rather than BASIC. This is the reasoning behind one of the reset switches of Jim Butterfield, which relies on simultaneously holding $E810 high and pulling the reset line to the 6502 low. In this way, BASIC is preserved. BASIC 1 does not have the monitor; the branch executes a diagnostic routine, which was dropped from later BASICs. It required some wiring to the ports (see e.g. 'The PET Revealed' for an account of this. Note that the keyboard connector too has to be disconnected from the main board and wired up).

The following operations are performed:
(i) Set the stack pointer to #FF (i.e. top of the stack); clear the decimal flag, which may be set; set the interrupt disable flag, in BASIC 4 – a precaution which the other ROMs don't take; set the I/O registers, calling E1E1/E1D1/E000 – and, in the 8032, tinkling the bell; this also sets the IRQ vector.
(ii) Point NMI to –/C389/B3FF, where it will simply print READY. (BASIC 1 has no NMI vector in RAM).
Point the BRK interrupt to the B* entry point of the monitor. (Not in BASIC 1).
Point USRCMD of the monitor to print ? when an unrecognised monitor command is entered.
(iii) Clear the interrupt flag and jump to initialise BASIC or exceptionally to C* in the monitor, if the diagnostic sense line is high.

	$FCD8	$FD1E	Set vectors except BRK.
$FD3F	$FCF3	$FD3E	Test PIA for diagnostic sense.
$FD9B			Diagnostic routines of BASIC 1. These are activated in a wired up PET on switchon. Like the tape routine they use interrupt vectors rather freely; the table of interrupts in BASIC>1 has gaps left over in it apparently because of this. Oddly, if an error is detected, the routine goes into an infinite loop; there are many of these in ROM. Some memory routines are usable without hardware; for example,* the entire contents of FDDD-FDFA (64989-65018) may be moved to RAM, with an RTS poked at the end; this routine will perform a checksum on ROM from C000-E7FF and F000-FFFF, the result of which should be zero, held in $0279 (633 decimal).
	$FCF8	$FD43	In BASIC 2, Monitor entry is FD11, BASIC initialisation E116. In BASIC 4, Monitor entry is D472, BASIC initialisation D3B6.
	$FCFE	$FD49	NMI vector. BASIC>1 performs indirect jump to (0094).
$FD28	$FD01	$FD4C	Table of IRQ vectors for tape handling routines (and diagnostics in BASIC 1).
	$FD11		Machine-language monitor. In BASIC 4 this has been moved to D472, so I have put its ROM details there, following the sequence of BASIC4 rather than BASIC 2.
	$FFB1		Millenial copyright statement: C 0978 CBM.

---

*This suggestion was made in an IPUG newsletter.

BASIC1 BASIC2 BASIC4

The 'Kernel': Disk routines only.

		Jump address:
$FF93	CONCAT. Jump address:	DAC7
$FF96	DOPEN	D942
$FF99	DCLOSE	DA07
$FF9C	RECORD	D7AF
$FF9F	HEADER	D9D2
$FFA2	COLLECT	DA65
$FFA5	BACKUP	DA7E
$FFA8	COPY	DAA7
$FFAB	APPEND	D977
$FFAE	DSAVE	DB0D
$FFB1	DLOAD	DB3A
$FFB4	CATALOG or DIRECTORY	D873
$FFB7	RENAME	DB55
$FFBA	SCRATCH	DB66
$FFBD	GET DS$ (DISK STATUS)	D995

$FFC0 to $FFEA ----   Kernel jump table: see next page.

BASIC1	BASIC2	BASIC4	
$F52A	$F521	$F560	$FFC0 (OPEN) - jump address.
$F2C8	$F2A9	$F2DD	$FFC3 (CLOSE) - jump address.
$F78B	$F770	$F7AF	$FFC6 (Set input device) - jump address.
$F7DC	$F7BC	$F7FE	$FFC9 (Set output device) - jump address.
$F27D	$F272	$F2A6	$FFCC (Restore default I/O) - jump address.
$F1DF	$F1E1	$F215	$FFCF (Input a byte) - jump address.
$F230	$F232	$F266	$FFD2 (Output a byte) - jump address.
$F346	$F3C2	$F401	$FFD5 (LOAD) - jump address.
$F69E	$F69E	$F6DD	$FFD8 (SAVE) - jump address.
$F4BB	$F4B7	$F4F6	$FFDB (VERIFY) - jump address.
$F695	$F684	$F6C3	$FFDE (SYS) - jump address.
$F339	$F30F	$F343	$FFE1 (Test Stop key) - jump address.
$F1CC	$F1D1	$F205	$FFE4 (Get 1 character) - jump address.
$F2A4	$F26E	$F2A2	$FFE7 (Abort all I/O) - jump address.
$F736	$F729	$F768	$FFEA (Update clock/ store key) - jump address.

$FFED	Turn cassette motor(s) off. This subroutine ends with RTS, #$60 in hexadecimal, which 'mangles' the low byte of the NMI vector.
---- ($FFFA) ----	**NMI vector:** BASIC 1: $CA60 enters a routine to print a character. BASIC 2: $FCFE BASIC 4: $FD49
---- ($FFFC) ----	**Reset vector:** BASIC 1: $FD38 BASIC 2: $FCD1 BASIC 4: $FD16
---- ($FFFE) ----	**IRQ vector:** BASIC 1: $E66B BASIC 2: $E61B BASIC 4: $E442

The 'Kernel': Routines common to all CBM BASICs.

FFC0 **OPEN** Identical to BASIC OPEN.

FFC3 **CLOSE** Identical to BASIC CLOSE.

FFC6 **SET INPUT DEVICE** LDX #logical file number/ JSR $FFC6 prepares the logical file number in X for input. The routine preserves A,X, and Y; file details of device and secondary address are taken from the tables. In the case of IEEE devices, 'TALK' is sent. This routine also checks that the file is ready: if the file isn't open, ?file not open error results; if the file is open to tape with non-zero secondary address, ?not input file results; and if the high bit of ST is set, ?device not present error results. (Other IEEE errors may be reflected in ST and DS).

FFC9 **SET OUTPUT DEVICE** LDX #logical file number/ JSR $FFC9 prepares the logical file number in X for output. The routine preserves A,X, and Y; file details of device and secondary address are taken from the tables. In the case of IEEE devices, 'LISTEN' is sent. This routine also checks that the file is ready; if the file hasn't been opened, ?file not open error will appear; if the file is open to the keyboard or to a cassette file with zero secondary address, ?not output file results; and if the high bit of ST is set, ?device not present error results. (Other IEEE errors may be reflected in ST and DS).

FFCC **RESTORE DEFAULT I/O** Makes the output device 3 (i.e. screen) and the input device 0 (i.e. keyboard). The locations involved are $B0 and $AF respectively ($0264 & $0263 in BASIC 1). In addition, an output device on the IEEE is unlistened, and an input device sent the untalk command. None of these files are closed by the routine. Note that X and Y registers are preserved. (BASIC<4 has a bug: see Chapter 14 on the IEEE bus).

FFCF **INPUT ONE CHARACTER** This routine is a subset of INPUT and INPUT#, which takes in a single character and – in the case of screen input – prints a flashing cursor and advances the cursor on input. (Some monitor source code calls it 'RDT'). It sets ST to zero, then, according to the input device number ($AF or $0263) separates into keyboard/ cassettes/ screen/ or IEEE routines. In each case on return A holds the input character. Note that the contents of X and Y are unchanged. If ST<>0, IEEE devices return #0D.

FFD2 **OUTPUT ONE CHARACTER** This routine is called by PRINT and PRINT#; it outputs the contents of A to any device. In the case of the screen, the character is treated as CBM 'ASCII', so e.g. LDA #$93/ JSR $FFD2 clears the screen, if the output device is 3. The output device (location $B0 or $0264) determines whether cassette, screen, or IEEE output obtains. The contents of A,X, and Y are preserved.

FFD5 **LOAD** Identical to BASIC LOAD.

FFD8 **SAVE** Identical to BASIC SAVE.

FFDB **VERIFY** Identical to BASIC VERIFY.

FFDE **SYS** Identical to BASIC SYS.

FFE1 **TEST STOP KEY** JSR $FFE1 (that's all!) within machine code tests the key-image stored by the clock update routine to see whether STOP was pressed. If so, files are aborted (effectively by FFCC) and READY appears. If $9B ($0209 in BASIC 1) is forced to #FF, this will no longer work.

FFE4 **GET ONE CHARACTER** Almost identical to FFCF except that keyboard input is taken from the keyboard buffer. So that, for example, the equivalent of 10 GET X$: IF X$="" GOTO 10 is JSR $FFE4 / BEQ -5. X and Y are retained.

FFE7 **ABORT ALL I/O** Sets the number of files open to zero, then in effect calls FFCC. The files are not closed. CLR and similar routines call this.

FFEA **UPDATE CLOCK AND STORE KEYPRESS** increments the jiffy clock (unless the correction clock resets) and saves the keypress which FFE1 uses.

CHAPTER 16: MATHEMATICAL PROGRAMMING

16.1 Computation

Accuracy   What do these three expressions have in common?

         .55 + .32 = .87
         3/5 * 5   = 3
         .01 + .05 = .06

The answer - which applies to those machines (PET, Apple, etc.) using Microsoft BASIC of single-precision accuracy - is that they are all false. At first sight this is a disturbing fact, but all computers with digital storage have this problem, so there is no need to be over-concerned. When I say that all computers have this problem, I mean that the difficulty is inherent in these machines. A number like 1/3, for example, which is expressible as an exact fraction, can't be stored as an expansion in decimal or binary form without losing accuracy. Large machines, or those with 16-bit chips or more precision in the way they store numbers, are less prone to problems of this sort than small ones, but the basic difficulty remains. Chips which perform only calculation, as used in the DAI for example, have internal registers which indicate when a result has been rounded, and also the lower and upper limits of the result. To ensure that no problems are met with in programming the CBM, we need to examine the number storage system. Leaving aside integers, with which no loss of precision is possible, floating-point values are stored in RAM as 5 bytes. Chapter 4 shows how numbers are stored both as variables and in the floating-point accumulator. The accuracy is greater in the accumulators than in variable form, because an extra byte is used to retain values which are later rounded when the result is stored further up in RAM as a variable's value. Such values are stored like this:

BYTE 1	BYTE 2	BYTE 3	BYTE 4	BYTE 5
EXPONENT	SIGN BIT AND MANTISSA 1	MANTISSA 2	MANTISSA 3	MANTISSA 4

This is a very standard arrangement, in which every increase in the exponent (byte 1) doubles the value, and where the mantissas are arranged in decreasing order of significance. A single bit holds the sign. To make this clear, let's consider some examples. The non-mathematically minded may like to skip this section (and probably the entire chapter too), although I don't recommend this.

Firstly, the exponent. If a variable, say X, is put equal to a number (X=3, for instance), the five bytes which store the number can be peeked from RAM, and we can attempt to decode what we see. X=3 and X=6 give these results:

3	130	64	0	0	0
6	131	64	0	0	0

The only difference between 3 and 6 as stored is in the exponent. A difference of 1 doubles/ halves the result. The exception is a zero exponent; the value is then zero.

Secondly, the sign bit. Two other specimen values give these results:

-1.5	129	192	0	0	0
1.5	129	64	0	0	0

The sign change is signalled by the high bit of Mantissa 1. This of course corresponds with the minus flag on the 6502, which makes for easier programming. There is no point in taking up more space than 1 bit, since a sign has only two alternative values.

Thirdly, the mantissas. Because the highest bit is used for the sign, the number's value is stored in 4 bytes less 1 bit, making 31 bits in all. The significance of these bits is hard to explain: they span a range from 1 to 1.999999... which, when multiplied by an exponent, itself of form 2^n, takes in the entire range from about 10^{-38} to 10^{38} with accuracy of 1 part in 10^{10}. The following formula will convert any numeral stored in this way into its numeric equivalent:

(-1) $\uparrow$ (M1 AND 128) * 2 $\uparrow$ (EXP-129) * (1 + ((M1 AND 127)+(M2+(M3+M4/256)/256)/256)/128)

The following examples may make the meaning of this formula less obscure. They cover a limited range, from 3 to 8:-

3	130	64	0	0	0
4	131	0	0	0	0
5	131	32	0	0	0
6	131	64	0	0	0
7	131	96	0	0	0
8	132	0	0	0	0

Numbers between 4 and 7.9999... have the same exponent, 131. The sets of figures here recur for the whole range of exponents, giving 8,10,12,14 next, 2,2½,3,3½ previously, and so on. The exponent, minus 129, and raised to the power 2, multiplied by the mantissa, gives the result, and we can see from the examples of 4 and 8 that a constant 1 is added to the mantissas. The following bits are weighted, so that division of the bytes by 128,256,256, and 256 successively scales down the final bit to an appropriately tiny value, and assigns the greatest weight to the earliest bits. The examples are translated like this:

$$3 = 2 \uparrow 1 \quad \text{multiplied by} \quad 1 + 64/128 \quad = 2*1\tfrac{1}{2} = 3$$
$$4 = 2 \uparrow 2 \quad \text{"} \quad 1 + 0 \quad = 4*1 = 4$$
$$5 = 2 \uparrow 2 \quad \text{"} \quad 1 + 32/128 \quad = 4*1\ 1/4 = 5$$
$$6 = 2 \uparrow 2 \quad \text{"} \quad 1 + 64/128 \quad = 4*1\tfrac{1}{2} = 6$$
$$7 = 2 \uparrow 2 \quad \text{"} \quad 1 + 96/128 \quad = 4*1\ 3/4 = 7$$
$$8 = 2 \uparrow 3 \quad \text{"} \quad 1 + 0 \quad = 8*1 = 8$$

To decode a number, using a pocket calculator (or a PET!) the easiest method is to start at the lowest byte, divide by 256, add the next, divide by 256, add the next, divide by 256, and finally add mantissa 1, less 128 if it exceeds 127, divide by 128 and add 1. The result, between 1 and 1.99999... must then be scaled up or down according to the exponent, and assigned positive or negative sign. This is what the formual on the previous page does. Here are some further examples, with their correct values at the foot of the page, for those who want to test their grasp of the idea:

125	124	185	35	163
136	16	192	0	0
155	62	188	31	224

Conversely, let's see how to express a real number in floating-point form. Let's take -13.2681 as an example. The minus sign means we must set the high bit of mantissa 1. 8 is the nearest power of 2 to 13.2681, as $2^3 = 8$. So the exponent is 129+3=132.
13.2681 = 8 * (1+5.2681/8) = 8 * (1+ .6585125). The value following 1 is the number which is to be approximated by the mantissa:

$$.6585125 * 128 = 84.2896,$$
$$.2896 * 256 = 74.\overline{1376},$$
$$.1376 * 256 = \overline{35}.2256, \text{ and}$$
$$.2256 * 256 = \overline{57}.7536.$$

Therefore, the nearest approximation is:

| 132 | 212 | 74 | 35 | 58 |

Section 16.9 gives machine-code routines to perform these conversions; this is useful when finding the values stored in ROM tables and in RAM. We can put this knowledge to work in avoiding rounding errors. Since 31 bits store the value of any number, a numeral which does not overrun the final bit will be held exactly. Integers up to about 2^{31} are held without loss of accuracy. The extreme value may be found by tests like PRINT 2 150 000 000 = 2 150 000 001,until the answer is -1, meaning 'true', for a large enough number, at which point the smallest bit no longer distinguishes the very last figure. Calculations on exact integers (i.e. from about -2^{31} to 2^{31} are, so far as I know, accurate, when addition, subtraction, multiplication and division are involved, although obviously functions such as SIN or LOG will involve rounding error. This is the reason that loops with integral values execute correctly. FOR J=1 TO 100000: NEXT executes correctly up to the last value, which is stored as though J=100000 had just been entered. Decimals are less immediately obvious. There will be errors if the fraction is not some combination of ½, 1/4, 1/8, 1/16,..., 2^{-31}. A loop with step-size 1.5, or 2.75, or 7.125, or .00390625, will execute correctly; but 3.4 or 9.26 or 12.87 will

---

Decimal values are .1234, 144.75, and 99999999 respectively.

not, since these are stored as repeating numbers in the binary system, like 1/3 in the decimal system. There is also a problem of the relative magnitude of the numbers concerned. .00390625 (=1/256) is stored exactly; but if it is added to a huge integer, which also is stored exactly, some precision is lost because of the difference between exponents of the two numbers. Some of the smaller number's bits will be lost; in effect it will be truncated.

As an example of the application of this knowledge, the program which follows, and a specimen run, divides two decimal numbers, printing the result with no loss of accuracy. The program run divides 123 by 19, printing 75 decimal places of the result. Its BASIC looks more complex than is really the case: line 20 checks the input values of numerator and denominator; lines 30 and 50 are complicated by the need to delete spaces before numbers, a frequent irritant with PET BASIC. Line 30 prints the number preceding the decimal point; line 40 updates the numerator, removing the equivalent of the part just printed by line 20 or 50; and line 50 prints a single digit. The lines with N stop the loop after N decimal places. The point is that no error is introduced by the calculations in 30 - 50.

INDEFINITE PRECISION DIVISION:

```
10 INPUT "X,Y,N"; X,Y,N
20 IF X > 21E8 OR Y > 21E8 OR Y <> INT(Y) OR X<>INT(X) THEN PRINT "ERROR"; END
30 PRINT INT(X/Y); "[LEFT].";
40 X = (X - INT(X/Y)*Y) * 10
50 PRINT MID$(STR$(INT(X/Y)),2);
60 N = N-1
70 IF N>1 THEN 40
READY.
```

```
EXAMPLE
       X,Y,N? 123
?? 19
?? 75
6.473684210526315789473684210526315789473684210526315789473684210526631
```

This decimal repeats every 18 digits (=19 - 1). This is a consequence of the fact that 19 is a prime number. The sequence of digits repeats indefinitely, and this process is similar to that by which pseudo-random numbers are generated, in which each number is derived from the previous, and the sequence repeats, but its cycle is large enough for the numbers to be considered 'random'. As a more ambitious example, this decimal repeats every 330 digits (45678 = 2*3*23*331).

```
X,Y,N? 12345
?? 45678
?? 75
0.270261394982267174569814790489951398922895047944305792722973860501773
```

If a calculation can be subdivided in this way, there is no risk of rounding error. But numbers may simply be too great to be susceptible to this approach. Multiplication is a good example; see DBL (Chapter 5) on this. Section 16.9 has a machine-code multiplication program accurate to 250 figures. Not many accountants believe their figures to be accurate to a few parts in ten billion, and it is often unnecessary to bother with large-figure accuracy. Small figures, paradoxically, may give trouble. I can remember a demonstration of an incomplete records accounting system which, at the end of a long period of input of figures, announced that the totals didn't balance. The balancing amount was revealed to be zero. Obviously the figure was held as .00000001 or something similar, and a test for 'equality' was used which required all the bits to be identical. The result should have been rounded to the nearest penny/ cent. Some computers (e.g. DEC) have an 'approximate equality' function, definable by the user so that two values are considered to be equal if they are within a certain small value of each other. The PET equivalent is ABS (X-Y) < .0001, or whichever value is selected. Accuracy has to be considered in any serious system involving numbers. New ROM issues of BASIC promise to have more precise arithmetic available, using the 6502 type chip's decimal mode; details of this are not available, but presumably decisions

will have to be made on which variables should use this mode, and perhaps how many bytes ought to be allocated. In the case of current CBM it may be worth performing arithmetic in integers only (i.e. floating-point numbers having integer values). Some calculations yielding non-exact values, for example percent markups, will have to be rounded at each stage. A trial program will show whether this is necessary. With luck it won't be. Some calculations though, which are within CBM equipment's capacity, can come past the point where small cumulative errors show: stocktaking ('inventory' in the U.S.) may show small errors between totals and subtotals.

In mathematical work, the rule should be to prefer methods which yield a result with the smallest relative error, where there is a choice. Three examples, involving the statistical concept of standard deviations, the solution of quadratic equations, and series summations should illustrate this idea adequately:

(i) <u>Standard deviation</u>. This statistical concept is related to the normal distribution and to the idea of the average or 'mean'. It is a measure of variation, equal to the average of the sum of squares of deviations of each item from the joint mean. Let's take a simple example: two numbers only have been taken, the result of some measurement, and these are 1000 and 1001. We assume they are completely accurate. Their mean is 1000.5, which we can assume is the mean of their total distribution: all we are interested in here are computational problems. The deviations from the mean are $-.5$ and $.5$; the squares of the deviations are $.25$ and $.25$; so the average of these deviations is $\frac{1}{2}(.25 + .25) = .25$. There is obviously no error in this result ascribable to the computing technique used. However, by standard algebra, we can prove this general result:

$$s.d.^2 = \frac{1}{n} * \Sigma(x-\overline{x})^2 = \frac{\Sigma(x^2)}{n} - \overline{x}^2$$

which implies that $\frac{1}{2}(1000^2 + 1001^2) - 1000.5^2$ can be used in routine calculations to evaluate the standard deviation. The equations are algebraically unimpeachable; but the calculation which results falls into the trap of producing a result by subtracting one number from a very similar number; in this case, each number is large, and it is easy to see that rounding errors might cast doubt on the result:

$$s.d.^2 = 1001000.5 - 1001000.25 = .25 \quad \text{... this time}$$

(ii) <u>Quadratic equations</u>. These are a common, rather boring, source of demonstration programs. Leaving aside the questions of imaginary solutions and repeated solutions, the general solution of $ax^2 + bx + c = 0$ is:

$$x = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$$

From the viewpoint of rounding errors, it is best to use the form with the negative square root, because otherwise, if a or c is a small value, the absolute values of the two expressions in the numerator will be close, and a large relative error will result after subtraction. The other solution can be found using the fact that, in

$$x^2 - px + q = 0,$$

both solutions add to p.

(iii) <u>Series summations</u>. The short BASIC program sums a well-known series,
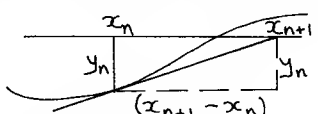
```
10 INPUT "VALUE";V
20 N=1: T=V:REM T=TERM
30 PARTIAL RESULT=PA+T
40 T = - T*V*V/(N+1)/(N+2)
50 PRINT SIN(V),PA
60 N = N+2: GOTO 30
```

$$SIN(V) = V - \frac{V^3}{3!} + \frac{V^5}{5!} - \frac{V^7}{7!} + \frac{V^9}{9!} - \dots$$

This series converges for any value of V, since the ratio between numerators increases with V^2 each time, whereas the denominators' ratio increases with every term without converging. If each term of the series is held by a computer with complete accuracy, SIN(10000) will be evaluated as accurately as, though more slowly than, SIN(.1). When this is not the case, as the program shows with values of 50 or 100 say, the result will be more or less swamped by the magnitude of individual early terms. The solution is to transform the value into some number which is easier to deal with; here, because of the cyclical nature of the sine curve, it is easy to subtract an appropriate multiple of pi from the original value, then compute the sine of the result. This is how CBM BASIC works, with the result that rounding errors with large arguments are greatly reduced, in fact becoming equivalent to the rounding error involved in the subsidiary calculation.

Transformations are common in statistical work; the standard deviation illustration above lends itself to a particularly easy type, since it can be shown that adding a constant to every value leaves the standard deviation unchanged. So instead of using 1000 and 1001, we can calculate the standard deviation of 0 and 1.

Solving equations: Newton's method   This is one of very many methods to discover solutions by iteration rather than analysis. Like all such methods, it is fallible – since ingenious exceptional functions can be found which can't be solved. The principle it uses is to improve on a guess using knowledge of a function (which may be pictured as a graph) and its gradient. Repetition of the process gives a set of approximate solutions: if these converge, so that consecutive trials are equal or nearly equal, a solution is presumed to have been found. Generally, the user has to supply a starting guess, although this could be done by the machine at the expense of computing time. Some Hewlett-Packard calculators have a 'Solve' function, where two estimates are asked for. The iterative relation is

$$x_{n+1} = x_n - \frac{y_n}{\text{gradient at } x_n}$$

Gradient at (x_n, y_n)
$$= y_n / (x_{n+1} - x_n)$$

The gradient is usually assumed to be the derivative, i.e. an analytically-found curve giving the gradient at all points; the BASIC routine below calculates the gradient, rather than requiring a supplied formula, so that expressions which are hard to differentiate are still solvable. Line 10 holds the function definition; line 40 calculates the gradient, using an arbitrary value for dx which may be changed; line 50 adjusts the best estimate so far; and line 60 stops if and when the improvement in the estimate is negligible.

```
1 REM ***********************************************************
2 REM ****    NEWTON'S METHOD FOR SOLVING SMOOTH FUNCTIONS    ****
3 REM ****              LINE 10 HOLDS THE FUNCTION            ****
4 REM **** CAN USE EG 10 LOG(X^N)-SIN(X),IF N IS ALLOWED FOR ****
5 REM ***********************************************************
6 REMEMBER DESCARTES' SIGN RULE: NO. OF CHANGES = NO. OF POSITIVE SOLUTIONS
10 DEF FN Y(X) = LOG(X) - SIN(X): REM DEFINE FUNCTION TO BE SOLVED = 0.
20 INPUT "GUESS"; GUESS          : REM USER MUST SUPPLY GUESS
30 DX = 1/1024                   : REM EXACT POWER OF 2 FOR ACCURACY.
40 GRADIENT AT GUESS = (FN Y(GUESS+DX) - FN Y(GU))/DX: REM STANDARD FORMULA
50 GUESS=GUESS - FN Y(GUESS)/GRADIENT              ;REM NEWTON'S FORMULA
60 IF ABS(GUESS-G1)<1E-7 THEN PRINT: PRINT"SOLUTION:"; GUESS: END: REM PICK
   ECISION
70 PRINT GUESS;                  : REM WATCH SUCCESSIVE APPROXIMATIONS.
80 G1=GUESS                      : REM STORE THE CURRENT GUESS ...
90 GOTO 40                       : REM ... AND TRY AGAIN.
```

```
EXAMPLE RUN, USING SQUARE ROOT OF 2

   10 DEF FN Y(X) = X^2-2          : REM DEFINE FUNCTION TO BE SOLVED = 0.


GUESS   1
   1.49975598 1.41668019 1.41421656 1.41421356
SOLUTION: 1.41421356
```

Solving equations: inverse interpolation   Suppose you have an elaborate formula which calculates a single value from several inputs. Such a formula typically is easy to use one way round, but difficult to solve the other way. For example, a mortgage calculation might give a monthly repayment figure for any rate of interest: perhaps 200 at 5%, 220 at 10%, 250 at 15%. How can the interest rate corresponding to 215 be found? The obvious way is to converge on the value by guessing as well as possible, testing the guess, and improving on the guess until a satisfactory approximation is found. The process is less elegant, but easier to understand, than methods of Newton's type, which do exactly the same job. In our example, the interest rate is obviously in the range 5 to 10%; we could guess 7½%, try this value, and improve it in the light of the result. Without a calculator or computer, this is tedious; with a computer, it is not a problem. Section 16.4 has an example, involving an actuarial calculation on interest rates, which shows the procedure to use.

Integration: Simpson's rule   Integration is an algebraic process of aggregation, used to compute areas, volumes, rates of flow, strengths of fields and so on. Numerical integration techniques carry out the process without the need for intermediate analysis. Simpson's rule is representative of the type of method. It can be visualised as a means to determine the area between a curve and its x-axis. The result it produces is based on the supposition that the sample points which it uses are joined by a section of a parabola. The rule uses an odd number of values, which we can refer to as $y_1, y_2, y_3,$ ...,y_n corresponding to x-coordinates $x_1, x_2, x_3, ..., x_n$. The precision improves as more x-coordinates are taken, up to the point at which rounding errors caused by the large number of calculations accumulate. The formula for Simpson's rule is:-

$$\text{Estimate of integral} = (\text{x step-size})/3 \ * \ (y_1 + 4y_2 + 2y_3 + 4y_4 + ... + 4y_{n-1} + y_n)$$

Weights of 4 and 2 alternate, except for the end values. The BASIC program listed below repeats this calculation, with finer gradations in x, until estimates agree within a small margin. Its example run shows that the criterion is too severe for this case; all the estimates are nearly identical. This is because the original curve is a quadratic, which Simpson's rule solves exactly. The program assigns a step-size, S, in line 30; this is repeatedly halved as the calculations proceed. The odd terms (weight 2) are summed separately from the even terms (weight 4). From one step-size to the next we need only compute the alternate, odd, values, as the diagram shows, so that at each stage the even values become the total calculated so far. This cuts down processing time, because each value of the function is only calculated once.
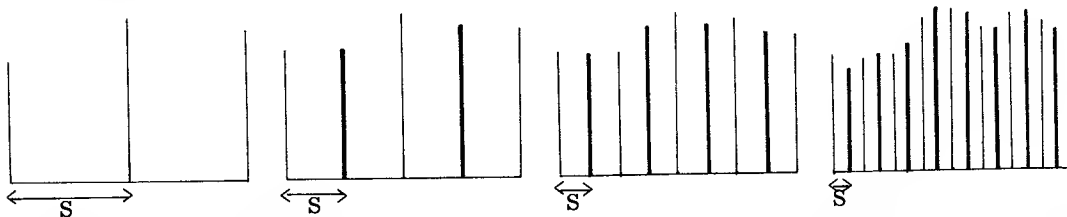

```
1 REM ***********************************************************************
2 REM * SIMPSON'S RULE.                                                     *
3 REM * INTEGRATES YOUR FUNCTION IN LINE 10.                                *
4 REM * SO=SUM OF ODD VALUES; SE=SUM OF EVEN VALUES; S=STEP SIZE.           *
5 REM *  NB: CHANGE LINE 90 IF LESS/MORE PRECISION IS REQUIRED.             *
6 REM *  ANOTHER VERSION OF SIMPSON, WITH GRADIENT END-CORRECTION, EXISTS   *
7 REM ***********************************************************************
8 REM
10 DEF FN Y(X) = [PI] * (100 - X*X): REM FUNCTION TO BE INTEGRATED
20 INPUT "INTEGRATE BETWEEN X1,X2"; X1,X2
30 S = (X2-X1)/2: SE = FNY(X1) + FNY(X2)
40 SE = SE + SO : SO = 0
50 FOR J = X1+S TO X2 STEP 2*S
60    SO = FN Y(J) + SO
70 NEXT
80 I=(2*SE + 4*SO - FN Y(X1) - FN Y(X2) ) * S/3
90 IF ABS (I -II) < 1E-8 THEN PRINT "INTEGRAL ="; I: END: REM CHOOSE
100 PRINT I:      REM WATCH APPROXIMATIONS
110 II = I:       REM STORE
120 S=S/2: IF S=0 THEN PRINT "DOESN'T CONVERGE": END
130 GOTO 40       : REM REPEAT FOR IMPROVED VALUE
```

THIS INTEGRAL GIVES THE VOLUME OF A SECTION OF A SPHERE:
EXAMPLE RUN

```
INTEGRATE BETWEEN X1,X2   8.745 10
     47.41092
     47.4109199
     47.4109198
     47.4109202
     47.4109201
INTEGRAL = 47.4109201
```

16.2 Statistics

Random numbers These are widely used in simulations of scientific and social phenomena, where overall behaviour of a system may be modelled as the outcome of many individually unpredictable events. The concept is also widely used to explain statistical distributions, using, at the introductory level, coins, dice, and cards. Before the widespread use of computers, 'random number tables' had to be prepared; computer power enabled pseudo-random numbers to be generated as required, which was more efficient than storing large tables. The usual method is to derive each pseudo-random number by a formula from the previous number. In this way a repeatable and testable series is generated. Recurrence relations are used: the number is multiplied by a large number, another large number is added, and the result forced into the correct range by taking the remainder after division by yet another number. There is plenty of scope for designing series which satisfy statistical tests for randomness. Such series always have a period of recurrence, but this is enormously long. Badly thought out series may have internal repetitive features of several types. Large computers store huge integers exactly and use these in their processing; Microsoft's random numbers work on similar, but not identical, lines. The differences are presumably an effect of the 31-bit storage method.

'RND' is explained in Chapter 5, and also in Chapter 15 in the ROM section. In view of the widespread confusion about this function (incidentally, it is implemented differently in different machines; don't assume that what follows will apply to non-CBM equipment) let me summarise its three main features:

(i) X=RND(0) or PRINT RND(0) are two expressions containing the function RND. RND behaves like all other arithmetic functions in BASIC, and can be assigned, printed, compared, calculated with, and so on. This should be straightforward to most people who have experimented with BASIC.

RND(0) is a 'truly' random number.* It is generated from four timers inside the VIA chip, which decrement every microsecond with the clock, so that any *single* use of this function generates a number between 0 and 1 which is non-repeatable in the usual sense. However, because the whole computer is controlled by a single set of timing pulses, *repeated* uses of this function are often non-random, unless (for example) an external event like a keypress influences the timing. For example, suppose a BASIC program happens to loop in exactly 65 milliseconds. Then RND(0) is exactly the same on each call! This is because the timer goes through a complete cycle in 65 thousand clock cycles. Try this with the 'random walk' program in Chapter 5. Note that BASIC 1 has a mistake in its ROM: evidently the VIA was moved at the last minute, and the addresses assumed in BASIC 1 were not updated from $9040-$904F.

(ii) RND (positive argument). The value of this function depends only on the stored random number, i.e. usually on the previous value returned by RND. RND(1) and RND(99999) in identical circumstances return equal values. A series of calls generates numbers from zero to one in a predictable sequence, each depending on the last. Their period of repetition is large.

(iii) RND(-ve). This function depends on the argument; it is always the same for a given argument. Thus, PRINT RND(-2);RND(-2);RND(-2) prints three identical numbers. The point of this is to enable programs containing RND to be tested: if RND(-1) say is entered at the start, rather than RND(0), all the subsequent RND(+ve) values will repeat when the program is re-run. This is helpful during debugging.

RND (+ve) works by multiplying by 11879546.4, adding $3.92767778 \times 10^{-8}$, interchanging two pairs of mantissas, setting the exponent to 128 (so the maximum value is .9999...), and normalising the result. RND(-ve) skips the two calculations, but is otherwise identical. These examples show how RND(-ve) appears in $88-8C, where RND is stored. (BASIC 1 uses $DA-DE). Note the small values produced by integers, a result of the interchange process:

Argument	Decimal value in location:				
	$88	**$89**	**$8A**	**$8B**	**$8C**
-.01	126	116	43	94	142
-.1	128	76	204	204	204
-.1234	128	35	35	185	252
-1	104	0	129	0	0
-2	104	0	130	0	0
-3.49	128	118	40	92	224
-5	104	32	131	0	0

*'Random' derives etymologically from the French 'Randir', to gallop.

The RAM workspace in which random numbers are stored is immediately after the GETCHR routine; on switchon or reset, both are together copied from ROM into RAM, so that there is a constant 'seed' value* when the machine is switched on. Section 16.3 includes examples of random numbers used in simulations.

Permutations and combinations   This statistical topic - sometimes called 'combinatorics' - uses the so-called 'frequency theory of probability' to construct theoretical models of actual distributions. For example, the possible of combinations of two dice throws can be listed by hand (1,1;1,2;1,3; ... ;6,6); there are 36 of them. If we postulate that each combination is equally likely to occur, we can construct a model of the distribution, which can be generalised into the binomial, Poisson and normal distributions. The details are too complex to summarise here. From the point of view of computation, it is worth knowing that factorials can be rapidly estimated:

A factorial[2] (2!=2*1=2; 3!=3*2*1=6; 4!=4*3*2*1=12; ...) is a rapidly-increasing function which turns up in many combinatorial calculations, since n! is the number of ways in which n different objects may be put into n pigeonholes. It is in fact a special function called the 'gamma function', but with integer arguments only. Stirling's formula approximates factorials:

$$n! \doteq (n/e)^n \sqrt{(2\pi n)} \; [1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3} + \frac{71}{2488320n^4} \cdots]$$

Rather than use this series directly, we can find $\log_e(n!)$ which gives results usable up to $n=10\uparrow36$ without overflow. The expression in square brackets can be approximated by $\exp(1/12n)$ without much loss of accuracy, giving

$$\log_e(n!) \doteq n \log_e n - n + \tfrac{1}{2}\log_e(2\pi n) + \frac{1}{12n}$$

$$\doteq (n+\tfrac{1}{2})(\log_e n - 1) + 1.4189 + \frac{1}{12n}$$

This short BASIC routine calculates the value of pCq, the number of ways in which q objects may be selected from p:

```
10 DEF FN LF(X) = (X+.5)*(LOG(X)-1) + 1.4189 + 1/(12*X)
20 INPUT P,Q
30 PRINT EXP(FN LF(P) - FN LF (Q) - FN LF (P-Q)):REM P!/(Q!(P-Q)!)
40 GOTO 20
```

The normal distribution   This well-known distribution, discovered by Gauss, is usually represented as a bell-shaped curve with two parameters, the mean and the standard deviation (m and s, say) with equation

$$n(x) = \exp \frac{(-(x-m)^2/2s^2)}{s\sqrt{(2\pi)}}$$

When the mean is zero and standard deviation one, the expression simplifies to

$$n(x) = \frac{\exp(-\tfrac{1}{2}x^2)}{\sqrt{(2\pi)}}$$

This distribution applies to measures (height, weight, length, etc.) in which the final result is influenced by a large number of individually small influences. It is not a very easy function to deal with; sometimes approximations are easier, such as this suggestion, where $-3 \leq x \leq +3$:

$$n(x) = .451 \; (1-\frac{x^2}{9})^5$$

Normal distributions can be simulated in a number of ways. The BASIC routine which follows uses an algorithm by Knuth[3] to generate a series of 'random' values with specified mean and standard deviation. The example simulates IQs as measured by one type of pencil-and-paper test (others have different standard deviations, so that their results vary more).

---

*The seed in fact cannot be assumed to be <u>exactly</u> constant. There is a programming mistake in the expression which moves GETCHR and the seed value; the loop is one byte too short, so the final byte of the seed in not actually transferred. So the seed's value depends on the contents of $8C ($DE in BASIC 1). The possible range of values is about .811635137 to .81165196.

[2] Note that 0! (not defined by the muliplicative series) is 1.

[3] Donald E. Knuth, 'The Art of Computer Programming'. 7 volumes.

KNUTH'S ALGORITHM FOR NORMAL DEVIATIONS

```
5 SIGMA=15: MEAN =100
10 V1 = 2*RND(1)-1
20 V2 = 2*RND(1)-1
30 S  = V1^2 + V2^2
40 IF S)= 1 THEN 10
49 REM DEVIATE IS:    SIGMA * V1 * SQR(-2*LOG(S) /S),
50 DEVIATE= SIGMA * V1 * SQR(-2*LOG(S) /S) + MEAN
55 PRINT DEVIATE
60 V1=V2: GOTO 20
READY.
```

```
104.691999
85.4889251
117.303867
75.6151702
94.0192871
81.4072698
61.5409044
100.408391
83.8830424
```

Probability distributions   The *binomial distribution* models the occurrence of independent events, giving the probability of the occurrence of n events on m occasions. If p is the probability of the event, and q (=1-p) is the probability of its non-occurrence, then m events occur on m occasions with probability $mCn \; p^n q^{m-n}$.
This is simply the nth term of the expansion of $(p+q)^m$. As we've seen, the expression mCn can be evaluated approximately using Stirling's formula, so that the entire expression for the probability is easily calculable, using the logarithms of mCn, p^n, and q^n. Example: the probability of throwing 55 sixes in 340 dice throws is

$$340C_{55} \; (1/6)^{55} \; (5/6)^{285}.$$

The *Poisson distribution* models events in the same way as the binomial distribution. It is a limiting form of the binomial, as the probability of an event becomes very small while the corresponding exposed-to-risk is large. A typical example is the number of printing errors on a page. The distribution is a function only of the mean number of events; the formula is

$$p(n) = \frac{e^{-m} m^n}{n!}$$

As an example, suppose there are on average 2 errors per page. (These events are   to be independent; this distribution won't model road accidents, where there is an obvious grouping effect). What is the probability of a page having four errors? The formula gives $\exp(-2)*2^4/4! = .09$.

The *Normal distribution* is important because (the result is derived from the 'Central limit theorem') samples taken from any other finite distribution are themselves normally distributed. Consequently many standard statistical tests and methods embody results which are true for the normal distribution. The t-test uses observed values to estimate ranges of values of the parent population; the chi-squared test estimates the squared normlised deviates from the mean (i.e. $(\frac{y-m}{s})^2$); and analysis of variance techniques

('ANOVA') attempt to sort out the separate influences of various factors - e.g. type of soil and type of fertiliser in crop-yield experiments - assuming a linear model. All these methods lend themselves to sausage-machine applications, notably amongst students in the US, where the computer packages to run them are freely available. Not many users of these facilities understand the statistical theories underlying the methods.*

---

*In 1981 errors were found in statistical packages in general commercial and educational use. The packages date (in part) from the mid nineteen sixties.

Non-parametric tests are relatively rough and ready, intended to provide guides where accurate measurement and calculation is too time-consuming or intrinsically difficult, in some quality control work, for example, and when dealing with subjective estimates or orderings. The 'sign test' is an instance: in say twenty consecutive fluctuating readings or measurements, about half can be expected to be above average, or to have a positive sign change from one to the next. A suitable warning-value can be decided by estimating the probabilities of 0,1,2,... variations. The 'cumulative sum' method which tests whether a mean value is correctly adhered to illustrates the same type of approach.

16.3 Simulation

Computer simulations have been attempted in a wide range of specialisations, with results of variable value. Weather forecasting relies on a vast amount of data, processed in vast machines. The earth's surface, or perhaps a hemisphere or other division, is notionally subdivided into small units, and a mathematical model employed to project present wind speeds and pressures at discrete altitudes within each unit. The results seem to be reasonably good, but not completely successful. More speculatively, mathematical models of economies have been constructed (by economists with the appropriate temperament) and are sometimes used to provide predictions. The results are not encouraging. Still more speculatively, 'World models' have been constructed, notably by J.Forrester in the early 1970s. These rely on hypothetical connections between food supply, population, resources, pollution, and so on, so that each variable in one time-period can be estimated, then each variable in the next, and so on. (It is assumed that 'pollution' can be measured as a single number). Less general models have been constructed by companies, particularly large ones, trying to quantify the results to be expected if sales suddenly increase, or interest rates fluctuate, or raw materials change in price, or some of the other vast number of factors influencing company performance come into play. 'Operations research', as it was called in the 1960s, dealt with some of these problems; techniques like 'linear programming' and the various types of numerical and dynamic programming algorithms date from this time. All these techniques, except those involving heavy number-crunching, can be run on microcomputers, provided the user is prepared to wait, and provided also that large scale data storage is supplied where necessary. A good example of a practical system is VisiCalc, a software package developed for microcomputers on a larger machine, which provides for row-and-column calculations, and is in effect a high-level language to input titles and mathematical formulas.

     It is extremely difficult to judge what proportion of computer simulation effort is simply a beguiling intellectual game. As with any model-building activity - notably in the fields of politics, economics and religion - results deduced from a model by those with an intellectual vested interest tend to be trusted to an excessive extent; results which run counter to common sense are, if anything, believed even more fervently.* However, microcomputers are perhaps less likely to lead their devotees into absurd errors than huge machines. We'll look at five examples of simple simulation; these are too small to approach any sort of sophistication, but show the type of thing involved in mathematical model-building. The first is a randomising routine, used for such purposes as simulating a card-shuffle. The next simulates words by selecting letters of the alphabet with their correct frequency. The third solves a well-known 'paradox' concerning birthdays; the fourth is a 'Monte Carlo' simulation of queue formation ('line' in the U.S.!); and the last embodies a simple biological theory of population

---

*Jay Forrester's books (published by M.I.T.) include 'World Dynamics' (1971) and a lesser-known work modelling towns, which assumes a basis of U.S. suburb style real-estate. 'Operations Research' by Ackoff and Sasieni (Wiley) covers typical O.R. methods and solutions. 'Newer Uses of Mathematics' (ed. J.Lighthill, Penguin in U.K.) and 'Mathematical Modelling' (Andrews & McLone, Butterworths) offer a survey of methods and 16 examples of modelling, respectively, which are interesting to the mathematical reader, although not particularly relevant to computers. As is true of most mathemat-ical works, little attempt is made to determine how far the constructs can be expected to apply in real life. There are many books on econometrics and related subjects; McGraw-Hill print a number of them. An out-of-print book by Andrew Wilson ('The Bomb and the Computer', Barrie & Rockliff, 1968) surveys military games from the 18th cent-ury to the present, claiming that many German errors in World War I were the result of inappropriate Prussian wargaming; and drawing similar conclusions about computer war-gaming in the Pentagon.

growth and decline of predator and host species.

(i) <u>Random Shuffling</u>. The algorithm used here is again the work of Knuth and simply selects one item at a time, reserving it in an early part of its array, so the remaining items have an equal chance of selection:

```
1 REM    THE ARRAY HOLDING THE VARIABLES IS PTRS(N), WHERE N IS THE DIMENSION.
2 REM    ('PTRS' STANDS FOR 'POINTERS', BUT THIS ISN'T A VALID NAME).
3 REM    THE ITEMS ARE PTRS(1) TO PTRS(N), I.E. NOT PTRS(0) WHICH IS SPARE.
4 REM    AFTER CALLING THE ROUTINE, PTRS(1) TO PTRS(M) ARE 'RANDOM'.
5 REM    TO SHUFFLE THE WHOLE ARRAY, PUT N-1 IN PLACE OF M.
10 FOR J = 1 TO M
20 J%      = J + RND(1)*(N-J+1)   :REM PICK RANDOM ELEMENT FROM J+1 TO THE END
30 TEMP    = PTR(J)               :REM AND SWAP IT WITH THE JTH
40 PTR(J)  = PTR(J%)
50 PTR(J%) = TEMP
60 NEXT
```

(ii) <u>Word generator</u>. The word 'Qume' was selected from a computer-generated list of 'meaningless words'. The following program may help you to do the same! The data statements are approximate frequencies of the alphabetic characters; they should add to 1. (They are guesses only). Line 130 chooses A-Z and space as the options; the list can include punctuation if required. Line 200 builds an array of cumulative frequencies (.06,.1,.14,.18,.27,... here) which line 310 compares with random number R from 0-1. Thus, R=.18 to R=.27 causes line 320 to print E, and in general the letters occur with the correct, or at least the specified, frequencies. The result can be made more lifelike by incorporating a 'stochastic' technique, i.e. taking account of previous letter(s) so that q always precedes u, for example, or capitals only follow full stop and space.

QUASI-WORDS

```
100 ALPHABET$="ABCUEFGHIJKLMNOPQRSTUVWXYZ ,."
110 DATA .06,.04,.04,.04,.09,.02,.03,.02,.05,.01,.005,.03,.03,.05:REM A-N
120 DATA .05,.02,.005,.03,.05,.07,.02,.01,.01,.005,.01,.005,.2:REM O-SP
130 N=27
200 DIM P(N): FOR J=1 TO N: READ P(J): P(J)=P(J)+P(J-1): NEXT
300 R=RND(1)
310 FOR J=1 TO N: IF R)P(J) THEN NEXT
320 PRINT MID$(AL$,J,1);
330 GOTO 300
```

(iii) <u>Birthdays</u>. Assuming birthdays are evenly spread throughout the year, how large a group of people must be selected to ensure an even chance that none of the group shares their date of birth? (I.e. ignoring the year of birth). This BASIC program prints a table of results. Line 130 calculates the probability p for n people that at least one pair have the same birthday, using the fact that if, say, 10 people have been chosen already, the 11th has a 355/365 chance of also being different.

```
100 PRINT "NUMBER    PROBABILITY     ALL DATES
104 PRINT "  OF      THAT SOME DATES DIFFERENT
108 PRINT "PEOPLE    ARE EQUAL:      ONLY ONCE IN:
110 FOR N = 5 TO 55 STEP 5
120 P=1: REM PROBABILITY IS 1 AT START, BUT DECREASES WITH EVERY PERSON.
130 FOR I = 1 TO N-1: P = P * (365-I)/365: NEXT
140 PRINT N"      "1-P"       "1/(P)
150 NEXT
```

(iv) <u>Monte-Carlo Queuing Simulation</u>. A 'Monte-Carlo' simulation generates random figures and puts them into a model. Roulette results can be simulated like this; a system can be tried out in this way, and can be expected to produce similar results in practice if the wheel is random, and the 'random numbers' are random, and the experiment is continued into 'the long run'. The example program (next page) models a situation where customers are served at several counters. Three variables are input at the start: the average time between customers, who are assumed to arrive evenly in the time period considered; the average time to serve a customer; and the number of

counters. All counters are presumed to be equivalent. Obviously, if the rate of service is too slow on average the queue length (which I've taken here to be the number of people waiting in line) will increase without limit. But even with adequate service there will be occasional queues. The program models consecutive intervals of time. For instance, suppose the interval taken is one minute, and the average time between customers is input as 5. The model assumes there is a one-fifth chance, as it enters its new interval, that a customer should enter. (Line 110). Line 160 models the chance of a customer leaving from any one of the counters in use. The program loops indefinitely while keeping running totals of the time period, the number of customers processed, and the number of people waiting (i.e. not being served) at any moment. This output shows the appearance on the screen:-

And this shows the result on stopping the program and entering GOTO 500 to see the summary:-

LENGTH	NUMBER
0	151
1	21
2	4
3	7
4	17
5	0
6	0
7	0
8	0
9	0
10	0

```
CUSTOMER OUT
 192 CUSTS: 9 QUEUE: 6 TOT CUST: 39
 193 CUSTS: 9 QUEUE: 6 TOT CUST: 39
 194 CUSTS: 9 QUEUE: 6 TOT CUST: 39
CUSTOMER OUT
 195 CUSTS: 8 QUEUE: 5 TOT CUST: 39
CUSTOMER IN
 196 CUSTS: 9 QUEUE: 6 TOT CUST: 40
CUSTOMER IN
CUSTOMER OUT
 197 CUSTS: 9 QUEUE: 6 TOT CUST: 41
 198 CUSTS: 9 QUEUE: 6 TOT CUST: 41
```

QUEUING THEORY SIMULATION ASSUMIMG RANDOM ARRIVALS:

```
0 DIM W%(100): REM HOLDS DISTRIBUTION OF LENGTHS OF LINE/ QUEUE
10 PRINT "[CLR][RVS] QUEUEING SIMULATION [DOWN][DOWN]
20 INPUT "AVERAGE TIME BETWEEN CUSTOMERS";C
30 INPUT "        AVERAGE TIME TO SERVE";S
40 INPUT "             NUMBER OF COUNTERS";N
50 PRINT "[DOWN]TYPICAL NUMBERS WAITING:"
100 REM ** IN ONE TIME INTERVAL: ***
110 IF RND(1)<1/C THEN P=P+1: TP=TP+1: PRINT "CUSTOMER IN"
120 IF P=0 GOTO 180
130 X=P
140 IF N<X THEN X=N: REM X IS SMALLER OF NO. OF PEOPLE, NO. OF COUNTERS
150 FOR J = 1 TO X
160 IF P>0 THEN IF RND(1)<1/S THEN P=P-1: PRINT "CUSTOMER OUT"
170 NEXT
180 Q=P-N: IF Q<0 THEN Q=0
190 W%(Q) = W%(Q) + 1
200 PRINT T; "CUSTS:"; P; "QUEUE:"; Q; "TOT CUST:"; TP
210 T = T + 1
220 GOTO 100
500 REM ** GOTO 500 PRINTS DISTRIBUTION OF QUEUES WHEN SIMULATION STOPPED
510 PRINT"LENGTH  NUMBER
520 FOR J=0 TO 20: PRINTJ; TAB(8); W%(J): NEXT
```

(v) Host-parasite population simulation. Simplifying somewhat, we can assume that the host population in a given time period increases by a natural rate of increase, which is reduced in proportion to the predator population. And we assume that the parasites die at some natural rate, unless there are hosts. This program displays the results following from the model:

```
100 INPUT "STARTING POPULATIONS OF HOST & PARASITE"; H,P
110 INPUT "RATE OF HOST INCREASE/ PARASITE DECREASE"; RH,RP
120 INPUT "EFFECT OF PARASITE ON HOST AND OF HOST ON PARASITE"; C1,C2
130 PRINT H,P
140 DH = (RH - C1*P)*H: DP = (-RP + C2*H)*P: REM LOTKA-VOLTERRA EQUATIONS
150 H = INT(H + DH): P = INT(P + DP): IF H<0 OR P<0 THEN END
160 GOTO 130
```

16.4 Accounting and actuarial programs

The type of programming in this subsection is concerned not so much with detailed record-keeping, but with techniques for solving well-defined problems of an arithmetic nature. The first example is an illustration of an accounting problem, namely to deduce pre-tax income from post-tax income, allowances, and reliefs. The second is an inverse interpolation example, used to solve an actuarial problem involving compound interest.

(i) <u>Tax Gross</u>. Five variables are used by the program, in addition to parameters which are set within the program. Lines 10-70 holds the number of tax bands, the step sizes of the bands (there are eight in the example) and the income tax rates applying to the bands. The figures given are not current. An example of the output, from a CBM printer, is shown. Lines 420-462 do the actual printing; the parameters A,C,K,S, and T are those input; X (with some rounding and formatting) is the gross amount. The point about denominators and numerators is to provide a correction where part years apply, where time has been spent abroad. Lines 200-310 perform all the calculations; TE is a test variable which is checked within the loop in line 250 (and in line 205 for low values).

```
10 DATA 8,0,750,9250,2000,3000,5000,5000,1E20
20 DATA0,.25,.3,.4,.45,.5,.55,.6
30 READNU:DIMBN(NU):DIMRT(NU)
40 FORI=1TONU
50 READBN(I):NEXT
60 FORI=1TONU
70 READRT(I):NEXT
170 REM CONTINUE
200 TE=S*(A-K)-T*C
205 IFTE<=0THENX=A:TA=0:GOTO320
210 BT=0
220 FORI=1TONU-1
230 TE=TE-BN(I)*(T-S*RT(I))
240 BT=BT+BN(I)
250 IFTE<=BN(I+1)*(T-S*RT(I+1))THENTA=RT(I+1):I=NU-1
270 NEXT
300 X=TE/(T-S*TA)
310 X=(X+BT+C)*T/S
320 REM END OF CALCS
420 PRINT"          "NAME$
422 PRINT
425 PRINT          "                    SALARY:"A
430 PRINT     " ALLOWANCES AND DEDUCTIONS:"C
433 PRINT"                      RELIEFS:"K
435 PRINT"          DENOMINATING FACTOR:"S
437 PRINT "          NUMERATING FACTOR:"T
450 PRINT
455 X=X* 100:GOSUB1000:X=X/100
458 Z1=1:GOSUB2000
460 PRINT"          | GROSS SALARY ="X" |"
462 Z1=2:GOSUB2000
```

```
              JONES      1978/79


                          SALARY: 10000
      ALLOWANCES AND DEDUCTIONS: 123
                        RELIEFS: 12
            DENOMINATING FACTOR: 19
             NUMERATING FACTOR: 20


      ┌────────────────────────────────┐
      | GROSS SALARY = 16896.86 |
      └────────────────────────────────┘
```

(ii) Compound interest calculations. The program on the next page has a fully documented format which explains the workings of inverse interpolation. Its expression is a simple one; line 500 has a present value function corresponding to the value, in money terms, of a sum of 100 payable in a year's time at interest rate 1%. At five per cent the formula gives 100/1.05 = 95.238. Lines 1000 to 2000 repeatedly calculate values until either a good estimate is found – its accuracy controlled by the comparison in line 1020 – or until it becomes clear that there is no solution. A typical output is this:

```
OFFER VALUE? 95
INTEREST RATE IS 5.26321411
     WHEN PRESENT VALUE = 94.9999493

AT INTEREST RATE = 5.263
        VALUE IS  95.0001425
AND AT INTEREST RATE = 5.264
        VALUE IS  94.9992401
```

from which it is clear that a present value of 95 corresponds to 5.263%. This figure is easy to check by solving the equation. The partial example which follows is less simple to check, involving such complications as repayments several times during the year, the deduction of income tax, and payments of capital gains tax on the difference between the redemption amount and the offer price. The formulas are standard ones, recognisable to people who use them; I'm not *certain* that all the details are correct. Note though that the input statements and function definitions can be inserted into the inverse interpolation program, although, because of the complexity of the expression, a subroutine rather than a function definition may have to be used in lines 1000-1040 when each value is being computed.

```
100 REM
101 REM ###################################################################
102 REM #                                                                 #
103 REM #      VALUATION OF A FIXED-INTEREST REDEEMABLE SECURITY          #
104 REM #                                                                 #
105 REM ###################################################################
106 REM # ASSUMES:     REDEEMABLE AMOUNT (C)                              #
107 REM #              AFTER A NUMBER OF YEARS (N)                        #
108 REM #              AT TRUE (NOT NOMINAL) RATE OF INTEREST, G          #
109 REM #              PAYABLE YEARLY P TIMES                             #
110 REM #              PURCHASED 1/M OF A YEAR AFTER LAST PAYMENT         #
111 REM #              WITH INCOME TAX AT RATE T,                         #
112 REM #              AND CAPITAL GAINS TAX AT RATE T1                   #
113 REM #              VALUATION RATE OF INTEREST I.                      #
114 REM ###################################################################
115 REM
120 DEF FN V(N) = (1+I)^(-N):  REM CALCULATE V TO THE N.
130 DEF FN S1(P) = (((1+I)^N)-1)/(P*((1+I)^(1/P)-1)): REM ACCUM. VALUE PAID PTHLY
140 DEF FN A(N) = (1 - FN V(N) )/I: REM VALUE OF ANNUITY OVER N YEARS
150 INPUT "               REDEEMABLE AMOUNT";C
160 INPUT "               AFTER HOW MANY YEARS";N
170 INPUT " ACTUAL ANNUAL INTEREST PAYMENT";G :G=G/C    :  REM G=ACTUAL RATE
180 INPUT "HOW MANY TIMES PAYABLE PER YEAR";P
190 INPUT "NO. OF DAYS SINCE LAST DIVIDEND";M : M=M/365:  REM M=FRACTION OF YR
200 INPUT "              RATE OF INCOME TAX";T : T=T/100:  REM DECIMAL
210 INPUT "     RATE OF CAPITAL GAINS TAX";T1: T1=T1/100:REM DECIMAL
220 INPUT "     VALUATION RATE OF INTEREST";I :I=I/100:   REM DECIMAL
300 IF P=1 THEN PV = (1+I)^M * (C*FNV(N) + G*(1-T)*C*FNA(N) - T1*(C-A)*FNV(N))
310 IFP<>1THEN PV=(1+I)^M * (C*FNV(N) + G*(1-T)*FNS1(P)*C*FNA(N) - T1*(C-A)*FNV(N))
```

I have insufficient space to consider problems of life assurance, life table calculations, etc. in detail. A major difficulty is the need to store large amounts of tabular data; some life tables have been calculated in the form of smoothed formulas, usually three or so curves which, between them, cover the entire age span. If such a formula is considered accurate, naturally there is a great saving in storage space (though not in computing time, probably) in calculating each value as it is needed. I have no information on the extent of small computer use in the insurance world. Obviously the companies involved use mainframes for storage of their bulk data; perhaps small machines

have their place too.

```
0 REM
1 REM###############################################################################
2 REM##  INVERSE INTERPOLATION DEMONSTRATION USING A PRESENT VALUE FUNCTION.  ##
3 REM##                                                                       ##
4 REM## METHOD: REPEATED CALCULATIONS, USING A BINARY CHOP, CONVERGE TO THE   ##
5 REM##         CORRECT VALUE, AND WHEN WITHIN AN ARBITRARILY SMALL AMOUNT OF ##
6 REM##         THIS VALUE, REPORT THE RATE OF INTEREST FOUND.                ##
7 REM##                                                                        #
8 REM##            LINES 500 - 1040 PERFORM THE MAIN CALCULATIONS.            ##
9 REM###############################################################################
10 REM
500 DEF FN PV(I) = 100 / (1+ I/100) : REM   VERY SIMPLE PRESENT VALUE FUNCTION
600 INPUT "OFFER VALUE"; V
700 IL = -10: IH = 50: REM CHOOSE -10% TO 50% AS LARGEST REASONABLE RANGE TO TRY
980 REM
981 REM###############################################################################
982 REM# INVERSE INTERPOLATION CALCULATION BEGINS.                                  #
983 REM###############################################################################
984 REM# NOTE:1) IL AND IH ARE LOW,HIGH ESTIMATES AT EACH STAGE OF ITERATION,#
985 REM#          STARTING AT -10% AND 50% TO BE SURE TO CATCH MOST VALUES.     #
986 REM#       2) AS INTEREST RATE 'I' INCREASES, VALUE DECREASES; THAT'S WHY  #
987 REM#          THE TESTS IN LINES 1030-1040 CHANGE LIMITS THE WAY THEY DO.  #
988 REM#       3) INTEREST RATES APPEAR AS NORMAL         EG 12%, AND NOT 0.12#
989 REM#       4) LIMITS IN 700,ACCURACYIN 1020,DEC.PTS.IN 3010,ARE ALTERABLE.#
990 REM###############################################################################
991 REM
1000 IF IL = IH THEN 2000: REM OUT OF RANGE OF START VALUES - CAN'T BE FOUND
1010 BEST = (IL + IH) /  2 : REM AVGE OF LOW AND HIGH RATES IS BEST ESTIMATE
1020 IF ABS (FN PV(BEST) - V) < 1 E-4  GOTO 3000: REM CLOSE APPROXIMATION FOUND
1030 IF FN PV(BEST) < V THEN IH = BEST : GOTO 1000: REM 'BEST' TOO HIGH
1040 IF FN PV(BEST) > V THEN IL = BEST : GOTO 1000: REM 'BEST' TOO LOW
1990 REM
1991 REM###############################################################################
1992 REM# REPORT HERE IF THE INTEREST RATE IS AN EXTREME OUT-OF-RANGE VALUE #
1993 REM###############################################################################
1994 REM
2000 REM NOT FOUND - RATE EITHER LESS THAN -10% OR GREATER THAN 50% .
2010 PRINT "NOT FOUND - ";
2020 IF BEST = -10 THEN PRINT "RATE IS LESS THAN -10%"
2030 IF BEST = 50 THEN PRINT "RATE EXCEEDS 50%"
2040 PRINT:PRINT
2100 GOTO 600: REM CONTINUE - INPUT NEXT VALUE
2990 REM
2991 REM###############################################################################
2992 REM#         REPORT HERE WHEN VALID INTEREST RATE HAS BEEN FOUND.           #
2993 REM#                                                                        #
2994 REM# UNROUNDED RATE AND TWO VALUES (FOR 3 D.P. RATES) ARE PRINTED OUT.###
2995 REM###############################################################################
2996 REM
3000 PRINT "INTEREST RATE IS"; BEST
3010 PRINT"    WHEN PRESENT VALUE ="; FN PV(BEST)
3020 PRINT
3030 I = INT (1000 * BEST) / 1000 : REM THIS TRUNCATES THE VALUE TO 3 DEC. PL.
3040 PRINT "AT INTEREST RATE =";I
3050 PV = FN PV(I)
3060 PRINT "        VALUE IS "; PV; "[LEFT]."
3070 PRINT "AND AT INTEREST RATE ="; I+.001 : REM NEXT VALUE AT 3 DEC. PL.
3080 PV = FN PV(I + .001)
3090 PRINT "        VALUE IS "; PV; "[LEFT]."
3100 PRINT:PRINT
3110 GOTO 600: REM CONTINUE - INPUT NEXT VALUE
READY.
```

16.5 Trigonometry

CBM machines all have sine, cosine, tangent and arctangent as standard. These were presumably taken over from FORTRAN by Microsoft, rather than specially written. The three angle functions all call the sine routine; tangent is calculated by dividing sine by cosine, and is therefore likely to have larger rounding errors, and be slower. The arctangent is quite useful in some analytical problems, since its range covers the whole real number spectrum from minus to plus infinity. This is a result of the fact that the tangent of an angle is the ratio of two unrelated sides; their ratio therefore can take any value, unlike sine and cosine. Trigonometrical functions typically have uses in engineering, surveying, perspective and drawing calculations. Each function relates two sides of a right-angled triangle to an angle; see Chapter 5. Since there are 3 sides, 6 possible ratios exist, which are, using the shorthand H=hypotenuse, A=side adjacent to the relevant angle, and O=opposite side,

$$\text{Sine}=\frac{O}{H} \quad \text{Cosine}=\frac{A}{H} \quad \text{Tangent}=\frac{O}{A} \quad \text{Cosecant}=\frac{H}{O} \quad \text{Secant}=\frac{H}{A} \quad \text{Cotangent}=\frac{A}{O} \; .$$

Most of this will be known by readers of this subsection already. It is worth recalling two simply-drawn examples which provide practice in these functions: the first is an equilateral triangle with sides of length two units each, and the second a right-angled triangle with two other angles of 45° and sides 1,1, and 1.414 (=$\sqrt{2}$). Then relationships involving angles of 30°,45°, and 60° are simple to check, e.g. $\tan(45°)=1$, $\cos(60°)=\frac{1}{2}$.

All of these functions are defined so that lines may be considered negative in length; this convention supplies the familiar repetitive sine curve with theoretical foundations. Some values, 0°,90°,180° and so on may give trouble in evaluation if division by zero becomes a possibility. We shall see how to program around this, and make the calculations crashproof. A potentially confusing point about notation is also worth clarifying: the inverse sine function, which gives the angle corresponding to a sine value, and which is called the 'arcsine', is written as $\sin^{-1}x$. This is easily confused with the reciprocal of sin x, which is $(\sin x)^{-1}$.

Crashproofing trig functions   There are three methods:

(i) Request the user to avoid values known to crash; typically 90° or 0° may have to be avoided.

(ii) Test the values input and disallow those which crash, printing either an error message or a known correct value instead. Often an expression can be simplified in some way: for example sin x/sin 2x will crash if x=0, but the expression can be shown to equal 1/2cos x, which is troublefree when x=0.

(iii) A more thorough implementation of (ii) modifies the function itself so that extreme values are replaced by values almost identical, and yielding the correct solutions, but protected against crashing. SIN(X + (X=0)*1E-9) in place of SIN(X) substitutes SIN(1E-9) for SIN(0) if this function appears in the lower half of a fraction, for example. A better example is provided by this crashproofed expression for arccos, which uses the relation   ARCCOS(X) = ARCTAN(X/SQR(1-X²)) :

```
100 DEF FN AC(X) = ATN(X/((X=1)*1E-9 + SQR(1-X*X))) * (-1)↑(N+1) + [PI]*N+[PI]/2
110 DEF FN DC(X) = FN AC(X) * 180 / [PI] :REM ARCCOSINE OF X IN DEGREES
200 FOR N = 0 TO 10: PRINT FN DC(0): NEXT
```

The example prints 90,270,450,...1710.,1890.

Trigonometric equations   A typical textbook general equation for solution is:

```
a sin x + b cos x = c
```

By dividing through with sqr(a²+b²) and using an expansion of cos(p+q), we can establish the general solution as

```
x = arccos (c/sqr(a²+b²)) - atn(-b/a).
```

```
10 DEF FN AS(X) = (-1)↑N * ATN (X/SQR(1-X*X)) + [PI]*N :REM ARCSINE
20 DEF FN AC(X) = [PI]/2 - FN AS(X):REM ARCCOS DEFINED IN TERMS OF ARCSIN
100 INPUT "a,b,c in a sin x + b cos x = c"; A,B,C
110 FOR N = -5 TO 5                :REM PRINT A SPECIMEN RANGE OF SOLUTIONS
120 THETA = FN AC(C/SQR(A*A+B*B)) - ATN(-B/A)
130 THETA = THETA*180/[PI]: PRINT THETA "DEGREES": NEXT
```

Trigonometrical functions

```
0 PI=[PI]
9 REM
10 REM *** FUNCTIONS OF ANGLES IN DEGREE ***
11 REM
20 DEF FN SI(X) = SIN(X*PI/180) : REM CALCULATE SINE OF ANGLE IN DEGREES
30 DEF FN CO(X) = COS(X*PI/180) : REM CALCULATE COSINE OF ANGLE IN DEGREES
40 DEF FN TA(X) = TAN(X*PI/180) : REM CALCULATE TANGENT OF ANGLE IN DEGREES
99 REM
100 REM *** INVERSE TRIGONOMETRICAL FUNCTIONS (RADIANS) ***
101 REM
110 DEF FN AS(X) = (-1)^N * (ATN (X/SQR(1-X*X)) ) + PI*N :      : REM ARCSIN
120 DEF FN AC(X) = PI/2-(-1)^N * (ATN (X/SQR(1-X*X)) ) + PI*N : REM ARCCOS
130 DEF FN AT(X) = ATN(X) + PI*N   : REM GENERAL ARCTAN
199 REM
200 REM *** INVERSE TRIGONOMETRICAL FUNCTIONS IN DEGREES ***
201 REM
210 DEF FN DS(X) = FN AS(X)*180/PI : REM ARCSINE OF X IN DEGREES
220 DEF FN DC(X) = FN AC(X)*180/PI : REM ARCCOSINE OF X IN DEGREES
230 DEF FN DT(X) = FN AT(X)*180/PI : REM ARCTANGENT OF X IN DEGREES
299 REM
300 REM *** FURTHER INVERSE TRIGONOMETRY FUNCTIONS ***
301 REM
310 REM ARC SEC(X)   = ARC COS(1/X)
320 REM ARC COSEC(X) = ARC SIN(1/X)
330 REM ARC COT(X)   = ARC TAN(1/X)
```

16.6 Arrays and Matrices.

Definitions and rules of manipulation   A 'matrix' in the mathematical sense is a two-dimensional array of numbers. In BASIC these can be conveniently stored as A(R,C) say, where R and C represent the dimensions of the array and mnemonically suggest the order rows then columns, which is the usual convention. Matrices may be manipulated mathematically in several conventional ways. Some BASICs (e.g. IBM's) have their own MAT statements to facilitate this, but Microsoft BASICs normally don't, presumably because of the comparatively small demand. There are difficulties in implementing commands of this sort in any case, because of the need to reduce rounding errors when dealing with the very largest matrices, and because of the memory requirements.

The reasons why matrices can be useful aren't especially easy to understand. The best point of entry is probably to look at simultaneous equations:

$$a + 10b + 100c = .03$$
$$a + 50b + 2500c = .29$$
$$a + 100b + 10000c = .98$$

Here we have three equations connecting three unknowns, a,b, and c. Small sets of equations like this one can be solved quite quickly by comparing pairs of equations and eliminating unknowns. The interpretation of these equations, in concrete terms, is not too difficult; for example, a, b, and c may represent weights in grams, and the three combinations of one of the first type of item, ten of the second, and one hundred of the third and so on plus the respective weights may have been determined empirically. Solving the equations gives the weight of each type of item. Another example: suppose an egg + two pieces of bacon costs 75 units, while an egg + one piece of bacon costs 50. Assuming the costs are straghtforward calculations, these equations represent the situation:

$$pe + 2pb = 75$$
$$pe + pb = 50$$

And of course the two prices pe and pb are 25 units each. Matrices deal with situations of this sort by separating out the block of factors from the block of variables, and the rules of matrix addition, subtraction, and multiplication are made consistent with this scheme. Consequently, matrices are usable whenever calculations of this sort

occur: in economics, attempts are made to construct matrices to model flows of raw materials and made-up goods between industries; price x quantity calculations may be made;* predictions and deductions about migration, genetics, and so on may be made. The idea is always similar to that embodied in our simultaneous equations. We can represent the first of them in this way:

$$\begin{bmatrix} 1 & 10 & 100 \\ 1 & 50 & 2500 \\ 1 & 100 & 10000 \end{bmatrix} * \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} .03 \\ .29 \\ .98 \end{bmatrix}$$

Which may help to clarify the idea of matrix multiplication: the point is that the row(s) of the first matrix are multiplied by the column(s) of the second and added to give the corresponding elements in the third. For this reason multiplication is meaningless if in A(R,C) * B(R',C') the value C is unequal to R'. Division is not defined directly, but is implicit in the idea of multiplication: a square matrix holding only zeros, except for its top-left to bottom-right diagonal which holds ones, is called the 'identity matrix' and corresponds to 1; any two matrices which multiply to give this are called 'inverses' of each other, so each is something like the reciprocal of the other, as the name 'inverse' implies. As we shall see, matrix inversion is a standard operation required in calculations using matrices. Adding matrices is simpler: the matrices must match both in the number of rows and the number of columns. Each corresponding element is then added. Subtraction is similar.

Although column vectors (there are two in the màtrix equation above) are often used in matrix multiplication, the generalision implied in the comments on the identity matrix and inversion enables any matrices of form A(R,C) and B(R',C') to be multiplied, provided C=R'. For example, these 3*2 and 2*2 matrices give a 3*2 result:-

$$\begin{bmatrix} 1 & 0 \\ 4 & 5 \\ 2 & -2 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 14 & 4 \\ -2 & 2 \end{bmatrix}$$

Matrix inversion and simultaneous equations By way of preliminaries, let's first see how to input and output an entire matrix, without worrying too much about format:

```
   O INPUT "NUMBER OF ROWS, COLUMNS"; R,C: DIM A(R,C)
  10 FOR X = 1 TO R: FOR Y = 1 TO C
  20 INPUT A(X,Y)         :REM INPUT ONE ROW AT A TIME. READ A(X,Y) ALSO USABLE.
  30 NEXT Y,X

 500 FOR X = 1 TO R: FOR Y = 1 TO C
 510 PRINT A(X,Y);        :REM PRINT ONE ROW AT A TIME
 520 NEXT Y: PRINT: NEXT X   :REM NEW ROW ON A NEW LINE
```

Lets also see how to multiply matrices; this is necessary for several purposes, including the testing of programs:

```
 400 REM MULTIPLY A(R,C) BY B(R',C') GIVING R(R,C') AS RESULT
 410 FOR I = 1 TO C1   :REM RESULT'S COLUMNS
 420 FOR J = 1 TO R    :REM RESULT'S ROWS
 430 FOR K = 1 TO C    :REM COMMON COLUMNS AND ROWS
 440 R(J,I) = R(J,I) + A(J,K)*B(K,I) :REM SINGLE ELEMENT ADDS TOGETHER C PRODUCTS
 450 NEXT K,J,I
```

Note that a matrix multiplication where the second array is a column, as in the example at the top of this page, the outermost loop is redundant, and can be omitted when for example the solutions of simultaneous equations are checked by substitution into the original equations. If a row array is multiplied by a column array, the result is a 1 x 1 array; in this case only the innermost loop is required.

The two matrices (each 4 x 4) are inverses of each other; when multiplied, irrespective of the order of multiplication, the result, allowing for rounding errors, is the 4 x 4 identity, consisting of 1s in the leading diagonal and 0s elsewhere. We can see how the inverse may be used to solve simultaneous equations, by premultiplying

---

*It is important to have a clear idea of the space requirements and processing times to expect with matrices. Although they permit exhaustive calculations to be made, often (e.g. in economic models) most of the elements are zero, which seems rather wasteful. One of Gerry Weinberg's stories (in 'The Psychology of Computer Programming') on this subject recounts how an organisation's whole price structure, to be used in invoicing etc., was to be held in an array. It turned out to be too big for the machine.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 12 & 4 & 0 \\ -1 & -5 & -7 & 12 \\ 0 & 0 & 555 & 8 \end{bmatrix} \text{ and } \begin{bmatrix} .75277 & -.22847 & -.24723 & -.0054058 \\ .00029949 & .083408 & .00029949 & -.00059898 \\ -.00089847 & -.00022462 & -.00089847 & .0017969 \\ .062332 & .015583 & .062332 & .00033693 \end{bmatrix} \text{ are inverses.}$$

both sides of an equation involving arrays by the inverse of the (square) array which holds the coefficients of the equations; if **A** is the array, so for example

$$A * \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 10 \\ 1 \\ 10 \\ 0 \end{bmatrix} \, ,$$

then on premultiplying we get these results:

$$A^{-1} * A * \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = A^{-1} * \begin{bmatrix} 10 \\ 1 \\ 10 \\ 0 \end{bmatrix} \text{ so } \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = A^{-1} * \begin{bmatrix} 10 \\ 1 \\ 10 \\ 0 \end{bmatrix} \text{ and } \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = A^{-1} * \begin{bmatrix} 10 \\ 1 \\ 10 \\ 0 \end{bmatrix} \, .$$

So the solution of these equations:

$$\begin{aligned} a + 2b + 3c + 4d &= 10 \\ 12b + 4c &= 1 \\ -a - 5b - 7c + 12d &= 10 \\ 555c + 8d &= 0 \end{aligned}$$

can be found by premultiplying the column matrix by the 4x4 inverse at the top right of this page, giving a=4.827, b=.08940, c=-.01819 and d=1.262.

A great deal of work has gone into finding and improving methods of matrix inversion to maximise speed and accuracy of different types of matrix. There is no room here to begin to summarise this work. Instead, I shall develop a program using a mundane algorithm for inversion, which should be usable for quite a number of purposes.* It is not particularly elegant, but it does work. We may as well note at this point that not all square matrices can be inverted; those corresponding to simultaneous equations which won't solve, either through inconsistency or insufficiency of data, have no inverses. For example:

$$\begin{aligned} x + 2y &= 10 \\ x + y &= 10 \end{aligned} \quad \text{and} \quad \begin{aligned} x + y &= 10 \\ 2x + 2y &= 20 \end{aligned}$$

have non-invertible matrices. Some matrices are described as 'ill-conditioned', by which is meant that their inverses exist, but are sensitive to small changes in the original matrix, so the inverse tends to have large errors in its elements; the matrix

$$\begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 & \dots \\ 1/2 & 1/3 & 1/4 & 1/5 & \dots \\ 1/3 & 1/4 & 1/5 & 1/6 & \dots \\ \dots & \dots & \dots & \dots & \end{bmatrix}$$

provides an illustration.

The matrix inversion program (in BASIC; see next page) uses row operations to convert the original matrix M(R,R) into the identity matrix. Identical operations are carried out on the identity matrix. The result is that M becomes the identity matrix and I(R,R), the duplicate matrix originally holding the identity, is transformed into the inverse. M goes through these stages:

$$\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix} \begin{bmatrix} 1 & x & x \\ 1 & 1 & x \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & x & x \\ 0 & 1 & x \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The program has four parts, excluding the input routine and any printout and/or checking routines.

(i) Lines 100-260 constitute a large loop which carries out the first two stages in the diagram above. Lines 100-170 perform divisions to convert the leading diagonal into 1s, while

---

*Donald Alcock ('Illustrating BASIC') has a shorter program which uses Gaussian elimination. CBM and other Microsoft BASICs need to add 405 IF I+1 > N THEN GOTO 430 to this program, since Alcock has assumed standard Dartmouth BASIC loop handling.

COMPLETED MATRIX INVERSION PROGRAM

```
10 INPUT"DIM";N
11 DATA 1,2,3,4,0,12,4,0,-1,-5,-7,12,0,0,555,8
12 DIMM(N,N),I(N,N)
13 FORY=1TON:FORX=1TON:READ  M(X,Y):NEXTX,Y
14 FORY=1TON:FORX=1TON:PRINTM(X,Y);:NEXT:PRINT:NEXT
16 FORX=1TON:            I(X,X)=1:NEXT

100 FOR X = 1 TO N
110  FOR Y = X TO N
120     D = M(X,Y) ; IF D=0 OR D=1 GOTO 170
130 FOR K = X TO N
140        M(K,Y) = M(K,Y)/D
141 NEXT K
142 FOR K = 1 TO N
150        I(K,Y) = I(K,Y)/D
160     NEXT K
170   NEXT Y
180 IF X=N GOTO 270
190   FOR Y = X+1 TO N
200     IF M(X,Y)=0THEN 250
210 FORK=XTON
220 M(K,Y)=M(K,Y)-M(K,X)
221 NEXT K
222 FOR K = 1 TO N
230 I(K,Y)=I(K,Y)-I(K,X)
240 NEXTK
250 NEXT Y
260 NEXT X

270 FORX=1TON:IFM(X,X)=1THEN NEXT
271 IFX()N+1THENPRINT"NOT INV":END
272 FOR X=N TO 2 STEP -1
280   FOR Y=X-1 TO 1 STEP -1
290     D=M(X,Y)
300 FOR K = XTO N
310        M(K,Y)=M(K,Y) - M(K,X)*D
311 NEXT K
312 FOR K = 1 TO N
320        I(K,Y)=I(K,Y) - I(K,X)*D
330     NEXT K
340     NEXT Y
350     NEXT X
```

Depending on the structure of the original matrix, inversion takes something like 30 seconds for a 10 by 10 matrix and 4 minutes for a 20 by 20 matrix. A 32K machine can handle matrices of about 55 square, leaving no room for anything but the inversion program. With BASIC at 1 MHz this takes about three quarters of an hour.

The program can be tested by (i) Inverting a matrix with a known inverse; the identity matrix should invert to itself, with some rounding error; (ii) Reinverting a matrix, and checking that the result is close to the original; or (iii) Multiplying by the original and checking that the answer is similar to the identity matrix.

Applications of the technique to simultaneous equations include the calculation of coefficients of regression equations in statistics; many more variables can be handled than are practicable by manual calculation. Typically this may be useful when an algebraic curve is to be fitted to data collected by experiment. Another example is in the solution of 'over-determined' simultaneous equations, where there are many observations of empirical material involving only a few variables. Suppose you have details of output involving units of raw material which have been used in different combinations: say a sheet of timber has been made into 15 items of type A, 100 of type B, and 30 of type C; and results are available like this for 100 sheets. What is the best estimate of the amount used per type of item? Perhaps figures are available of the

weekly output from a group producing a mix of different size items of the same type. How can you estimate (for sensible pricing) the time taken on size 1, size 2, etc.? One method is to premultiply both sides of the matrix expression by the transpose of the matrix; the result provides the best least-squares estimates of the variables' values. This simple case illustrates the method:

$$
\begin{array}{lll}
\text{4 empirically} & 2x+y=8, & \text{Correspond to} \\
\text{obtained equations:} & x+y=7, & \text{this matrix} \\
& 2x+y=9, & \text{arrangement:} \\
& x+3y=6. &
\end{array}
\qquad
\begin{bmatrix} 2 & 1 \\ 1 & 1 \\ 2 & 1 \\ 1 & 3 \end{bmatrix}
\begin{bmatrix} x \\ y \end{bmatrix}
=
\begin{bmatrix} 8 \\ 7 \\ 9 \\ 6 \end{bmatrix}
$$

As it stands, this is insoluble; the least-squares solution, however, is obtained like this:

$$
\begin{bmatrix} 2 & 1 & 2 & 1 \\ 1 & 1 & 1 & 3 \end{bmatrix}
\begin{bmatrix} 2 & 1 \\ 1 & 1 \\ 2 & 1 \\ 1 & 3 \end{bmatrix}
\begin{bmatrix} x \\ y \end{bmatrix}
=
\begin{bmatrix} 2 & 1 & 2 & 1 \\ 1 & 1 & 1 & 3 \end{bmatrix}
\begin{bmatrix} 8 \\ 7 \\ 9 \\ 6 \end{bmatrix}
\quad \text{which} \atop \text{simplifies to:} \quad
\begin{bmatrix} 10 & 8 \\ 8 & 12 \end{bmatrix}
\begin{bmatrix} x \\ y \end{bmatrix}
=
\begin{bmatrix} 47 \\ 42 \end{bmatrix}
$$

and this matrix expression is solvable. Estimates of the error in the result can be calculated. This is, of course, only one of innumerable mathematical techniques which are available. It is a relatively easy one to program. If we set NE=number of equations and NV=number of variables, this routine calculates the square matrix S which results from the multiplication of the transpose of M by M itself:

```
1000 FOR C = 1 TO NV
1010 FOR R = 1 TO NV
1020 FOR K = 1 TO NE
1030 S(R,C) = S(R,C) + M(K,R)*M(R,K)
1040 NEXT K,R,C
```

16.7 Number Theory.

Number theory typically deals with large integers; these are available only with specially-written routines in Microsoft BASIC, and as the subject is not of wide interest, I shall illustrate a solution only of a small-scale problem, using BASIC to assist in the solution. The problem is: find a telephone number (of 3 digits followed by 4) which, if the first part is subtracted from the second, and the result squared, equals the original seven-digit number.

I.e. put x=abc, y=ghjk in form. Then we have:

$(y-x)^2 = 10000x + y$. There are about $999*9999$ combinations which *could* be tested. (The precise number depends on whether a number like 000 counts). This will take a long time; probably a few weeks on the computer. However, a little algebraic shuffling (solving a quadratic for y) produces the more severe restriction that

$40004x + 1$ must be a perfect square if the equation is true.

```
10 FOR X=0 TO 999: REM TEST WHOLE RANGE OF PREFIX NUMBERS
20 Y=40004*X + 1
30 IF ABS (SQR(Y) - INT(SQR(Y)) < .001 THEN PRINT X
40 NEXT: END
```

This program isn't too long in run-time. Line 30 tests for exactness of the square root; the range of $Y^{\frac{1}{2}}$ is about 200 to 6000, so the test should adequately screen out wrong values. The actual form of the test isn't really important; the point is to get some figures, which can easily be checked. In fact, only 120 prints on the screen, and the solution is 120 1216.

16.8 Curve fitting.

Least squares methods   The object of curve-fitting is to discover some fairly simple formula to represent empirical data; the point is to get the representation into a form which is easy to handle, rather than (say) as a graph or as a table. The techniques are well-known; related topics include regression and correlation, the formulas of which are also readily available in textbooks. The least-squares formula, to be briefly explained here, is a process which calculates coefficients of a linear expression which minimise the sum of squares of predicted and actual results. Usually some attempt is made to determine a likely form of a function, which is then fitted to the data and checked for significance. This process can be automated partially. There is a BASIC

program ('CURFIT') of Jim Butterfield's which fits data to several types of curve and reports the results.

The least-squares method starts with a linear function:

$$y = a + bx_1 + cx_2 + dx_3 + \ldots + \text{random error}$$

Where x_n can be for example x^2 or $\log(x)$ or some function of x (although this makes error estimation more complex). For example, functions of the following types can be represented like this, in some cases using transformations to convert some of the data items into more usable forms:

$y = a + bx$ straightforward linear relation,
$y = a + bx + cz$ linear relationship with 2 variables,
$y = b/x + a + cx + dx^2$ where $x_1 = 1/x_2$ and $x_3 = x_2^2$,
$y = kxyz^2$ where a logarithmic conversion is needed, which gives
$\log y = \log k + \log(xyz^2)$, or $y' = a + x'$, with only one constant to compute.

The mathematics involved in computing the best estimates of the constants is this:-

For each observation, the difference between the calculated and observed values is error $= (y - a - bx_1 - cx_2 - dx_3 - \ldots)$
So the sum of the squares of the errors, which we will minimise by our choice of a,b, c, ..., is $\Sigma(y - a - bx_1 - cx_2 - dx_3 - \ldots)^2$.

The minimum occurs when the partial derivatives of each with respect to a,b,c,.. are all zero.

So $\frac{\partial}{\partial a} \Sigma(y - a - bx_1 - \ldots)^2 = 0$ or $\Sigma(y - a - bx_1 - \ldots) = 0$,

$\frac{\partial}{\partial b} \Sigma(y - a - bx_1 - \ldots)^2 = 0$ or $\Sigma x_1(y - a - bx_1 - \ldots) = 0$,

and so on.

As a concrete example, not too long, consider the linear relationship where two variables determine the answer, of this form:

$y = a + bx + cz$. The procedure above gives:

$a.n + b\Sigma x + c\Sigma z = \Sigma y$
$a\Sigma x + b\Sigma x^2 + c\Sigma xz = \Sigma xy$
$a\Sigma z + b\Sigma xz + c\Sigma z^2 = \Sigma yz$

With manual methods, values of x and z and the result y are tabulated, along with the products xy, xz, and yz, and the squares x^2 and z^2. The column totals give the figures for insertion into the simultaneous equations, which can be solved by matrix inversion (giving results identical to the transpose multiplication method).

Other methods   Other methods of curve fitting, adapted for less well-behaved functions, tend to be less easily computerised. The easiest methods assume that a formula is exact, and calculate parameters accordingly, perhaps averaging different estimates. For example, suppose $y = ma^x + nb^x$ is to be fitted to data on mortality. Any four equally-spaced sets of x and their observed y values correspond to exact values of m,n,a, and b. The ratio of $y(1)y(3) - y(2)^2$ and $y(2)y(4) - y(3)^2$ can be shown to be ab, and by algebraic juggling values can be extracted from the data. In practice graphs are plotted and the results tested for adequacy of fit. Another example: to fit data to the hypothetical underlying curve $y = a + bx + cd^x$ we can use the ratio of $y(4)-2y(3)+y(2)$ to $y(3)-2y(2)+y(1)$ to estimate the value of d^5. In examples of this type the initial algebra is probably easier to do manually than by computer. Other approximate methods exist, some of them dating from the days before the general availability of computers, but there is not sufficient space here to deal with them.

16.9 Machine-code programming with mathematics.

The floating-point accumulators and BASIC number storage   We have seen (16.1) how numbers are stored in BASIC and in tables in ROM, in 5 bytes of form exponent + sign bit + the rest as mantissa. Calculations are performed by Microsoft BASIC largely in two locations, called the floating-point accumulators. The more important of the two, 'Floating-point accumulator #1', which we can call FPAcc.#1, occupies $5E to $63 in the zero-page of RAM. (In BASIC 1 the locations are $B0 to $B5). Floating-point accumulator #2 occupies $66 to $6B ($B8 to $BD in BASIC 1). Each accumulator is thus six bytes long, one byte longer than variable storage. The arrangement of bytes is very similar in each case:

Floating-point accumulator #1

$5E	$5F	$60	$61	$62	$63
EXP	BIT+M1	M2	M3	M4	SIGN

Floating-point accumulator #2

$66	$67	$68	$69	$6A	$6B
EXP	BIT+M1	M2	M3	M4	SIGN

The sign byte is to some extent independent of the high bit in M1: if they are both set (i.e. sign byte has its high bit set, typically #FF) and the high bit of M1 is on, the number is considered negative when it is transferred to other parts of RAM. In addition to the bytes shown here, there is an overflow byte and a low byte used for rounding. Most of the results of calculations are stored in FPACC.#1; the other accumulator typically holds the number to be added or multiplied. Immediately below the two accumulators are 10 bytes which store numerals from RAM; these are not used in the same way for major calculation. These accumulators hold numbers with the maximum precision available to BASIC. As we have already noted, some calculations are intrinsically less accurate than others; for instance, 7*7*7*7*7*7*7 is evaluated exactly as 40353607, because the integer is held without error; but 7↑9, which uses an intermediate stage in calculation of finding the logarithm of 7, emerges as 40353607.1. The accumulators' contents can be watched by programs similar to those in Chapters 2 and 13. Is there a method to determine the value held in floating-point form? Not surprisingly, the answer is yes, and the following routine is a simple way to program it. The routine is relocatable, and the version below, for BASIC 2, starts at $033A. In this case therefore SYS 826 runs the routine, which prints a flashing cursor and awaits the input of four hex bytes and the 'Return' key. It prints the value of the 5-byte number starting at that location. For example, the following four values, which apply to BASIC 2, come from a table which BASIC uses to calculate LOG:

```
D8C8 1
D8CE .434255942
D8D3 .576584541
D8D8 .961800759
```

```
., 033A 20 B6 E7    JSR $E7B6    ;$E7B6 inputs a hexadecimal byte (e.g. A4)
., 033D A8          TAY          ;       into accumulator A.
., 033E 20 B6 E7    JSR $E7B6    ;$DAAE moves 5 bytes pointed to by A (low),
., 0341 20 AE DA    JSR $DAAE    ;       Y (high) into FPAcc.#1 and unwraps
., 0344 20 E9 DC    JSR $DCE9    ;       the sign byte.
., 0347 20 1C CA    JSR $CA1C    ;$DCE9 converts FPAcc.#1 into an ASCII string.
., 034A A9 0D       LDA #$0D     ;$CA1C prints the string.
., 034C 20 D2 FF    JSR $FFD2    ;$FFD2 prints carriage return.
., 034F D0 E9       BNE $033A
```

```
.: 033A 20 B6 E7 A8 20 B6 E7 20
.: 0342 AE DA 20 E9 DC 20 1C CA
.: 034A A9 0D 20 D2 FF D0 E9
```

Pressing the Stop key will terminate the loop. The routine can be used from within the monitor, by changing USRCMD or perhaps more simply by changing some command which is comparatively little used in a RAM monitor (e.g. Extramon) to point to this routine's start instead. The BASIC 4 version is this:

```
.: 027A 20 63 D7 A8 20 63 D7 20
.: 0282 D8 CC 20 93 CF 20 1D BB
.: 028A A9 0D 20 D2 FF D0 E9
```

Hexadecimal to decimal conversion and vice versa   The machine-code programs which are presented here can be called either from BASIC or from a machine-code monitor. Some versions of Supermon have a ready-made N command, which is normally disused, pointing only to the start of the monitor. For example, if Supermon when loaded in a BASIC 2 machine has a table near the end of commands (TFHD etc.) which include N, then search a little further forward for a 2-byte address pointing to $FD55. Change this to $0339 (1 byte before $033A) and the following routine will run:

```
.,  033A  20 6F C4    JSR $C46F        ;$C46F inputs a screen line into the buffer.
.,  033D  A9 00       LDA #$00         ; ($77) is set to $0200.
.,  033F  85 77       STA $77          ;$CC9F inputs and evaluates BASIC.
.,  0341  A9 02       LDA #$02         ;$D6D2 converts the contents of FPAcc.#1 into
.,  0343  85 78       STA $78                a 2-byte integer, A high, Y low.
.,  0345  20 9F CC    JSR $CC9F        ;$E775 prints A as a hex byte (e.g. F5).
.,  0348  20 D2 D6    JSR $D6D2        ;$FD56 start of monitor
.,  034B  20 75 E7    JSR $E775
.,  034E  98          TYA
.,  034F  20 75 E7    JSR $E775
.,  0352  A9 0D       LDA #$0D
.,  0354  4C 56 FD    JMP $FD56
```

```
.:  033A  20 6F C4 A9 00 85 77 A9
.:  0342  02 85 78 20 9F CC 20 D2
.:  034A  D6 20 75 E7 98 20 75 E7
.:  0352  A9 0D 4C 56 FD 00 00 00
```

This routine is designed to print one single value only, then jump to monitor; a loop may be added, to print repeat values, or an RTS to return to BASIC. The routine works like this:

```
.N 12345
3039
```

printing a single hex figure. This is the BASIC 4 version:

```
.:  027A  20 E2 B4 A9 00 85 77 A9
.:  0282  02 85 78 20 98 BD 20 2D
.:  028A  C9 20 22 D7 98 20 22 D7
.:  0292  4C BA D4
```

The next routine performs conversions the other way, from hex to decimal. Again we can take advantage of ROM routines which already exist, and again the routines are relocatable, though not between different versions of BASIC. The routine includes a loop, so that a series of conversions is possible; the stop key terminates the routine and returns to the monitor. This can be incorporated into a monitor, like the decimal-to-hex program. If it is, this type of keyboard transaction takes place:

```
.N B000 45056
.N 0200 512
.N DAAE 55982   etc.
```

This version works with BASIC 2; note the loop at the end, which is always taken:

```
.,  033A  20 EB E7    JSR $E7EB        ;INPUT CHARACTER (E.G. N FROM MONITOR)
.,  033D  20 B6 E7    JSR $E7B6        ;INPUT HEX BYTE INTO A
.,  0340  A8          TAY
.,  0341  20 B6 E7    JSR $E7B6
.,  0344  AA          TAX
.,  0345  98          TYA
.,  0346  20 D9 DC    JSR $DCD9        ;PRINTS 256*A + X (DECIMAL)
.,  0349  A9 0D       LDA #$0D
.,  034B  20 D2 FF    JSR $FFD2        ;OUTPUT CHR IN ACC'R
.,  034E  D0 ED       BNE $033D
```

BASIC 2 and BASIC 4 monitor dumps follow:

```
.:  033A  20 EB E7 20 B6 E7 A8 20        ;BASIC 2
.:  0342  B6 E7 AA 98 20 D9 DC A9
.:  034A  0D 20 D2 FF D0 ED

.:  027A  20 98 D7 20 63 D7 A8 20        ;BASIC 4
.:  0282  63 D7 AA 98 20 83 CF A9
.:  028A  0D 20 D2 FF D0 ED
```

SYS 829 /SYS 637 from BASIC ignores the subroutine which inputs 'N'.

ROM routines   Mathematical routines in ROM can be classified in several different ways. We can distinguish four main types of operation, all of which are necessary to a full BASIC:

(i) Routines which perform calculations using FPAcc.#1 only, leaving the result in FPAcc.#1. For example, adding .5, multiplying by 10, rounding, and evaluating trigonometrical functions are all operations which may be performed by ROM routines using only FPAcc.#1. Some of the results may be in non-standard format: INT leaves FPAcc.#1 holding 2 bytes equivalent to the floating-point value.

(ii) Routines to interchange the accumulators.

(iii) Routines to interchange floating-point accumulator(s) with RAM variables. One set stores the accumulator's contents into a position in RAM determined by pointers; another set takes the RAM value, putting it into a floating-point accumulator before carrying out calculations on it.

(iv) Binary operations, in which FPAcc.#1 is combined – added, subtracted, or whatever – either with FPAcc.#2 or other RAM contents according to pointers; or in which FPAcc.#2 is combined – perhaps by division or a power calculation – with a RAM value, and the result deposited in FPAcc.#1.

The destination of most calculations is FPAcc.#1; it is often necessary to store intermediate values, perhaps in the 10 bytes of RAM immediately below the two accumulators which I have already mentioned.

ROUTINES WHICH USE ONLY FLOATING-POINT ACCUMULATOR #1.

----ADDRESSES----			----FUNCTION----*
BASIC 1	BASIC 2	BASIC 4	
D6DA	D6D2	C92D	FLOATING-TO-FIXED converts FPAcc.#1 into a 2-byte integer in ($11) and, with the order of bytes reversed, in ($61). On exit A holds the high byte, Y the low.
D71E	D72C	C97F	ADD .5 adds ½ to FPAcc.#1.
D8BF	D8F6	CB20	LOG$_e$ converts FPAcc.#1 into its logarithm.
D81C	D853	CA7D	TWO'S COMPLEMENT replaces FPAcc.#1 with its 2's comp.
D9B4	D9EE	CC18	MULTIPLY BY 10.
DAED	DB27	CD51	ROUND FPACC.#1 rounds using the extra byte.
DAFD	DB37	CD61	FIND SIGN on exit, A=0 (if FPAcc.#1 = 0),1 (if +ve), or #FF (if -ve).
DB2A	DB64	CD8E	ABS converts FPAcc.#1 into ABS(FPAcc.#1).
DB2D	DB67	CD91	COMPARE compares FPAcc.#1 with the 5-byte floating-point value to which A (low byte) and Y (high byte) point On exit, A=0 if the numbers are equal, 1 if FPAcc.#1 > memory, or #FF if FPAcc.#1 < memory.
DB6D	DBA7	CDD1	FLOATING-TO-FIXED converts FPAcc.#1 into a 2-byte integer in $61 (high) and $62 (low). Unlike D6DA/ D6D2/ C92D, the range is not validated.
DB9E	DBD8	CE02	INT finds INT of FPAcc.#1, leaving it in floating-point.
DBBB	DBF5	CE1F	ZEROISE puts nulls in FPAcc.#1 if the exponent is zero.
DE67	DEA1	D14B	NEGATE changes the sign of FPAcc.#1.
DEA0	DEDA	D184	EXP converts FPAcc.#1 into e (FPAcc.#1).
DF45	DF7F	D229	RND entry point (followed by branches for +ve, zero, and -ve arguments).
DF88	DFC2	D26C	FORCE RND RANGE ensures random number now stored in FPAcc.#1 will be within the range 0-1.
DFA5	DFDF	D289	SINE computes sine of FPAcc.#1, assuming the argument is measured in radians. Note that COS leaves pi/2 in FPAcc.#2, and therefore TAN does as well.
E048	E08C	D32C	ARCTANGENT

*Zero-page addresses apply to BASICs 2 and 4; BASIC 1 has different values, listed in Chapter 15. This list is not intended to be exhaustive.

OTHER MATHEMATICAL ROUTINES IN ROM

----ADDRESSES----			----FUNCTION----
BASIC 1	BASIC 2	BASIC 4	
C863	C873	B8F6	FETCH INTEGER FROM BASIC and leave its value in ($11)
C91C	C928	B9AB	ADD ASCII DIGIT TO FPACC.#1. ($1F),Y points to it.
CCB8	CC9F	BD98	INPUT AND EVALUATE ANY BASIC EXPRESSION. See Chapter 15 on this. Note that earlier entry points enable tests for string or numeric functions to be included.
CED6	CEC8	C086	OR performed between two 2-byte integers,
CED9	CECB	C089	AND performed between two 2-byte integers, leaving the result in FPAcc.#1.
D09D	D08D	C2DD	INPUT AND EVALUATE INTEGER EXPRESSION converts a BASIC expression which evaluates to 0-65535 into a 2-byte integer in $61 (high) and $62 (low).
D285	D27A	C4C9	POS puts cursor position on line into FPAcc.#1.
D287	D27C	C4CB	STORE Y REGISTER IN FPACC.#1 in floating-point form.
D654	D656	C8B2	LEN.
D663	D665	C8C1	ASC.
D685	D687	C8E3	VAL. Each of these leaves the result in FPAcc.#1.
D275	D733	C986	SUBTRACT replaces FPAcc.#1 by FPAcc.#2 - FPAcc.#1. See Chapter 15 for two entry points, one of which loads FPAcc.#2 from pointers, while the other uses current contents.
D73C	D773	C99D	ADDITION replaces FPAcc.#1 by FPAcc.#2 + FPAcc.#1. See Chapter 15 for entry-points.
D8FD	D934	CB5E	MULTIPLICATION replaces FPAcc.#1 by FPAcc.#1 * FP Acc.#2. See Chapter 15 for entry-points.
D95E	D998	CBC2	LOAD FPACC.#2 loads the 5-byte value to which A (low) and Y (high) point into FPAcc.#2.
D9D0	DA0A	CC34	DIVIDE BY 10. FPAcc.#2 is overwritten by FPAcc.#1.
D9E1	DA1B	CC45	DIVISION replaces FPAcc.#1 by FPAcc.#2 / FPAcc.#1. See Chapter 15 for entry points.
DA74	DAAE	CCD8	LOAD FPACC.#1 from pointers A (low) and Y (high).
DA99	DAD3	CCFD	STORE FPACC.#1 INTO MEMORY converts FPAcc.#1 into a 5-byte value stored in RAM. The position at which the bytes are stored is determined by pointers; see Ch. 15.
DACE	DB08	CD32	MOVE FPACC.#2 TO FPACC.#1 overwriting FPAcc.#1.
DADE	DB18	CD42	ROUND FPACC.#1 AND MOVE RESULT TO FPACC.#2.
DBC5	DBFF	CE29	CONVERT ASCII STRING TO NUMERAL IN FPACC.#1.
DC9F	DCD9	CF83	PRINT LINENUMBER prints 256*A + X on next line. (The value may be 0-65535).
DCAF	DCE9	CF93	CONVERT FPACC.#1 INTO ASCII STRING where the string is put into a buffer starting at $0100, ready to be printed, e.g. as a linenumber.
DE24	DE5E	D108	SQR where FPAcc.#2 holds .5
DE2E	DE68	D112	POWER converts FPAcc.#1 into (FPAcc.#2) (FPAcc.#1).
DF09	DF43	D1ED	SERIES EVALUATION ROUTINE. See section 16.6 on this.
DF9E	DFD8	D282	COS puts pi/2 into FPAcc.#2; leaves cosine in FPAcc.#1.
DFEE	E028	D2D2	TAN puts pi/2 into FPAcc.#2 and leaves tangent in FPA#1.

Many ROM routines are arranged so that different entry points give different results, according to the arrangement of pointers on entry. For example, SQR loads FPAcc.#2 with .5, then drops into the power routine, which automatically performs the square root; differently set pointers would cause the function to evaluate some other power. The routine at D8F9/ D930/ CB5A sets pointers and performs multiplication, having the effect of multiplying FPAcc.#1 by $\log_e 2$. These pointers are particularly important in the four major binary calculations of addition, subtraction, multiplication and division, and in those routines which load data from BASIC and store results back into RAM under BASIC control.

MATHEMATICAL TABLES IN ROM.

----ADDRESSES----			----VALUES----
BASIC 1	BASIC 2	BASIC 4	
CDBC	CDA3	BEA0	PI
D099	D089	C2D9	-32768.005
D891	D8C8	CAF2	1
			SERIES FOR LOG$_e$ counter = byte of 3, values are .434255942, .576584541, .961800759, 2.88539007. FOUR OTHER VALUES: 1/SQR(2), SQR(2), -.5, and LOG$_e$2 = .693147181.
D9CB	DA05	CC2F	10
DC85	DCBF	CEE9	99999999.9, 999999999.75, and 1000000000.
DDE3	DE1D	D0C7	.5
			15 constants held as 4-byte signed integers for use in string-to-numeral conversions and TI$ calculations. First 9 values (for strings) are -100000000,10000000, -1000000,100000,-10000,1000,-100,10,and -1. Last 6 values (for TI$) are -2160001, 216000, -36000, 3600, -600, and 60.
DE72	DEAC	D156	TABLE FOR EXP EVALUATION has 1/log$_e$2 followed by series counter = byte of 7, then values: .0000214987637, .00014352314, .00134226, .00961401, .0555051, .2402263, .693147186, 1.
DF3D	DF77	D221	2 CONSTANTS FOR RND 11 879 546.4 is multiplied, 3.927 677 78 E-8 added.
E01A	E054	D2FE	PI/2, 2*PI, .25 TABLE FOR SIN EVALUATION has series counter = 5 followed by: -14.38139, 42.007797, -76.70417, 81.605223, -41.3417021, and 2*PI.
E078	E0BC	D35C	TABLE FOR ATN EVALUATION has series counter = 11 followed by: -6.84793412 E-4, 4.85094216 E-3, -.0161117018 .034209638,-.0542791328, .0724571965, -.0898023954, .110932413, -.142839808, .19999912, .333333316, and 1.
E0CD	E111	D3B1	RND SEED of .811635157

Examples of the use of ROM routines to perform floating-point addition, subtraction, multiplication, and division. As a preliminary, to show how routines can be strung together, try the short routine which follows. It searches for a BASIC variable in RAM, loads the value into FPAcc.#1, converts the result into an ASCII string, and prints the ASCII string. The effect with variable XY, say, is identical to PRINT XY.

```
$0302 LDA $0300; RETRIEVE VARIABLE NAME FROM 768-769
$0305 STA $42   ; AND STORE IT IN ($42)
$0307 LDA $0301
$030A STA $43
$030C JSR $CFC9;SEARCH FOR VARIABLE IN MEMORY [BASIC 2]
$030F LDA $44
$0311 LDY $45
$0313 JSR $DAAE;LOAD POINTED-TO VALUE INTO FPACC.#1
$0316 JSR $DCE9;CONVERT FPACC.#1'S CONTENTS INTO ASCII STRING IN $0100ff.
$0319 JMP $CA1C;PRINT THE RESULT
```

POKE 768,65: POKE 769,65: SYS 770 prints the current value of AA; locations 0300 and

0301 hold the ASCII values of the variable's name (the second of these being #00 if the name has one character only). Note that some routines (e.g. DAAE) require the pointers to be loaded before they are called, whereas other groups of routines may have their pointers in common (e.g. DCE9 then CA1C) so there's no need to set the pointers. The version is BASIC 2; Chapter 15's list of equivalent addresses permits conversion to BASIC 4 or 1. Note that VARPTR (see Chapter 5), in conjunction with this and the following routines, provides a powerful way to interact with BASIC when performing calculations. Alternatively pure machine-code can be used, allocating floating-point values their own 5 bytes of storage in RAM. In this way, mathematical problems can be solved much more rapidly than BASIC permits, at the expense of the extra effort needed.

    The next example demonstrates addition and subtraction of two BASIC variables, which I've assumed have single-character names only, so the routine is kept short. Typical results look like this:-

```
A=23.1245: B=12340000
POKE 768,65: POKE 769,66: SYS 770: REM ADD
 12340023.1

A=15: B=3.14159265
POKE 768,65: POKE 769,66: SYS 770: REM SUBTRACT
 6.14305265
```

Only one subroutine needs to be called to change the operation from addition to subtraction. As we'll see, the routine can be converted to multiply and divide (and perform other calculations) too. This version is BASIC 2; conversion to either ROM 1 or ROM 4 is no problem.

```
$0302 LDA $0300; FIRST VARIABLE'S NAME
$0305 STA $42
$0307 LDA #$00 ; BOTH VARIABLES HAVE #0 HERE
$0309 STA $43
$030B JSR $CFC9; SEARCH FOR VARIABLE 'A'
$030E LDA $44
$0310 LDY $45
$0312 JSR $DAAE; LOAD FPACC.#1 WITH THE VALUE OF A (OR 0 IF NOT FOUND)
$0315 LDA $0301
$0318 STA $42
$031A JSR $CFC9; SEARCH FOR SECOND VARIABLE, E.G. 'B'
$031D LDA $44
$031F LDY $45
$0321 JSR $D773; LOAD FPACC.#2, THEN ADD RESULT TO FPACC.#1
$0324 JSR $DCE9; CONVERT FPACC.#1 INTO ASCII STRING, AND ...
$0327 JMP $CA1C; PRINT IT
```

D773 can be replaced by D998/ D77B which first loads FPAcc.#2, then enters the subroutine to add the two accumulators. In our example this makes no difference, but we could perhaps make use of this by checking the value in FPAcc.#2 or in some other way.

    Subtraction can be demonstrated with the identical routine, except that D773 is replaced by D733, or by the equivalent D998/D736. Again, these are BASIC 2 locations, which must be converted to the correct values for your ROM.

    Multiplication and division are easy to demonstrate with the same driver program; typically, this sort of result will appear:

```
A=55: B=-3
POKE 768,65: POKE 769,66: SYS 770:REM MULTIPLY A WITH B
-165

A=123456: B=654321
POKE 768,65: POKE 769,66: SYS 770: REM DIVIDE B BY A
 5.30003402
```

The BASIC 2 locations to multiply are D934 or JSR D998 then LDA 5E/ JSR D937. The point of loading A with FPAcc.#1's exponent is to exit if the exponent is zero, because this is a convention which shows that FPAcc.#1 contains zero, and therefore ought always to give a product of zero. Division is performed by DA1B or by D998 then LDA 5E/ JSR DA1E. Here, the extra test looks for division by zero errors.

Example of long-precision calculation   The most elegant way to implement extra precision in calculations (from BASIC) is to assign strings with the two values concerned, and return the result in a string. There is insufficient room here to explain how this can be done with all four main operations. This example multiplies two integers, with no loss of precision, in machine-code. A limit of 128 digits each has been used, so the maximum length of the result is 256. The routine is not particularly fast. It uses the screen locations to store intermediate results, so the process can be watched running. This diagram shows how the screen is used as three buffers; the illustration has 12345678*11246 set up. The order in which the numbers are entered affects the timing; in the same way that **9897** is easier than   **11.**

           x11   **x9897**

`0000000012345678`	`00011246`	`0000000000000000`
↑	↑	↑
$8000 = Number	**$8100 = Multiplier**	**$8200 = Solution**
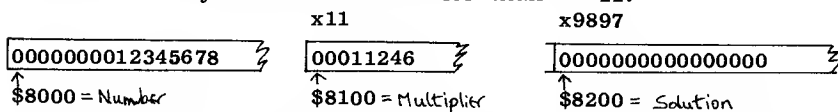
The first buffer is moved one step at a time to the left, and zero inserted at the right; this value is added to the third buffer as many times as the second buffer indicates. For example, the first loop starts by adding the contents of $8000ff six times to $8200 ff; when $8000ff has been moved one byte leftward (multiplying by 10), the result is added four times to $8300ff, and so on. The routine works independently of ROM; it is not relocatable – line 60 of BASIC pokes in the length of the numbers in use, and the very first command, 'AE 83 03', is LDX $0383, which does not relocate; however, the changes required are small. Some specimen runs of this program are shown. Addition and subtraction are both easy with this type of approach. Division is more difficult.

```
0 REM
1 REM **** INPUT NUMBERS AS STRINGS ****
2 REM
10 INPUT " FIRST NUMBER"; N1$
20 INPUT "SECOND NUMBER"; N2$
30 IF LEN(N2$) < LEN(N1$) THEN N2$="0"+N2$: GOTO 30.
40 IF LEN(N1$) < LEN(N2$) THEN N1$="0"+N1$: GOTO 40
50 N = LEN(N1$)
60 POKE 899,N-1: POKE 911,2*N-1: POKE 952,2*N-1
99 REM
100 REM **** POKE IN ZEROS AND INITIALISE ****
101 REM
110 FOR L = 32768 TO 32768+ 2*N-1: POKE L,0: NEXT
120 FOR L = 33024 TO 33024 +   N-1: POKE L,0: NEXT
130 FOR L = 33280 TO 33280 +2*N-1: POKE L,0: NEXT
199 REM
200 REM **** POKE IN VALUES ****
201 REM
210 FOR L=32768+N TO 32768 + 2*N-1: POKE L, VAL(MID$(N1$,L-32768-N+1,1)):NEXT
220 FOR L=33024 TO 33024+N-1: POKE L,VAL(MID$(N2$,L-33024+1,1)): NEXT
230 SYS 900
299 REM
300 REM **** PEEK AND PRINT RESULT ****
301 REM
310 FOR L = 33280 TO 33280+ 2*N-1: V$=STR$(PEEK(L)): V$=RIGHT$ (V$,1)
320 IF V$<>"0" THEN F=1
330 IF F=1 THEN   PRINTV$;
340 NEXT
```

```
.: 0384 AE 83 03 BD 00 81 F0 1A
.: 038C AA 18 A0 07 B9 00 80 79
.: 0394 00 82 C9 0A 30 03 38 E9
.: 039C 0A 99 00 82 88 10 ED CA
.: 03A4 D0 E7 CE 83 03 30 17 A2
.: 03AC 00 A0 01 B9 00 80 9D 00
.: 03B4 80 E8 C8 E0 07 D0 F4 A9
.: 03BC 00 9D 00 80 F0 C2 60 FF
```

FIRST NUMBER ? 137137137137137

SECOND NUMBER ? 99999999999

13713713713576562862862863

FIRST NUMBER ? 55555555555555555555555555

SECOND NUMBER ? 7777777777777777777

432098765432098765388888845679012345 67901235

The series calculation routine   Mathematical functions are not evaluated by a table lookup method, but by calculating a fixed number of terms of a series; the length of the series depends on its speed of convergence. We shall see in this subsection how the process works and how to write functions which can use the evaluation routine. Firstly, let's see where it is in ROM: in fact there are two routines, one of which is called as a subroutine of the other, and which is only once called to evaluate a series (by EXP). Most routines call the more complex first routine. The locations are:

> BASIC 1: DEF3 (main routine) & DF09 (subroutine). Pointer is ($C0).
> BASIC 2: DF2D (main routine) & DF43 (subroutine). Pointer is ($6E).
> BASIC 4: D1D7 (main routine) & D1ED (subroutine). Pointer is ($6E).

In fact, the first routine is largely concerned with housekeeping, i.e. making sure that the numerous values which are stored do not get overwritten. The second routine performs all the calculations. To use it, load the pointer with the starting byte of the series. For example, DE72/DEAC/D156, depending on ROM, holds a table of eight floating-point values preceded by a single '7'. The single byte is a counter, which the evaluation routine uses to count its multiplications. Slightly confusingly, it uses a total of one more constant than appears in the byte – 8 in the present example – where the final value is simply added to the cumulative total. Let's take a concrete example, with BASIC 4 this time:

```
LDA #$56
STA $6E
LDA #$D1
STA $6F   ; POINTER ($6E) NOW POINTS TO $D156, I.E. '7' THEN 8 NUMERALS
JMP $D1ED; EVALUATE SERIES AND LEAVE RESULT IN FPACC.#1
```

After this routine, FPAcc.#1 holds

$$1 + .693...x + .24...x^2 + .0555...x^3 + ... + .000021...x^7$$

where x = the starting value in FPAcc.#1. The coefficients are taken from the table for EXP evaluation, several pages before this one. Note that they appear in reverse order, because a recurrence relation like this has been used:

$$Result = (((..((Value_1*x + Value_2)*x + Value_3) * x + Value_4) ...)+ Value_n.$$

USR provides an easy way to observe the results we can get so far, because it puts the argument into floating-point accumulator #1, jumps to our routine, then prints the value now in FPAcc.#1 if we have USR(X), say. We find, on POKEing locations 1 and 2 so that they point to the short routine above, that if X is in the range 0-1, PRINT USR(X) is almost exactly 2^X; outside this range the approximation progressively worsens. This is how CBM BASIC calculates functions. Of course, functions generally aren't restricted so that their arguments appear only in a small range like 0-1; a transformation is used with some values, and not others, to put any value into a form in which systematic errors are minimised. For example, SIN not only takes the remainder after dividing by 2 pi, but also subtracts the result from $\pm\frac{1}{2}$, depending on the sign, so that the powers of x converge more rapidly.

We can write our own series, and calculate our own functions in machine-code, using this routine. For example, POKE 1,122: POKE 2,2 [i.e. $027A] and

				with			
$027A	LDA	#$00			$0300	1	;DECIMAL VALUE
$027C	STA	$6E			$0301	128	
$027E	LDA	#$03			$0302	0	
$0280	STA	$6F			$0303	0	;=½ IN FLOATING-POINT FORMAT
$0282	JMP	$D1ED;	BASIC 4		$0304	0	
					$0305	0	
					$0306	0	
					$0307	0	;=0 IN FLOATING-POINT FORMAT
					$0308	0	
					$0309	0	
					$030A	0	

is an easy example, PRINT USR(X) giving $0+\frac{1}{2}X = \frac{1}{2}X$. For instance PRINT USR(10) prints 5, PRINT USR(44.7) prints 22.35. (The first byte in the table cannot be 0; if it is, 256 terms will be evaluated, which is probably not the intention). The maximum power of x equals the single byte at the start of the table; so if x^5 provides enough accuracy, a single byte of 5 must start a table of 6 values, making 31 bytes in all.

It may be worthwhile writing a series routine for functions which are often used in some specialised field; the resulting calculations will be faster than a BASIC function definition. The point about Microsoft's method is that it does *not* simply employ an infinite series which has been truncated; instead, the maximum error within a defined range is kept low by finding an expression of best fit according to least squares criteria. Taking sine (x) as an example, this can be expressed as a series:

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

but, given a defined range, we can always improve on the truncated series; thus, if x is between 0 and pi/2 only, and we want an approximation only as far as cubed terms, this series:

$1.01x - .0424x^2 - .126x^3$　　is better than $x - x^3/6$, having a smaller maximum error of about .002 as against .08 .

A fairly routine method to calculate such series approximations is presented here; the interested reader should read further in Chebyshev and Legendre polynomials et al. The chief problem is that the accuracy is limited to that of the CBM, assuming that machine does the calculations; this means it is impossible to get results precise down to the last bit. In view of this difficulty, I have selected an easy method, rather than one giving the best possible results. The object is to minimise the sum of squares of differences between the function and its series approximation over some range, selected in a way that enables any (valid) argument to be processed by the series. To minimise over a range, we minimise an integral. As an illustration, we'll take a general function f(x) and approximate it by $(a + bx + cx^2)$ over the range x = 0 to x = 1. This shows the method, with the minimum of arithmetic.

At each value of x, the error = $a+bx+cx^2-f(x)$.

So the sum of squares of errors in the range 0 to 1 is given by:
$\int_0^1 (a+bx+cx^2-f(x))^2 \, dx$.

To minimise the entire expression we integrate the partial differentials with respect to a,b, and c; this gives three equations, since there are three unknowns:-

$$\int_0^1 (a+bx+cx^2-f(x)) \, dx = 0,$$

$$\int_0^1 x(a+bx+cx^2-f(x)) \, dx = 0,$$

and $\int_0^1 x^2 (a+bx+cx^2-f(x)) \, dx = 0$.

The expressions in x are easy to integrate; the expressions in f(x) may be integrable analytically, or a rule such as Simpson's can be used which will probably be quick and easy. The intermediate result is these equations:

$$[\frac{a}{1}x + \frac{b}{2}x^2 + \frac{c}{3}x^3]_0^1 - \int_0^1 f(x)dx = 0,$$

$$[\frac{a}{2}x^2 + \frac{b}{3}x^3 + \frac{c}{4}x^4]_0^1 - \int_0^1 xf(x)dx = 0,$$

and $[\frac{a}{3}x^3 + \frac{b}{4}x^4 + \frac{c}{5}x^5]_0^1 - \int_0^1 x^2 f(x)dx = 0$.

And these can be simplified into this final form:

$$a/1 + b/2 + c/3 = \int_0^1 f(x)dx,$$

$$a/2 + b/3 + c/4 = \int_0^1 xf(x)dx,$$

and $a/3 + b/4 + c/5 = \int_0^1 x^2 f(x)dx$.

These equations can be solved using our matrix inversion program, and estimates of a,b, and c found. The matrix is not ideal from the computational point of view, but we needn't worry too much about that. Note that the matrix has different elements if the limits are not 0 to 1; and the limits of the integrals of course differ too.

Let's finish with a short example: what is the best approximation to e^x in the form $(a + bx)$ where x varies from 0 to 1 only? The solution for a and b comes from:

$a + b/2 = \int_0^1 e^x dx = e^1 - e^0 = e-1,$

and $a/2 + b/3 = \int_0^1 xe^x dx = [xe^x - e^x]_0^1 = 1.$

Which gives $e^x \doteq .873 + 1.690x$ in the required range.

CHAPTER 17: PROGRAMMING FOR BUSINESS AND EDUCATION

17.1 Business programming.

17.1.1 Types of systems

"If predictions of millions of computers in the English-speaking world are true,
then the present situation is not even a drop in the ocean" (PL)

At the time of writing there are reported to be about 100,000 microcomputer systems in the U.K., of which a substantial proportion are Commodore machines. About 2% of users are in user groups. The general level of expertise is not very high; considering the complexity of these machines, this is, of course, not surprising. Most CBM hardware is distributed through official CBM dealers, and the presence of such dealers helped Commodore to achieve its leading market position. Software is a rather different matter. Most microcomputer manufacturers want to make and sell hardware, which is in any case logically prior to software; sources of software are far more diffuse and various, and may not come into existence for years after the introduction of a machine, so there is no guarantee that software which is perfectly feasible technically will actually exist. There are broadly four microcomputer markets: business, science, education, and personal computing ('home' computing is sometimes distinguished from 'personal' computing). In the U.K., the home computing market is small, because of the cost of the machines; or at least this was the case until Clive Sinclair introduced his ZX-80 and -81. On the other hand, this market sector is probably much larger than appears from the figures, because many machines bought for 'business', for tax reasons, must effectively be used personally. Estimating the proportion of machines in *serious* use is difficult; my own impression is that many microcomputers in 'education' are very much underutilised, and that a significant proportion of business machines fall into disuse after a fairly short time. Whether failures are principally a hardware matter or caused by software remains a further unclear area. However, it is at least clear that experienced hardware support and after-sales service are likely to be necessary to a successful system.

As regards software, the most widely-used business systems appear to be those which use data which is not crucial in day-to-day running of a business; mailing lists, price lists, in-house telephone directories, address book systems, quarterly requests for fees, illustrate the type of thing. Sales and purchase ledgers, order processing, and payroll, although potentially almost universal, seem more resistant to micro-computerisation. From the buyer's viewpoint, systems are very hard to assess, as we shall see, so caution is understandable. The situation is not particularly easy for the programmer and/or analyst either; a good system may be expensive to produce, perhaps prohibitively so, and a user may not appreciate the problems which may occur with relatively insecure systems. A package may be copied or bootlegged; a client may request impossible things; a specification may be changed at the last moment. The sections which follow will, I hope, cast some light on these issues, without necessarily offering definite solutions.

17.1.2 One-off ('bespoke') systems

"It will be nice to have a machine that does exactly what I want" (Anon)

Most one-off systems rely on previous programming work; in an extreme case, only a client's name or company need be changed throughout a set of programs. Normally, standard routines or methods can be used. Consequently, such systems can legitimately vary enormously in price. Another source of variation is the error-trapping and validation of the programs. This ought to be tailored to match the level of skill of the users. If unmotivated staff are going to be expected to key in large amounts of data, properly validated input, the use of menus, and comprehensive instructions have to be provided, otherwise there will inevitably be errors. In any case, some form of audit trail or record on paper must be printed in case of loss of data. If it can be arranged automatic file backup is desirable, since the users may not understand the importance of copies of data. Much of this is far less important in the case of programs of the sort described in the last section, which (for example) scan an address file by name or occupation, or perform some set inland revenue calculation. Normally, the category into which a program falls is fairly obvious to a programmer of even moderate experience: it is usually clear whether or not a proposed system is too large or likely to

overstretch the hardware. Sometimes it is impossible to be sure, so a preliminary trial may be needed. For example, is it possible to write a system which can plan the routing of film cameras and other news equipment by air to several cities (say, some capitals in Europe) in an efficient way?

When computerisation is considered, a client may not appreciate the importance of a clear specification of a system's function; in particular, the fact that although almost any system *can* in principle be programmed, a cost-effective approach may require the drawing of more-or-less artificial boundaries. For example, mailing-lists have 'dead wood' procedures, which remove names after (perhaps) a certain number of non-responses, but respondents must be categorised in a common-sense way, ensuring that Category 'A' are not removed until a larger number of mailings than Category 'B'. The fineness of the categorisation can be decided on the basis of experience.

As an illustration of requirements which clients may request of software houses, let's consider the following typical list (modified from an article by P Crozier in Computer Weekly, Jan.'80): (1) The software should be modifiable by the unskilled layman himself whenever his requirements change; (2) The system should be as foolproof as possible. For example, it should be impossible for programs to be run in the wrong order, and there should be no chance of data being lost through lack of backup copies. (3) The system should be simple to operate. A single rule should control the operation of the entire system. (4) The system should be a 'fully integrated business system'. (5) Other applications programs should work with the system, even when they are written by other organisations or individuals. (6) Reports (i.e. printouts of significant aspects of the data as it is currently stored) should be obtainable at any time. (7) Help must be available - 'within one hour's drive' is the criterion suggested. (8) Take account of ongoing costs, not just purchase price. How realistic is a list of this sort? In the case of most microcomputers, we can immediately supplement it with the requirement that a validation program of some sort should be available which can verify that the stored data has not been corrupted. Hashtotal techniques provide one method. Without this, a suspicion that the floppy disks may hold 'bad' data may be always present. The use of passwords to access the system is also sometimes thought to be desirable. Most of the list's desiderata are difficult to achieve, and are much more restrictive and stringent than most mainframe systems apply. Let's look at the eight points in turn:

(1) <u>Software modifications.</u> Programs which are parameterised, or which keep current parameters on file, are easily modified by users: for example, a set of percent increases applicable to a number of classes of items can be soft-coded so that the user has the option of altering them. Similarly, titles, headings, and printed comments in general may be programmed in a way enabling them to be changed. Apart from these rather elementary examples, things are less straightforward. Programmers may not want their programs to be accessible to the users; this depends on the commercial relationship between the software supplier and user. There is a further problem of determining responsibility in cases of failure of a modified system.

(2) <u>Foolproofing.</u> There is no way to make a system completely foolproof; the attempt may even be counterproductive, if an elaborate set of checks gives the users too much confidence in a system's ability to recover from mistakes. The single most valuable insurance against error is a competent user to whom the backup procedure, the operating procedures, and the function of individual programs has been explained.

(3) <u>Simplicity.</u> It is a mistake to assume that the shortest commands are the most efficient, i.e. that the number of keystrokes is inversely related to efficiency. Single-key triggering of important system functions can be disastrous. This of course is the reason for queries like 'Are you sure?' in BASIC 4. A uniform set of data-entry conventions is important in any system. These should incorporate the normal validation features, but must also make provision for the correction of wrong entries, by (for example) redisplaying the contents of a record before it is filed, and allowing any of its fields to be called by number to be corrected. See section 17.1.4 for more on this subject.

(4) <u>Fully integrated system.</u> The point here - which is probably more relevant to packages - is that all parts of a system should ideally be developed as the result of an overall analysis, and not assembled in a piecemeal fashion which may cause unexpected failures if the subprograms do not fit together correctly. From the point of view of a buyer, there is no way of knowing whether or not a system is 'integrated' in this sense, so there seems little point in laying stress on this.

(5) <u>Compatibility with other programs.</u> Every system relies upon its files having a structure which may well be unique to the program. It is impossible to ensure that any system will be *generally* compatible with all other software. To a limited extent, however, this is possible; a number of packaged software products are able to use each others' files, notably in word processing packages where the file structure is often a relatively simple dump of consecutive ASCII strings.

(6) <u>Immediate availability of reports.</u> This of course is correct. Easily available reports are valuable not only because of their immediate usefulness but also as a check on the working of a system. However, there may be some problems with reports in which values are reset. A program to provide end-of-period summaries, which reset totals to zero in preparation for the next period, cannot be run at any time, so it may be necessary to separate the reporting part of a program from the resetting part.

(7) <u>Availability of experienced help.</u> In any serious system, this is vital. It is not very easy to ensure continuity over time. Software people may be reluctant to provide continual after-sales service, which might be perceived as a drain on the resources of the company. There is a further problem that a system may change over a period, so that there may be no software person familiar with some version of a program, although this is usually a problem only with packages. Moreover, a company may have so many different systems written that keeping track of them all is very difficult. In practice, therefore, even 'experienced' help may be less useful than the owner of a system hopes. Some software houses offer no maintenance whatsoever.

(8) <u>Costs.</u> Although it seems obvious that the costs should allow for the possibility that changes in programs will be wanted, users generally have no way of estimating the extent of such enhancements or their costs, which may be large. One way around this is the fixed-price or 'turnkey' system. See section 17.1.9 for comments on the advantages and drawbacks of such schemes.

17.1.3 Packages

"Standard software packages are unlikely to be faulty" (Dept. of Industry guide)
"Don't ever buy a system without seeing it demonstrated. The reason?
There's so much under development. The good ones are pleased to
demonstrate it. The others ... well ..." (MH)

From a programming point of view, packaged systems are not very different from one-off systems. The main differences are (i) the target market must be fairly well understood, and (ii) problems of copying and bootlegging may arise. Section 17.1.10 looks at this second point. Very often the first is taken care of by one of the partners in the software-producing venture: accounting, medical, legal, and games packages have been produced with the co-operation of people with expertise in these fields. The intention is of course to sell a relatively large number of standardised program products at a price lower than would be possible with a once-only program. All purchasers can be offered similar service terms, and possibly regular updates to their programs should bugs be found or (for example) government rules changed. This is however not as easy as it appears at first sight. If all users want a product which has been thoroughly tested ('received its baptism of fire...') by other users, it is difficult to see who is to undertake the initial testing. And as a product improves, its qualities make it automatically a candidate for copying. Copyproofing is not easy; it is possible that Commodore may offer help in this, but they may not.

It might appear that purchasers have an easy time, but this is not really the case. For reasons discussed in 17.1.5, reviews of packages are not likely to be of much value. Users may be offered old versions of packages, and be completely unable to find out how up-to-date is the version being sold them. The exceptions are widely-sold packages, such as the -Calc range of programs which started with Visi-Calc (TM), the characteristics of which are well-known. Word-processing packages are also widely known and understood: Wordpro (TM) and Wordcraft (TM) illustrate the type of thing obtainable at present. A user should however still test such products, if he is concerned to actually use them. He may find that the facilities for name and address insertion in letters are inadequate, or that his printer is not catered for, or that his operators cannot use it. Also, of course, at any given time there is a 'state of the art' to which most packages conform and which may not have been pushed to a high level of development. For example, no word processor packages so far as I am aware offer proportional spacing, even though modern daisywheel printers are equipped to handle this, on microcomputer systems. Automatic hyphenation, using a library of prefixes and suffixes to insert breaks in words (sometimes unsuccessfully: pro-mpt) also seems to be non-existent, although it is technically feasible.

17.1.4 Input/Output

"It can take years to key in data..." (MH)

We've seen (Chapter 4) some examples of 'input' statements which use 'GET' to give full control over a system's input. The object of this is to avoid INPUT, which has many features making it unsuitable for *serious* applications, such as rejecting commas, not distinguishing between shift-space and unshifted space, and allowing unwanted cursor-control movements to take place. Systems designed for use by typists may need special validation, for example to ensure that the figure '1' is not entered as lower-case 'L'. The most thorough utility provided by Commodore is the 'Standard User Data Entry Environment', published as assembler source-code in CPUCN, Vol. 3, Issue 3, in an article by Paul Higginbottom. BASIC 2 and 4 versions are printed in the same article. The routine is too long for description here, but basically is stored in high RAM where it inputs strings into a predimensioned string array. The position of the fields is *not* determined by numerical parameters; instead the screen is scanned for delimiters, which the published version takes to be '<' and '>' to mark the start and end positions respectively. Since it is scanned from $8000 upwards, the fields are automatically input in consecutive order. This is the 'Data Entry Editor'; it is only a part of the 'Environment' which Commodore say in the article that all software must roughly conform to in order to receive 'Commodore approval'. Readers may therefore be interested in the following brief summary of the standards:

(1) A title on the top line(s) should describe the current program. The bottom line should be reserved for error messages or prompts ('Enter YES if data OK'), and perhaps messages ('Please wait while search continues').

(2) 'C' should continue from one screen to the next; shift-return should be the code to accept an entire screen of information. [I do not personally believe that the use of shift-return in this manner is sensible, because typists don't usually accept a distinction between shifted and unshifted carriage return].

(3) [CLR] should return every field to its initial value. [HOME] should move the cursor to the first field. [UP] and [DOWN] should permit movement between fields. [INSERT], [DELETE], [LEFT], and [RIGHT] should allow editing of each field. This condition is likely to prove more difficult to program than any other feature. [RETURN] validates its field and moves the cursor to the next field if the field was not detectably invalid. [STOP] should provide a 'help' facility (i.e. a display of instructions, or - this may be more difficult - a return to the main menu).

A screen can enter 'screen accept/reject mode' when a final [RETURN] or cursor down leaves the last field of the screen, or when [SHIFT-RETURN] is pressed at any time. In this mode, cursor up, [HOME] or [CLR] are to act as rejections, and shift-return as accept. So two consecutive shift-returns at any stage accept data. [Personally, I prefer a more meaningful 'YES' or 'NO'].

A data entry method which is used on large computers is the keying-in of identical data by two different punchpersons, each set of data being separately stored on file, and compared by the computer. Discrepancies can then be corrected. The idea is that neither person keying in data bothers to correct anything. In the case of small machines, there may not be storage space to file duplicate data, or time to enter the data twice, so such methods are probably unsuitable.

While numeric data is standard in format, alphabetic data isn't, and it may be worthwhile to decide on standards so that reports etc. are uniform in appearance. See the examples below, one extracted from a list of standards, the other showing a report incorporating them.

A	[Amps]
C	[Centigrade] [Capacitance]
Cd	[Cadmium]
Hz	[Hertz = frequency]
L	[Liters]
Ni	[Nickel]

NKT-940-090E	1	128	Outer Vessel	B	SL	2.35	300.80
NKT-940-110B	4	103	Condenser	B	SL	2.20	226.60
NKT-940-130S	2	507	Peg Stopper	B	SL	0.57	288.99
NMT-410-B	3	3	Nitrometer,Lunge	C		38.25	114.75
OVB-201-010P	1	10	Oven,Griffin 200C	D	Y	73.44	734.40
OVH-700-Q	1	3	Floor Stand,Oven/Incubator	B	L	31.50	94.50
OVH-720-C	2	14	Floor Stand,Oven/Incubator	B		33.00	462.00
OVH-740-L	1	14	Floor Stand,Oven/Incubator	B		34.50	483.00
OVL-240-504G	5	7	Rack	B	L	9.43	66.01
OVL-240-509T	6	72	Clips	B	L	3.43	246.96
OVL-350-210Y	1	1	Drying Cabinet,220-240V ac	F	P	122.55	122.55
OVL-578-050S	1	17	Thermometer	B	L	3.53	60.01

Hardcopy of input data may be produced as the data is input, providing a record of a series of transactions: see the example below, which records two transactions only.

START OF BOOK STOCK IN ENTRY FOR 8/ 5/81

CATALOGUE NO.	QTY.IN	UPDATED STOCK FIGURE /	NO. TO ISSUE.	MAIN DESCRIPTION OF ITEM	COST PRICE	DELY.	CROSS-REF	LOCN.
CKL-721-071H	15	New Stock Level:	,Issue: 15.	Thermistors	35.26			
CKL-721-131H	41	New Stock Level:	,Issue: 41.	Injection Port Seals	.68			

END OF BOOK STOCK IN ENTRY FOR 8/ 5/81

17.1.5 Testing systems
"Some of the faults [in packages] were quite incredible" (RW)
The quotation above came from a manager of a microcomputer department of a retail chain, whose policy was to stock only thoroughly tried and tested products. They employed external consultants to report on the good and bad points of software packages. They found plenty of each, but the relevant fact as far as this section is concerned is that the process was expensive, equivalent to about a year's salary per package. This is the reason for the fact (mentioned before) that reviews of packages, where they exist, are very likely to be superficial, and are often little more than quotations from handouts and press-releases on systems. It is also the reason that many users wait until a solid base of users is established before buying a package. From the programmer's point of view, bugs are not necessarily undesirable, since a client's independence is undermined to some extent if errors appear in his system. Moreover, a perfect program is a program eminently suitable for copying and piracy. Most mainframe packages are continually modified and upgraded, often with regular monthly update sheets circulated to registered users, although one hesitates to describe this as deliberate policy. With the computer industry at its present state of evolution, it is difficult to be dogmatic in this area.

17.1.6 Users and programmers
*"This guy rang and said his computer wasn't working. He'd typed in
'What is my birthday?' and it hadn't told him" (LS)*
The following short notes are intended to instruct, and warn, of potential hazards in the microcomputer arena caused by people rather than machines.

(1) Salesmen. Many microcomputer salesmen don't really know much about their machines, and perhaps can hardly be expected to. Advice from the technically competent will often be better.

(2) Typists. A computer like a CBM has a keyboard and is therefore automatically categorised by many office workers as a typist's thing. There are however a number of differences in style between typing and data input (for example, with regard to error correction) which may make the stereotype inappropriate. In addition, don't assume that someone can sit in front of a screen for eight hours a day; this may be too much. Four may be better.

(3) 'Users'. A 'user' can be defined as someone with immediate responsibility for a system. It may prove impossible to explain such concepts as disk copying, the need for tidiness and cleanliness, or the way a system works.

(4) Departments. Inter-departmental rivalries and differences in attitude may cause problems in any organisations other than the very smallest. Errors or omissions which are well-known in one department may never be formally mentioned to the people who are working with the microcomputer.

(5) Computer departments. Big organisations' computer departments may be actively opposed to microcomputers, partly for the good reason that they may go wrong. (This is why micros are often described as 'calculator with video display' or in some such terms when the department's budget requests are submitted). There may be a fear of 'distributed processing', or the managers may know nothing of micros. It is also quite common for mainframe people to be unable to appreciate the limitations of microcomputers, which inevitably lack most big-machine features.

(6) <u>Programmers.</u> Some readers may be interested in the general perception of programmers and analysts within the computer industry. Computing has evolved with little in the way of formal training and qualifications, so assessment of would-be computer personnel is fairly difficult. One hears of people describing themselves as 'very experienced programmer' who turn out to have two weeks' experience on a machine. The first belief is that analysts are extraverts and programmers introverts. This is at least a clear-cut theory, something which can hardly be said of the assortment of theories on programmers. I've heard it suggested that programming ability goes with neatness in form-filling, with the capacity to use jargon, with interest in chess and/or bridge, and with the desire to feel control over the machine. Disabled people have been recommended as potential programmers. People who are good at I.Q. tests are likely to be a good bet (I imagine), since the pencil-and-paper nature of the work and its emphasis on formal logic resembles the tests quite closely. Yet another belief is that programming skill can be expected to correlate with fluency in English.

17.1.7 Documentation
"Make sure you've got paper, paper all the way" (MH)
There is a British Standard on documentation: BS5515:1978 is the Code of practice for Documentation of Computer-based systems. This is exhaustive, but too comprehensive for microcomputer systems. Instead a small subset taken from the code of practice will probably serve most purposes, perhaps along these lines:

(1) <u>Operator's Manual.</u> This might include the procedures for switching on and off, and for handling disks, loading paper, loading programs, and so on. It should include an explanation of input conventions (e.g. as described in section 17.1.4), and also of error messages, even if these are supposed to be self-explanatory. If there are conventions to be followed when inputting data (as in 17.1.4) these should be listed, and a troubleshooting section of a reasonably elementary kind will help avoid panics caused by the printer running out of paper and problems of that sort.

(2) <u>User Manual.</u> This is intended to explain the system, without going to the lengths of including program listings and other technical documents. It could include an explanation of the file structure, a chart of the processing sequences of the programs, an explanation of the backup procedures to be adopted and the validation techniques to check for successful running, and a hardware section listing the suppliers with details of purchase dates, maintenance contracts, contact names, and so on.

(3) <u>System manual.</u> This should provide a complete reference to the working of a system. Typically a specification will be included, and a detailed breakdown of the file structures used, with field types, field lengths and so on. If the system is partly in BASIC, listings, subroutine maps by linenumber, variable tables, and details of wedges and IRQ alterations must be listed. Memory-maps of machine-code and annotated machine-code listings are needed. Finally, a log of updates and their (intended) effects should be kept.

17.1.8 Security
Sufficient security can usually be got by simply locking up the machine, perhaps taking home important disks or keeping them locked in lockable diskette cases. Backup copies of data should be kept separately, though. Some microcomputers are portable enough to be carried away; the PET/CBM range are on the heavy side for this. It is not unknown for chips to be taken out, however. Some users may like to ensure that no disks are taken into their computer room or taken out. Passwords may be useful if several groups of users run programs on the same machines, but these are likely to be vulnerable to competent programmers and are effective only with so-called 'naive users'. Complete duplicate systems are often feasible and may be worthwhile.

17.1.9 Contracts
A software house typically has a contract with its clients along these lines: For a fixed price, a system specification and programs will be produced in (say) twelve weeks, 'with no liability for variation'. After the system specification has been agreed with the client, work goes ahead; the client is expected to supply test data, and on delivery of the system the results produced by the system using the test data are supplied to the client, who is expected to check whether the results accord with the specificati . After an interval (perhaps four weeks) the programs are deemed to accord with the specification.

There may be penalty clauses if delivery is late; occasionally stories circulate of freelance programmers sued for malpractice. But generally something like the scheme above is adopted. It is in fact rather unfair on the purchaser; he may have to wait much longer than he thought, and be asked to supply test data without any real appreciation of how to do this - for instance, it may not occur to him to supply data with deliberate mistakes to check that the system rejects them. Also he may be asked to approve a specification with only a vague idea of what the resulting system will appear like in practice. Finally, the lapse of time allowed to detect bugs may be insufficient, some programs never being run.

17.1.10 Copying and 'piracy'

We can distinguish copying, where a friend or acquaintance copies a program for his own use, from 'piracy' or 'bootlegging', in which the copied program is not only copied but also sold. The cassette games market is said to have been killed by copying, some companies no longer bothering to sell them because copies quickly cause a severe drop in sales. Cassettes can of course be copied by audio methods, so that any software protection is simply bypassed. Disk programs are much more copyproof-able, but the methods are not widely known. In any case, it may only be a matter of time before programs to copy 'uncopyable' disks start to appear: this happened in 1981 to Apple, whose disk operating system is in RAM and more accessible than CBM's to disassembly. There are other, more subtle forms of copying too. I met an enthusiast at an exhibition who told me that he'd written a trade estimation program, which had reappeared, in improved form, but using exactly his methodology, as a commercial system. Beyond remarking on this problem, which is also endemic in the recorded music field, it is hard to suggest any solutions.* One suggestion is that users might be willing to pay for a newsletter of updates, or that an elaborate manual for a system might be easier to spot as a copy than a disk; and that a system might include spurious routines that perform no useful purpose, but can be looked for in a suspected pirate copy. But even if a pirate copy is certainly identified, legal action may hardly be worthwhile.

17.2 Programming in education.

"They work in an orderly way for hours. The attention-getting capacity [of microcomputers] is remarkable"   (DL)

17.2.1 Costs

Microcomputer costs have continued to drop; VIC is Commodore's low-price machine which is intended to compete in the cheap home and education markets. The apparent costs have dropped more rapidly than actual costs: when allowance is made for external TVs, cassette recorders, RAM packs (more expensive than RAM chips!), and other equipment, particularly disks and printers, much of the apparent saving of 'cheap' systems may disappear. In the U.K. it is official policy to support, or at least lean towards, British products; education authorities for example may fund only RML hardware and software, and perhaps Acorn machines with BBC BASIC. Something like this may happen in the U.S.A. if Japanese hardware becomes more popular. Cost is therefore not the only consideration, especially for schools. It has to be said that many people in the education sector have an unrealistic attitude to hardware and software. Letters are printed in the magazines from teachers who want to introduce computing at a cost of £2 per head. There is a widespread belief that schools spend vast sums on technical gadgetry: video recorders, televisions, and so on. The fact seems to be that wages and salaries take the lion's share of the huge sums paid on education, leaving not much for items like microcomputers. Often, parents (via parent-teacher associations) or companies which allot part of their budget to charities can be persuaded to part with money. Sometimes special funds for career training or for gifted students can be tapped. The technique, so far as I know, is to (a) appoint an

---

*Hardware solutions are successful with most users, and are likely to remain so unless imitations become widespread. The 'dongle' is a device fitting one of the ports. In its simplest form it might fit over the user port (which is more likely to be free than the IEEE port) and perhaps ground the diagnostic sense pin. This could be checked by the program in the same way that the reset routine selects between BASIC and the monitor (or - originally - the diagnostic routine), and, if not grounded, erase the program. More sophisticate versions might include timers and other circuitry. ROMs, e.g. in slot 9000-9FFF, can supplement the security of disk systems, and incorporate an identification number, although EPROM copiers make them less than foolproof.

enthusiast willing to do the work, and (b) encourage this person to approach likely sources of funds at as high a level as possible, telling them that the aim is to develop their skills in this field or that.

17.2.2 Programs
"We found most of Blanksoft's programs were execrable" (MB)
Computer Aided Learning ('CAL'), programmed learning, and teaching machines were first introduced in the sixties, and appear to have been a near-total failure. Present technology at least offers the hope of greater success, but this cannot be taken for granted. This section discusses some of the qualities which good educational software can be expected to possess, and some of its promising applications and topics.

(1) Multiple-choice questions. Because of their ease of marking, tests of this sort are fairly popular. The principle is simple enough: a question is posed, and a small number, say four, alternative 'answers' offered. Only one is supposed to be correct; the others may be deliberately chosen, perhaps empirically, to resemble the correct answer closely, or to be the correct answer to a slightly different question. The simplest scoring method is to give 1 mark for a correct solution, and deduct 1/3 of a mark for each incorrect question, where there are four alternative answers. This is a so-called 'guessing correction'. The rationale is that a respondent who answers questions at random will on average score 0, because three incorrect answers will just cancel out a single correct guess. In this way, reckless guessers are not rewarded. A language like Pilot (see appendices) can be used to generate programs of this sort, although some versions of Pilot may not enable scores to be kept.

(2) Tests graded by year and subject. A general question-and-answer session, provided as an off-the-shelf package, may be valuable. There is no need to adhere to a strict format; there may be calculation questions, vocabulary questions, comprehension questions. Ideally, therefore, several different programs on (say) second-year economics could be available, to be used by a student for self-assessment. Programs like this are quite difficult to write, and a standardised approach is vital if any sort of reasonable productivity is wanted.

(3) Packages explaining single concepts. Many scientific, mathematical and linguistic concepts can be made the subject of programs. Where the screen editing and graphics capacity is used well, the resulting program may be a valuable supplement to a lesson. Examples include: i. A program to demonstrate how histograms (bar charts) vary as the scales on which they are plotted, and the number of bars used, change. ii. Demonstrations of the relationships between the frequency of a sound and its pitch, including intervals such as thirds and octaves. iii. Simulations using random numbers. Any probability distribution can be tried; examples include the normal distribution, elementary distributions involving coins and dice, biological population models, and mathematical results derived from physics and chemistry. iv. Graph plotting: the idea of co-ordinates, the use of rectangular axes, the equations of some simpler curves. v. Series summation and the idea of a 'limit'. Special cases can be looked at: pi, e, and the golden section. vi. Concepts of calculus: differentiation can be taught as the calculation of the limiting gradient at a point, and integration as the addition of lengths, areas, or volumes of arbitrary smallness. vii. Mathematical economics. Supply and demand curves and deductions from them, fixed and variable costs, average and marginal costs are all readily computerisable as demonstrations. More complex simulations, such as the 'business cycle', can be illustrated too. viii. Simple linguistic ideas. Languages generally are too complex for microcomputers to get much purchase on, but useful programs can be written in restricted areas, such as vocabulary and translation testing. A successful program to test knowledge of German numbers (printing the correct German version of a written number, highlighting the response where it was incorrect) illustrates on sort of approach.

17.2.3 General attitudes
It is worthwhile to be aware of the two fundamentally different underlying attitudes possessed by converts to the cause of microcomputers in education. Both make claims which may be suspected to be excessive. The first group concerns itself with supplementary training-courses for teachers, with organising pupils so that each of the older pupils is allotted a certain amount of time per week, and with converting other teachers. This group is likely to produce popular programs, since the more subtle pitfalls require a fairly hard-headed approach to detect and avoid. Their arguments in favour of microcomputers read like this: Microcomputers are unparalleled

at teaching logical thought. They provide great opportunities for students to display their creativity. Learning about microcomputers may be the most important part of their schooling...

The second group has a more romantic approach, and is less concerned with matters of cost or pupil access, or of trying to assess the benefits of computer education. One pictures a roomful of 'disadvantaged' children, all concentrating on their computers, and in fact playing a number game. This group's argument for microcomputers reads: It is quite remarkable to see them working in an orderly way for hours. With microcomputers, increased equality of education is possible. Children who dropped out find their interest reawakened, and their confidence grows...

As far as teachers who are not involved in computer studies are concerned, microcomputers may be thrust upon them either by way of packages of the sort previously described, or in an administrative role. Programs to help plan timetables illustrate this latter category. A few hints show the sort of approach which may need to be adopted when planning educational software which has to receive these peoples' approval. In the first place, the cosmetic side of programs needs some attention. Lively and interesting graphics make a great deal of difference in all subjects, but perhaps particularly the more concrete subjects like biology and geography. Good graphics effects unfortunately are not easy to achieve. Another aspect of a program's appearance is the text: the CBM is fortunate in having lower-case, which is generally more readable than capitals only. Attention should obviously be given to the wording: it is not only teachers of English who object to being told 'Please get it rite' or asked to enter 'Any > A,B,C,D'. Sometimes teachers may be worried about the machine taking over from them. For example, they may reject a system which gives the student references on the topic under discussion. They may take the view that such information ought not to be issued too freely.

A great deal of software has been written, and software directories and indexes appear from time to time in the computer press. The U.K. reader interested in finding out more should contact this address:

The Council for Educational Technology
3 Devonshire Street
London
W1N 2BA
(Tel: (01)-636-4186)

Explicitly Commodore-related information can be obtained direct from Commodore or their dealers. MUSE (Microcomputer Users in Secondary Education) is what it says. Other interested parties include CAL News, based at Imperial College Computer Centre, and the Association of London Computer Clubs. I haven't listed addresses for these organisations, many of which are rather mobile.

TABLE OF OPCODES AND THEIR FUNCTIONS, BIT STRUCTURE,

Opcode	Description	Bit structure	N	V	B	D	I	Z	C
ADC	Add memory with carry to accumulator	011bbb01	N	V				Z	C
AND	Logical AND memory with accumulator	001bbb01	N					Z	
ASL	Shift memory or accumulator one bit left	000bbb10	N					Z	C
BCC	Branch if carry bit clear	10010000							
BCS	Branch if carry bit set	10110000							
BEQ	Branch if zero bit set	11110000							
BIT	AND with A, storing Z and bits 6 and 7	0010b100	M7	M6				Z	
BMI	Branch if N (negative) flag set	00110000							
BNE	Branch if zero bit clear	11010000							
BPL	Branch if N bit is not set	00010000							
BRK	Force break to IRQ	00000000			1		1		
BVC	Branch on internal overflow bit clear	01010000							
BVS	Branch on internal overflow bit set	01110000							
CLC	Clear the carry bit	00011000							0
CLD	Clear decimal flag (for hex arithmetic)	11011000				0			
CLI	Clear interrupt disable flag	01011000					0		
CLV	Clear internal overflow flag	10111000		0					
CMP	Compare memory to accumulator	110bbb01	N					Z	C
CPX	Compare memory to X register	1110bb00	N					Z	C
CPY	Compare memory to Y register	1100bb00	N					Z	C
DEC	Decrement memory location	110bb110	N					Z	
DEX	Decrement X register	11001010	N					Z	
DEY	Decrement Y register	10001000	N					Z	
EOR	Logical exclusive-OR memory with A	010bbb01	N					Z	
INC	Increment memory location	111bb110	N					Z	
INX	Increment X register	11101000	N					Z	
INY	Increment Y register	11001000	N					Z	
JMP	Jump to new address	01b01100							
JSR	Jump to new address, saving return	00100000							
LDA	Load accumulator from memory	101bbb01	N					Z	
LDX	Load X register from memory	101bbb10	N					Z	
LDY	Load Y register from memory	101bbb00	N					Z	
LSR	Shift memory or accumulator one bit right	010bbb10	0					Z	C
NOP	No operation	11101010							
ORA	Logical inclusive-OR memory with A	000bbb01	N					Z	
PHA	Push accumulator onto stack	01001000							
PHP	Push processor status flags onto stack	00001000							
PLA	Pull stack into accumulator	01101000	N					Z	
PLP	Pull stack into processor status flags	00101000	N	V	B	D	I	Z	C
ROL	Rotate memory or A one bit left, inc. C	001bbb10	N					Z	C
ROR	Rotate memory or A one bit right, inc. C	011bbb10	N					Z	C
RTI	Return from interrupt	01000000	N	V	B	D	I	Z	C
RTS	Return from subroutine called by JSR	01100000							
SBC	Subtract memory and C-complement from A	111bbb01	N	V				Z	C
SEC	Set the carry bit	00111000							1
SED	Set the decimal flag (for BCD arithmetic)	11111000				1			
SEI	Set the interrupt disable flag	01111000					1		
STA	Store accumulator into memory	100bbb01							
STX	Store X into memory	100bb110							
STY	Store Y into memory	100bb100							
TAX	Transfer accumulator to X register	10101010	N					Z	
TAY	Transfer accumulator to Y register	10101000	N					Z	
TSX	Transfer stack pointer to X register	10111010	N					Z	
TXA	Transfer X register to A	10001010	N					Z	
TXS	Transfer X register to stack pointer	10011010							
TYA	Transfer Y register to A	10011000	N					Z	

HEXADECIMAL VALUES, TIMING AND PROCESSOR FLAGS

Abs	Abs,X	Abs,Y	Zer	Zer,X	Zer,Y	Implied	Immed	Rel	Acc	(Ind,X)	(Ind),Y	Ind	
6D 4	7D* 4	79* 4	65 3	75 4			69 2			61 6	71* 5		ADC
2D 4	3D* 4	39* 4	25 3	35 4			29 2			21 6	31* 5		AND
0E 6	1E 7		06 5	16 6					0A 2				ASL
								90[2] 2					BCC
								B0[2] 2					BCS
								F0[2] 2					BEQ
2C 4			24 3										BIT
								30[2] 2					BMI
								D0[2] 2					BNE
								10[2] 2					BPL
						00 7							BRK
								50[2] 2					BVC
								70[2] 2					BVS
						18 2							CLC
						D8 2							CLD
						58 2							CLI
						B8 2							CLV
CD 4	DD* 4	D9* 4	C5 3	D5 4			C9 2			C1 6	D1* 5		CMP
EC 4			E4 3				E0 2						CPX
CC 4			C4 3				C0 2						CPY
CE 6	DE 7		C6 5	D6 6									DEC
						CA 2							DEX
						88 2							DEY
4D 4	5D* 4	59* 4	45 3	55 4			49 2			41 6	51* 5		EOR
EE 6	FE 7		E6 5	F6 6									INC
						E8 2							INX
						C8 2							INY
4C 3												6C 5	JMP
20 6													JSR
AD 4	BD* 4	B9* 4	A5 3	B5 4			A9 2			A1 6	B1* 5		LDA
AE 4		BE* 4	A6 3		B6 4		A2 2						LDX
AC 4	BC* 4		A4 3	B4 4			A0 2						LDY
4E 6	5E 7		46 5	56 6					4A 2				LSR
						EA 2							NOP
0D 4	1D* 4	19* 4	05 3	15 4			09 2			01 6	11 5		ORA
						48 3							PHA
						08 3							PHP
						68 4							PLA
						28 4							PLP
2E 6	3E 7		26 5	36 6					2A 2				ROL
6E 6	7E 7		66 5	76 6					6A 2				ROR
						40 6							RTI
						60 6							RTS
ED 4	FD* 4	F9* 4	E5 3	F5 4			E9 2			E1 6	F1* 5		SBC
						38 2							SEC
						F8 2							SED
						78 2							SEI
8D 4	9D 5	99 5	85 3	95 4						81 6	91 6		STA
8E 4			86 3		96 4								STX
8C 4			84 3	94 4									STY
						AA 2							TAX
						A8 2							TAY
						BA 2							TSX
						8A 2							TXA
						9A 2							TXS
						98 2							TYA

```
 *  +1 if index crosses page
 2  +1 if branch is taken,
    +1 more if page crossed
```

DECIMAL – HEXADECIMAL INTERCONVERSION TABLE

Hex	Low Dec.	High Dec.	Hex	Low Dec.	High Dec.	Hex	Low Dec.	High Dec.	Hex	Low Dec.	High Dec.
$00	0	0	$40	64	16384	$80	128	32768	$C0	192	49152
$01	1	256	$41	65	16640	$81	129	33024	$C1	193	49408
$02	2	512	$42	66	16896	$82	130	33280	$C2	194	49664
$03	3	768	$43	67	17152	$83	131	33536	$C3	195	49920
$04	4	1024	$44	68	17408	$84	132	33792	$C4	196	50176
$05	5	1280	$45	69	17664	$85	133	34048	$C5	197	50432
$06	6	1536	$46	70	17920	$86	134	34304	$C6	198	50688
$07	7	1792	$47	71	18176	$87	135	34560	$C7	199	50944
$08	8	2048	$48	72	18432	$88	136	34816	$C8	200	51200
$09	9	2304	$49	73	18688	$89	137	35072	$C9	201	51456
$0A	10	2560	$4A	74	18944	$8A	138	35328	$CA	202	51712
$0B	11	2816	$4B	75	19200	$8B	139	35584	$CB	203	51968
$0C	12	3072	$4C	76	19456	$8C	140	35840	$CC	204	52224
$0D	13	3328	$4D	77	19712	$8D	141	36096	$CD	205	52480
$0E	14	3584	$4E	78	19968	$8E	142	36352	$CE	206	52736
$0F	15	3840	$4F	79	20224	$8F	143	36608	$CF	207	52992
$10	16	4096	$50	80	20480	$90	144	36864	$D0	208	53248
$11	17	4352	$51	81	20736	$91	145	37120	$D1	209	53504
$12	18	4608	$52	82	20992	$92	146	37376	$D2	210	53760
$13	19	4864	$53	83	21248	$93	147	37632	$D3	211	54016
$14	20	5120	$54	84	21504	$94	148	37888	$D4	212	54272
$15	21	5376	$55	85	21760	$95	149	38144	$D5	213	54528
$16	22	5632	$56	86	22016	$96	150	38400	$D6	214	54784
$17	23	5888	$57	87	22272	$97	151	38656	$D7	215	55040
$18	24	6144	$58	88	22528	$98	152	38912	$D8	216	55296
$19	25	6400	$59	89	22784	$99	153	39168	$D9	217	55552
$1A	26	6656	$5A	90	23040	$9A	154	39424	$DA	218	55808
$1B	27	6912	$5B	91	23296	$9B	155	39680	$DB	219	56064
$1C	28	7168	$5C	92	23552	$9C	156	39936	$DC	220	56320
$1D	29	7424	$5D	93	23808	$9D	157	40192	$DD	221	56576
$1E	30	7680	$5E	94	24064	$9E	158	40448	$DE	222	56832
$1F	31	7936	$5F	95	24320	$9F	159	40704	$DF	223	57088
$20	32	8192	$60	96	24576	$A0	160	40960	$E0	224	57344
$21	33	8448	$61	97	24832	$A1	161	41216	$E1	225	57600
$22	34	8704	$62	98	25088	$A2	162	41472	$E2	226	57856
$23	35	8960	$63	99	25344	$A3	163	41728	$E3	227	58112
$24	36	9216	$64	100	25600	$A4	164	41984	$E4	228	58368
$25	37	9472	$65	101	25856	$A5	165	42240	$E5	229	58624
$26	38	9728	$66	102	26112	$A6	166	42496	$E6	230	58880
$27	39	9984	$67	103	26368	$A7	167	42752	$E7	231	59136
$28	40	10240	$68	104	26624	$A8	168	43008	$E8	232	59392
$29	41	10496	$69	105	26880	$A9	169	43264	$E9	233	59648
$2A	42	10752	$6A	106	27136	$AA	170	43520	$EA	234	59904
$2B	43	11008	$6B	107	27392	$AB	171	43776	$EB	235	60160
$2C	44	11264	$6C	108	27648	$AC	172	44032	$EC	236	60416
$2D	45	11520	$6D	109	27904	$AD	173	44288	$ED	237	60672
$2E	46	11776	$6E	110	28160	$AE	174	44544	$EE	238	60928
$2F	47	12032	$6F	111	28416	$AF	175	44800	$EF	239	61184
$30	48	12288	$70	112	28672	$B0	176	45056	$F0	240	61440
$31	49	12544	$71	113	28928	$B1	177	45312	$F1	241	61696
$32	50	12800	$72	114	29184	$B2	178	45568	$F2	242	61952
$33	51	13056	$73	115	29440	$B3	179	45824	$F3	243	62208
$34	52	13312	$74	116	29696	$B4	180	46080	$F4	244	62464
$35	53	13568	$75	117	29952	$B5	181	46336	$F5	245	62720
$36	54	13824	$76	118	30208	$B6	182	46592	$F6	246	62976
$37	55	14080	$77	119	30464	$B7	183	46848	$F7	247	63232
$38	56	14336	$78	120	30720	$B8	184	47104	$F8	248	63488
$39	57	14592	$79	121	30976	$B9	185	47360	$F9	249	63744
$3A	58	14848	$7A	122	31232	$BA	186	47616	$FA	250	64000
$3B	59	15104	$7B	123	31488	$BB	187	47872	$FB	251	64256
$3C	60	15360	$7C	124	31744	$BC	188	48128	$FC	252	64512
$3D	61	15616	$7D	125	32000	$BD	189	48384	$FD	253	64768
$3E	62	15872	$7E	126	32256	$BE	190	48640	$FE	254	65024
$3F	63	16128	$7F	127	32512	$BF	191	48896	$FF	255	65280

---------- OPCODE HIGH NYBBLE ----------

---------- OPCODE LOW NYBBLE ----------

	0	1	2	4	5	6	8	9	A	C	D	E
0	BRK	ORA (Ind,X)			ORA Zer	ASL Zer	PHP	ORA Imm	ASL A		ORA Abs	ASL Abs
1	BPL	ORA (Ind),Y			ORA Zer,X	ASL Zer,X	CLC	ORA Abs,Y			ORA Abs,X	ASL Abs,X
2	JSR	AND (Ind,X)		BIT Zer	AND Zer	ROL Zer	PLP	AND Imm	ROL A	BIT Abs	AND Abs	ROL Abs
3	BMI	AND (ind),Y			AND Zer,X	ROL Zer,X	SEC	AND Abs,Y			AND Abs,X	ROL Abs,X
4	RTI	EOR (Ind,X)			EOR Zer	LSR Zer	PHA	EOR Imm	LSR A	JMP Abs	EOR Abs	LSR Abs
5	BVC	EOR (Ind),Y			EOR Zer,X	LSR Zer,X	CLI	EOR Abs,Y			EOR Abs,X	LSR Abs,X
6	RTS	ADC (Ind,X)			ADC Zer	ROR Zer	PLA	ADC Imm	ROR A	JMP Ind	ADC Abs	ROR Abs
7	BVS	ADC (Ind),Y			ADC Zer,X	ROR Zer,X	SEI	ADC Abs,Y			ADC Abs,X	ROR Abs,X
8		STA (Ind,X)		STY Zer	STA Zer	STX Zer	DEY		TXA	STY Abs	STA Abs	STX Abs
9	BCC	STA (Ind),Y		STY Zer,X	STA Zer,X	STX Zer,Y	TYA	STA Abs,Y	TXS		STA Abs,X	
A	LDY Imm	LDA (Ind,X)	LDX Imm	LDY Zer	LDA Zer	LDX Zer	TAY	LDA Imm	TAX	LDY Abs	LDA Abs	LDX Abs
B	BCS	LDA (Ind),Y		LDY Zer,X	LDA Zer,X	LDX Zer,Y	CLV	LDA Abs,Y	TSX	LDY Abs,X	LDA Abs,X	LDX Abs,Y
C	CPY Imm	CMP (Ind,X)		CPY Zer	CMP Zer	DEC Zer	INY	CMP Imm	DEX	CPY Abs	CMP Abs	DEC Abs
D	BNE	CMP (Ind),Y			CMP Zer,X	DEC Zer,X	CLD	CMP Abs,Y			CMP Abs,X	DEC Abs,X
E	CPX Imm	SBC (Ind,X)		CPX Zer	SBC Zer	INC Zer	INX	SBC Imm	NOP	CPX Abs	SBC Abs	INC Abs
F	BEQ	SBC (Ind),Y			SBC Zer,X	INC Zer,X	SED	SBC Abs,Y			SBC Abs,X	INC Abs,X

TABLE OF 6502 OPCODES

EXAMPLES OF ADDRESSING MODES WITH THE 6502

Absolute	i.	CPX $12CF Compares the contents of the X-register with that of location $12CF. Both X and $12CF are unchanged, and the N,Z,and C flags are reset. The point is that the 'absolute address' is used.
	ii.	STA $8000 Stores the accumulator in location $8000.
	iii.	JSR WAIT Assembler notation for a subroutine call to 'WAIT', which the assembler identifies as a 2-byte address.
Absolute,X	i.	LDA $FF00,X The X register is treated as an offset; its value (0-255) is added to $FF00, and the accumulator loaded with the byte found at this new address. Thus, if X holds #$F5, in the example the accumulator will be loaded with the contents of $FFF5.
	ii.	ADC $7100,X Adds the contents of $7100, offset by X, plus the carry bit, to A. The result remains in A. This indexed instruction (like all such instructions) provides easy access to a range of addresses, in association with DEX, INX, and related commands.
Absolute,Y	i.	LDX TABLE,Y Absolute addressing indexed by Y is exactly analogous to X indexing, although fewer opcodes have this facility. In the example, X is loaded with the byte at TABLE+Y.
	ii.	STA $8000,Y Stores the accumulator into a location between $8000 and $80FF, depending on the current value held in Y.
Zero page	i.	LDA $70 Loads A with the contents of location $70.
	ii.	SBC $43 Subtracts the contents of $43 from A.
	iii.	ROL ZPG 128 Unusual assembler version of what normally appears as ROL $80, which rotates the contents of $80, and C, to the left.
Zero page,X	i.	INC $15,X Analogous to absolute addressing indexed by X, except for the use of the single-byte address to which the offset X is added. *BUT only* zero page addresses are generated: if X holds #$0A, then $15,X refers to location $1F. However, if X holds #$F0 then $15,X is address $05, *not* $0105.
Zero page,Y	i.	LDX $AB,Y Only two instructions can use this mode. Both involve
	ii.	STX $10,Y the X register. The operation of register Y on the zero page address is exactly similar to the previous example.
Implied	i.	BRK A large number of instructions do not operate on external
	ii.	CLC RAM or ROM, but on flags, registers, and the stack, which
	iii.	PHA are internal to the chip. Other examples: TXS, SEI, CLD, INY.
Immediate	i.	LDA '$ Assembler form of instruction to load A with ASCII $.
	ii.	LDA #$30 Loads A with hex 30. The third example is from a
	iii.	LDY IMM 48 decimal assembler which quotes the mode, rather than let it be deduced by the form of the instruction. Note that immediate mode is the only mode handling direct data values.
Relative	i.	BEQ $0294 Branches to $0294 if the zero flag is set; the next
	ii.	BCS L1 example is an assembler version, branching to a label.
	iii.	BVC +117 This last example shows a different convention, which corresponds to the way the opcode is stored. Here, there's an offset of 117 bytes forward from the next instruction.
Accumulator	i.	LSR A A few commands act on the contents of A, either rotating it
	ii.	ROL or shifting it. It can be considered an implied mode.
Indexed indirect	i.	ORA ($00,X) Displacement X is added to the zero page address, to give a new address in the zero page. This address, and its subsequent byte, together point to an absolute address which (in the example) is ORed with A. If there is a collection of pointers in the zero page, this is useful. When X holds zero, the mode becomes in effect straightforward indirect addressing. Example: X holds #5, location 5 holds #1, location 6 holds 128. LDA ($00,X) loads the accumulator from $8001.
Indirect indexed	i.	LDA ($12),Y The address in ($12), that is, having $13's contents as its high and $12's as its low byte, *plus* the offset within Y, is accessed and loaded into A. This is useful when dealing with RAM data arranged consecutively, e.g. messages and tables.
Absolute indirect	i.	JMP ($0090) Only JMP=6C hex uses this mode. In the example, if $90 holds #$2E and $91 holds #$E6, JMP($0090) jumps to $E62E.

6502 TIMING: QUICK REFERENCE CHART

ADDRESSING MODE	TIME ...	... EXCEPTIONS Dec, Inc, Rotate, Shift	Others
Absolute	4	6	JMP=3 JSR=6
Abs,X and Abs,Y	4 (+1 over page)	7	STA = 5
Zero Page	3	5	
Zer,X and Zer,Y	4	6	
Implied	2		Stack PH=3, PL=4 RTS=6 RTI=6
Immediate	2		BRK=7
Relative	2 (if no branch)/ 3 (if branch taken +1 over page)		
Accumulator	2		
(Ind,X)	6		
(Ind),Y	5 (+1 over page)		STA (Ind),Y =6
(Absolute)	5		

All figures are clock cycles (one millionth of a second for CBM computers)

PROCESSOR STATUS REGISTER

Example: A processor status register (SR with CBM's monitor) of 32 hexadecimal means that the Break flag (B) and the zero flag (Z) were set on entering the monitor. Some of the combinations may at first sight appear impossible; how can the negative bit (N) and the zero bit (Z) be simultaneously on? But the BIT opcode can accomplish this; and generally PHP can be used to set flags.

```
7 6 5 4 3 2 1 0
N V 1 B D I Z C
```

High nybble	
2	
3	B
6	V
7	V B
A	N
B	N B
E	NV
F	NV B

Low nybble	
0	
1	C
2	Z
3	ZC
4	I
5	I C
6	I Z
7	I ZC
8	D
9	D C
A	D Z
B	D ZC
C	D I
D	D I C
E	D IZ
F	D IZC

FURTHER ASPECTS OF THE 6502

[1] **ROR.** 6502s made before 1977 may not possess this command. Machine-code is therefore sometimes written without it if there are old 6502s in the field.'Byte' of June '81 has an article on an Atari BASIC with ROR replaced by equivalent code.

[2] **JMP.** There is a bug in the 6502's processing of the indirect jump ($6C) instruction. When the indirect address straddles a page boundary, the high byte is taken from the address in its own page. Thus, JMP ($03FF) jumps to the new address whose low byte is in $03FF, as it should be, but whose high byte comes from $0300, not the correct $0400. No CBM code contains a jump of this sort.*

[3] **Addressing modes.** Examination of tables of opcodes shows that there are many periodic patterns in the distribution of the codes. Given the way logic circuits work, this is not surprising. All those opcodes which have more than one addressing mode are dependent on bits 2,3, and 4 to determine the mode; the table following shows the relationships. If the addressing mode doesn't exist for an opcode, then that part of the table does not of course apply:

Opcode = xxxbbbxx   **Values of bbb represent:**

b	bb
0 Not post-indexed	00 (Indirect,X) 01 Zero Page 10 Immediate/ Accumulator
1 Post-indexed	00 (Indirect),Y *or* when followed by 00, Relative 01 Zero Page,X 10 Absolute,Y *or* when followed by 00, Implied 11 Absolute,X

[4] **Pseudo-Opcodes.** The sequential tables of opcodes and addressing modes omitted columns corresponding to -3, -7, -B, and -F. This is for the good reason that the manufacturers do not specify any function of the chip corresponding to these values. There are other gaps in the table: in fact, only 151 opcodes are implemented of the possible 256 (or more) maximum. Nevertheless, many pseudo-opcodes(for want of a better term)appear to exist, though the makers don't encourage correspondence on this point. IPUG (Jan '81) published an article by B Grainger on empirical work done on the 6502, in which descriptions accompanied by 3 letter 'opcodes' account for 93 of the 105 mystery values - if they are correct. For those who are interested in arcana of this sort, I present later the substance of his article with notes on timing and testing pseudo-opcodes and possible applications. In the absence of theoretical underpinning it is hard to know where to start on such an investigation; for one thing there is no guarantee that a non-standard code will not behave in entirely unexpected ways, corrupting registers perhaps, or executing repetitive, meaning-less loops, like 'Halt and catch fire' on the Z80.

Jim Butterfield has pointed out that codes ending in bits -11 simultaneously execute two commands, those ending in -01 and in -10. (Except that the timing may fail in some circumstances, transferring only 6 bits). All codes ending -3,-7, -B, and -F are of this type. This means that that a pseudo-opcode ending in -A, say, may combine the functions of the two codes ending -8 and -9 next to it. Something similar seems to happen in other cases not of this type. The X2 crash, for instance, in which an opcode ending -2 causes the chip to loop indefinitely until interrupted, appears to operate by virtue of the fact that all the branch commands end with -0, and are near neighbours. And the 'SKW' or skip word pseudo-opcode which skips over the next two bytes and 'SKB' which skips one byte are each found, respectively, in the absolute and zero page area of the opcode table, suggesting that some of these pseudo-codes may corrupt one or two memory locations.

---

*I have heard that there is a bug related to one of the registers, but know nothing else about this alleged malfunction.

6502 PSEUDO-OPCODES*

Instruction	Abs	Abs,X	Abs,Y	Zer	Zer,X	Zer,Y	(Ind,X)	(Ind),Y	Imm
ASO (ASL,ORA)	0F	1F	1B	07	17		03	13	0B
RLA (ROL,AND)	2F	3F	3B	27	37		23	33	2B
LSE (LSR,EOR)	4F	5F	5B	47	57		43	53	4B
RRA (ROR,ADC)	6F	7F	7B	67	77		63	73	6B
AXS (STX,STA)	8F			87		97	83		
LAX (LDX,LDA)	AF		BF	A7	B7		A3	B3	
DCM (DEC,CMP)	CF	DF	DB	C7	D7		C3	D3	
INS (INC,SBC)	EF	FF	FB	E7	F7		E3	F3	
ALR (LSR,EOR)									4B
ARR (ROR,ADC)									6B
XAA (TXA,)									8B
OAL (TAX,LDA)									AB
SAX (DEX,CMP)									CB
NOP	1A, 3A, 5A, 7A, DA, FA								
SKB	80, 82, C2, E2, 04, 14, 34, 44, 54, 64, 74, D4, F4								
SKW	0C, 1C, 3C, 5C, 7C, DC, FC								

The table shows some pseudo-opcodes. Those in bold type are more credible than those which are not. Probably, most anomalies will occur when addressing modes of neighbouring opcodes, or their timings, don't match. For instance, the code BF, combining LDX Abs,Y with LDA Abs,X might be expected to give an odd resulting addressing mode, particularly if one or both instructions crosses a page and so takes another clock cycle to execute. The 'mnemonics' given in the table have the following significance:

ASO   ASL then ORA the result with the accumulator
RLA   ROL then AND the result with the accumulator
LSE   LSR then EOR the result with the accumulator
RRA   ROR then ADC the result to the accumulator
AXS   Store the result of A AND X
LAX   LDA and LDX with the same data
DCM   DEC memory and CMP the result with the accumulator
INS   INC memory then SBC the result from the accumulator
ALR   AND the accumulator with data and LSR the result
ARR   AND the accumulator with data and ROR the result
XAA   Store X AND data in the accumulator
OAL   ORA the accumulator with #$EE, AND the result with data, then TAX
SAX   SBC data from A AND X and store result in X
NOP   No operation
SKB   Skip byte (ie branch of +1)
SKW   Skip word of 2 bytes (ie branch of +2)

Many of the codes show repetitiveness, derived from regularities in the 6502, but there are many one off possibilities too: 9B combines TXS and STA Abs,Y will the net effect of storing 'A AND X in the stack pointer and bit 0 of A AND X in memory'.

Applications. These commands are not part of the chip's specification, so they should best be avoided in machine-code routines for sale or general use. Sometimes they are helpful in debugging machine-code, when a jump or branch has been taken to a wrong address. Mainly, though, the potential application lies in the fact that hidden routines - to print out identification messages, or save a program using a special technique, for example - for the programmer's use only can be written in a way which disassemblers will be unable to decipher easily.

---

*Most of the mnemonics and descriptions are from B Grainger's Jan. '81 IPUG article.

SUPERMON LISTINGS

SUPERMON 1.

A version of Supermon for BASIC 1, by D A Hills, was published in CPUCN 2, #7. It is written for the 8K version only, and is not in relocatable form.

SUPERMON 2.

For the sake of variety I have written this monitor in the form of two BASIC programs on the theory that some people prefer BASIC to the machine-language monitor in the CBM. The long program, run first, puts machine-code into position for relocation by the following relocating loader:

```
155 POKE42,182:POKE43,6:CLR
160 L=PEEK(52)+PEEK(53)*256
170 N=L-1466:P=3391:FORJ=L-1TONSTEP-1
180 X=PEEK(P):IFX>0GOTO190
185 P=P-2:X=PEEK(P+1)+PEEK(P)*256:IFX=0GOTO190
186 X=X+L-65536:X%=X/256:X=X-X%*256:POKEJ,X%:J=J-1
190 POKEJ,X:P=P-1:PRINT"[HOME]";X;"[LEFT]  ":NEXTJ
200 X%=N/256:Y=N-X%*256:POKE52,Y:POKE53,X%:POKE48,Y:POKE49,X%
210 PRINT"[CLEAR][DOWN]LINK TO MONITOR -- SYS";N
220 PRINT:PRINT"SAVE WITH MLM:"
230 PRINT".S ";CHR$(34);"SUPERMON";CHR$(34);",01";:X=N/4096:GOSUB250
240 X=L/4096:GOSUB250:END
250 PRINT",";:FORJ=1TO4:X%=X:X=(X-X%)*16:IFX%>9THENX%=X%+7
260 PRINTCHR$(X%+48);:NEXTJ:RETURN
```

SUPERMON 4.

This monitor for BASIC 4 consists of a BASIC title program which also calls a machine-code routine to relocate SUPERMON to the current top-of-memory. SUPERMON is long, so the keying-in process is recommended only for those users who do not have access to this public-domain program. Type the BASIC program in the usual way (note that the spacing of the commands is not critical. Lines 170-220 do the work). Then enter the monitor (e.g. with SYS 4) and enter the machine-code. *SAVE the code before you try it out - a single mistake may cause the program to hang up.* In order to save the machine-code with BASIC, change the end-of-program pointers to include the entire machine-code program, with   .M 002A 002A

           .: 002A F0 0C xx xx xx xx xx xx

Where the xx's are unchanged.

```
100 PRINT" SUPERMON4!"
110 PRINT"      DISSASSEMBLER D BY WOZNIAK/BAUM
120 PRINT"         SINGLE STEP I BY JIM RUSSO
130 PRINT"MOST OTHER STUFF ,HAFT BY BILL SEILER
150 PRINT"TIDIED & WRAPPED BY JIM BUTTERFIELD"
170 L=PEEK(52)+PEEK(53)*256:SYS1536:M=PEEK(33):N=PEEK(34)
180 POKE52,M:POKE53,N:POKE48,M:POKE49,N:N=M+N*256
210 PRINT"LINK TO MONITOR -- SYS";N
220 SYS N
```

M/code loader.
↓

```
0600 A9 CB 85 1F A9 0C 85 20
0608 A5 34 85 21 A5 35 85 22
0610 A0 00 20 38 06 D0 16 20
0618 38 06 F0 11 85 23 20 38
0620 06 18 65 34 AA A5 23 65
0628 35 20 43 06 8A 20 43 06
0630 20 50 06 90 DB 60 EA EA
0638 A5 1F D0 02 C6 20 C6 1F
0640 B1 1F 60 48 A5 21 D0 02
0648 C6 22 C6 21 68 91 21 60
0650 A9 80 C5 1F A9 06 E5 20
0658 60 AA AA AA AA AA AA AA
```

Start of SUPERMON 4
Data [contd. overleaf]. →

```
0680 AD FE FF 00 85 34 AD FF
0688 FF 00 85 35 AD FC FF 00
0690 8D FA 03 AD FD FF 00 8D
0698 FB 03 00 00 A2 08 DD DE
06A0 FF 00 D0 0E 86 B4 8A 0A
06A8 AA BD E9 FF 00 48 BD E8
06B0 FF 00 48 60 CA 10 EA 4C
06B8 9A FA 00 A2 02 2C A2 00
06C0 00 B4 FB D0 08 B4 FC D0
06C8 02 E6 DE D6 FC D6 FB 60
06D0 20 98 D7 C9 20 F0 F9 60
06D8 A9 00 00 8D 00 00 01 20
06E0 79 FA 00 20 6B D7 20 57
06E8 D7 90 09 60 20 98 D7 20
06F0 54 D7 B0 DE AE 06 02 9A
06F8 4C A4 D7 20 31 D5 CA D0
0700 FA 60 E6 FD D0 02 E6 FE
```

```
708 60 A2 02 B5 FA 48 BD 0A CA 0A 2C
710 02 95 FA 68 9D 0A CA 0C 20
718 00 F1 60 AD 0B 02 AC 0C 02
720 02 4C CE FA 20 A5 FD A4 00
728 FE 38 E5 A8 0D 1B 02 98 20
730 FC A8 CE FB 8D 02 60 20
738 81 0D 1B 02 A5 FC 02 20
740 FA 00 FA 00 44 D7 20 92
748 FA 00 FA 00 D7 20 92 FA
750 D7 90 15 A6 CA FA 00 65
758 C1 FA 00 20 20 00 FD
760 D7 20 EB 20 C1 FA 00 39
768 FD 20 A8 FA 90 00 20 AD
770 1B 02 00 20 65 FD 20 81
778 FE 85 FD 02 AF FA 98 65
780 DE 00 3D A1 FB 81 20 A6
788 D0 11 20 00 20 34 00 65
790 00 20 81 FA 00 B0 FA AD
798 00 92 20 FA 00 4C 1B D5
7A0 20 20 68 00 00 44 20 81
7A8 98 D7 20 63 00 20 1B FB
7B0 B5 A6 D7 A2 00 14 00 20
7B8 00 0C DE 20 D0 00 98 20
7C0 39 D5 4C 98 C9 27 D7 20
7C8 4C BA D4 20 E0 D0 10 02
7D0 44 D7 20 92 FA 00 10 00
7D8 4C 20 98 D7 20 00 00 EC
7E0 98 D7 C9 27 D0 14 20 86
7E8 D7 20 C9 20 E8 20 00 63
7F0 C9 0D F0 10 02 CF FF 20
7F8 1C 8E 00 00 01 20 D0 F1
800 D7 90 C6 9D 00 00 20 6B
808 CF FF C9 0D 10 02 E8 20
810 D7 20 34 B1 FB DD A2 00
818 B4 20 B6 E0 D5 00 00 00
820 00 C8 00 00 E4 B4 D0 F3
828 0C 00 00 31 D5 20 39 D5
830 17 D7 20 20 CA 20 00 00
838 A6 DE DE 4C 92 20 CA FA
840 00 8D 0D 2C BA D4 20 00
848 B0 DE DE A5 FC A5 20 81
850 DE 4C 92 20 2C BA D4 FA
858 02 2E 0A A2 02 A9 00 85
860 FF A9 0A 85 B5 20 06 FC
868 00 2E 64 FC 00 85 20 D2
870 FC 20 A9 16 0A A2 09 91
878 D2 FF 4C BA D4 A0 2C 20

880 79 A2 0A 48 20 17 D7 20 31 D5 9F8
888 88 A2 00 00 00 FB 20 FC 20 FC
890 00 48 20 BB FC 20 20 20 20
898 D3 00 03 FC 06 E0 03 68 20
8A0 13 AC 1C 00 A2 E0 A5 74 FC
8A8 C9 1C 02 A1 FB 00 5C 90 5C
8B0 00 20 06 FB 20 06 FF 5C FD
8B8 BD 51 FD 00 F2 06 90 90 90
8C0 00 88 B0 45 FF B0 1C 20 20
8C8 0E BD 00 20 06 20 45 FD
8D0 BD FC 57 FD FF 00 FF FD 00
8D8 D0 00 00 FF 20 03 20 00 02
8E0 A2 79 00 0B 4A AA 60 8A 29
8E8 20 A9 00 48 4A 4A 4A 4A 20
8F0 4A 00 13 29 0F 09 30 C9 3A
8F8 FF 0A 00 00 00 85 FB 20

9F8 9F8 91 85 A9 A0 8D 4E E8 CE
A00 13 E8 2E 8D 48 A9 FC 00
A08 E8 00 8D 49 2E A9 06 20
A10 9A 4C 00 C0 FC 68 20 49
A18 18 8D 05 02 68 8D 04 02
A20 8D A9 2E 68 8D 03 02 68
A28 A9 20 02 8D 02 02 68 8D
A30 BA 8D 01 02 BA 8E 00 02
A38 20 02 68 8D FD 01 58 20
A40 20 23 34 00 00 00 1A 1A
A48 AE 06 15 9C 9C 00 00 7C
A50 58 84 13 34 11 A5 69 00
A58 FB 03 C9 F7 20 79 44 4C
A60 D0 35 F3 17 20 44 44 33
A68 35 20 85 D7 20 44 44 33
A70 81 FA D7 FC AD 00 22 D0
A78 FF F0 A0 01 C8 48 A9 00
A80 4A 4A 4A 4A 20 86 44 4A
A88 AA BD 1C 03 8D 1C 59 00
A90 90 17 C8 AA A9 10 22 00
A98 98 80 29 0F 09 BD 44 00
AA0 03 7F 0F 30 44 33 00
AA8 AA B4 60 8A 10 60 08
AB0 BD 10 8A D4 20 8A 44
AB8 D0 C8 F2 20 4A 72 00
AC0 04 86 04 20 BD 00 86
AC8 8E 02 0B E8 00 04 0B
AD0 F0 9D 10 D7 A4 53 AE
AD8 20 A2 00 78 A2 19 19
AE0 60 03 8A EB D7 1E 24
AE8 00 05 A0 00 00 00 00
AF0 86 FF 9E 8D 00 A1 00
AF8 03 D0 0E 8A 8A A1 00

B70 88 D0 F8 A5 DE 91 FB 20
B78 64 FC 00 85 FC A0 A0
B80 41 20 79 D5 20 17 D7
B88 31 D5 4C 98 DD 11 98
B90 FE 00 FD F0 02 FD F0
B98 86 A6 B5 DD 10 98 DD
BA0 E8 B4 A6 B5 03 C9 47
BA8 86 30 A6 B5 C9 60 38
BB0 40 90 03 D0 22 45 60
BB8 30 86 B5 45 33 C9 28
BC0 40 22 45 45 33 C8 1A
BC8 30 02 45 45 33 C4 CA
BD0 40 45 45 33 D0 08 1A
BD8 22 B3 33 D0 08 08 26
BE0 40 30 D0 08 08 08 59
BE8 09 78 A9 13 78 A9 00
BF0 40 82 81 92 86 4A 00
BF8 21 40 09 82 82 24 53
C00 29 4D 58 24 28 24 53
C08 10 D8 88 23 28 19 24
C10 19 AE 69 1D 86 1D AE
C18 23 9D 88 8B 8B A1 00
C20 23 92 82 86 19 23 00
C28 19 AE 69 8B 19 24 53
C30 1B 24 53 19 24 72 F2
C38 1A 5B 58 A5 24 A2 74
C40 AE 00 29 00 7C AE 00
C48 00 15 9C 9C 6D 29
C50 53 84 13 34 11 A5 23
C58 A0 D8 62 48 26 5A 94
C60 88 54 C8 68 68 84 E8
C68 94 00 00 44 CC 72 B4
C70 D5 20 4C D8 11 FD F0
C78 A4 28 6E 8A A2 A2 F2
C80 DB 00 74 74 00 68 32 B2
C88 FD 26 74 74 44 AA B2
C90 00 48 44 A2 C8 1A 26
C98 44 50 2C 41 49 4E 48
CA0 DF 00 0A 20 A2 C8 CA
CA8 C5 88 DF D7 AC 02 C5
CB0 B5 D0 20 20 20 FC 46
CB8 20 BC 57 FF EB BD 48
CC0 88 D0 06 00 3A B9 FD
CC8 09 20 00 00 00 33 00
CD0 CA FA D4 7F 4A 4D 00
CD8 CD 10 02 00 00 00 00
CE0 AA AA AA AA AA 00 00
```

BASIC 2 SUPERMON DATA LOADER.

Note: M 0028 0028 may give something like: 26 7A 03 04 03 04 03 04, where the start
address is inconsistent. For compatibility with BASIC change 26 7A to 01 04.

```
100 PRINT "[REVS]SETTING UP DATA FOR SUPERMON..."
110 FOR J = 1767 TO 3391: READ X: POKE J+5500,X: NEXT
120 FOR J = 1767 TO 3391: POKE J, PEEK (J+5500): NEXT
130 PRINT"[DOWN][REVS]NOW RUN SUPERMON BASIC LOADER":END
500 DATA 173,255,254,0,133,52,173,255,255,0,133,53,173,255,252,0,141,250,3,173,255
501 DATA 253,0,141,251,3,0,0,0,162,8,221,255,222,0,208,14,134,180,138,10,170,189
502 DATA 255,233,0,72,189,255,232,0,72,96,202,16,234,76,247,231,162,2,44,162,0,0
503 DATA 0,180,251,208,8,180,252,208,2,230,222,214,252,214,251,96,32,235,231,201
504 DATA 32,240,249,96,169,0,0,0,141,0,0,1,32,250,140,0,32,190,231,32,170,231,144
505 DATA 9,96,32,235,231,32,167,231,176,222,76,247,231,32,205,253,202,208,250,96
506 DATA 230,253,208,2,230,254,96,162,2,181,250,72,189,10,2,149,250,104,157,10,2
507 DATA 202,208,241,96,173,11,2,172,12,2,76,250,221,0,165,253,164,254,56,229,251
508 DATA 133,207,152,229,252,168,5,207,96,32,250,148,0,32,151,231,32,250,165,0,32
509 DATA 250,190,0,32,250,165,0,32,250,217,0,32,151,231,144,21,166,222,208,100,32
510 DATA 250,208,0,144,95,161,251,129,253,32,250,183,0,32,213,253,208,235,32,250
511 DATA 208,0,24,165,207,101,253,133,253,152,101,254,133,254,32,250,190,0,166,222
512 DATA 208,61,161,251,129,253,32,250,208,0,176,52,32,250,120,0,32,250,123,0,76
513 DATA 251,39,0,32,250,148,0,32,151,231,32,250,165,0,32,151,231,32,235,231,32,182
514 DATA 231,144,20,133,181,166,222,208,17,32,250,217,0,144,12,165,181,129,251,32
515 DATA 213,253,208,238,76,247,231,76,86,253,32,250,148,0,32,151,231,32,250,165
516 DATA 0,32,151,231,32,235,231,162,0,0,0,32,235,231,201,39,208,20,32,235,231,157
517 DATA 16,2,232,32,207,255,201,13,240,34,224,32,208,241,240,28,142,0,0,0,1,32,190
518 DATA 231,144,198,157,16,2,232,32,207,255,201,13,240,9,32,182,231,144,182,224
519 DATA 32,208,236,134,180,32,208,253,162,0,0,0,160,0,0,0,177,251,221,16,2,208,12
520 DATA 200,232,228,180,208,243,32,106,231,32,205,253,32,213,253,166,222,208,146
521 DATA 32,250,217,0,176,221,76,86,253,32,250,148,0,141,13,2,165,252,141,14,2,169
522 DATA 4,162,0,0,0,133,184,134,185,169,147,32,210,255,169,22,133,181,32,252,16
523 DATA 0,32,252,109,0,133,251,132,252,198,181,208,242,169,145,32,210,255,76,86
524 DATA 253,160,44,32,21,254,32,106,231,32,205,253,162,0,0,0,161,251,32,252,124
525 DATA 0,72,32,252,194,0,104,32,252,216,0,162,6,224,3,208,18,164,182,240,14,165
526 DATA 255,201,232,177,251,176,28,32,252,101,0,136,208,242,6,255,144,14,189,255
527 DATA 74,0,32,253,77,0,189,255,80,0,240,3,32,253,77,0,202,208,213,96,32,252,112
528 DATA 0,170,232,208,1,200,152,32,252,101,0,138,134,180,32,117,231,166,180,96,165
529 DATA 182,56,164,252,170,16,1,136,101,251,144,1,200,96,168,74,144,11,74,176,23
530 DATA 201,34,240,19,41,7,9,128,74,170,189,254,249,0,176,4,74,74,74,74,41,15,208
531 DATA 4,160,128,169,0,0,0,170,189,255,61,0,133,255,41,3,133,182,152,41,143,170
532 DATA 152,160,3,224,138,240,11,74,144,8,74,74,9,32,136,208,250,200,136,208,242
533 DATA 96,177,251,32,252,101,0,162,1,32,250,176,0,196,182,200,144,241,162,3,196
534 DATA 184,144,242,96,168,185,255,87,0,141,11,2,185,255,151,0,141,12,2,169,0,0
535 DATA 0,160,5,14,12,2,46,11,2,42,136,208,246,105,63,32,210,255,202,208,234,76
536 DATA 205,253,32,250,148,0,32,213,253,32,213,253,32,151,231,32,250,165,0,32,151
537 DATA 231,32,202,253,32,250,217,0,144,9,152,208,19,165,207,48,15,16,7,200,208
538 DATA 10,165,207,16,6,32,117,231,76,86,253,76,247,231,32,250,148,0,169,3,133,181
539 DATA 32,235,231,32,167,253,208,248,173,13,2,133,251,173,14,2,133,252,76,251,241
540 DATA 0,197,185,240,3,32,210,255,96,169,3,162,36,133,184,134,185,32,208,253,120
541 DATA 173,255,250,0,133,144,173,255,251,0,133,145,169,160,141,78,232,206,19,232
542 DATA 169,46,141,72,232,169,0,0,0,141,73,232,174,6,2,154,76,241,254,32,123,252
543 DATA 104,141,5,2,104,141,4,2,104,141,3,2,104,141,2,2,104,141,1,2,104,141,0,0
544 DATA 0,2,186,142,6,2,88,32,208,253,32,191,253,133,181,160,0,0,32,154,253,32
545 DATA 205,253,173,0,0,0,2,133,252,173,1,2,133,251,32,106,231,32,252,24,0,32,1
546 DATA 243,201,247,240,249,32,1,243,208,3,76,86,253,201,255,240,244,76,253,96,0
547 DATA 0,0,0,32,250,148,0,32,151,231,142,17,2,162,3,32,250,140,0,72,202,208,249
548 DATA 162,3,104,56,233,63,160,5,74,110,17,2,110,16,2,136,208,246,202,208,237,162
549 DATA 2,32,207,255,201,13,240,30,201,32,240,245,32,254,240,0,176,15,32,203,231
550 DATA 164,251,132,252,133,251,169,48,157,16,2,232,157,16,2,232,208,219,142,11
551 DATA 2,162,0,0,0,134,222,162,0,0,0,134,181,165,222,32,252,124,0,166,255,142,12
552 DATA 2,170,189,255,151,0,32,254,213,0,189,255,87,0,32,254,213,0,162,6,224,3,208
553 DATA 18,164,182,240,14,165,255,201,232,169,48,176,29,32,254,210,0,136,208,242
554 DATA 6,255,144,14,189,255,74,0,32,254,213,0,189,255,80,0,240,3,32,254,213,0,202
555 DATA 208,213,240,6,32,254,210,0,32,254,210,0,173,11,2,197,181,208,89,32,151,231
556 DATA 164,182,240,43,173,12,2,201,157,208,28,32,250,217,0,144,9,152,208,74,166
557 DATA 207,48,70,16,7,200,208,65,166,207,16,61,202,202,138,164,182,208,3,185,252
558 DATA 0,0,0,145,251,136,208,248,165,222,145,251,32,252,109,0,133,251,132,252,160
559 DATA 65,32,21,254,32,106,231,32,205,253,76,253,222,0,32,254,213,0,134,180,166
560 DATA 181,221,16,2,240,12,104,104,230,222,240,3,76,247,231,232,134
561 DATA 181,166,180,96,201,48,144,3,201,71,96,56,96,64,2,69,3,208,8,64,9,48,34,69
562 DATA 51,208,8,64,9,64,2,69,51,208,8,64,9,64,2,69,179,208,8,64,9,0,0,0,34,68,51
563 DATA 208,140,68,0,0,0,17,34,68,51,208,140,68,154,16,34,68,51,208,8,64,9,16,34
564 DATA 68,51,208,8,64,9,98,19,120,169,0,0,0,33,129,130,0,0,0,0,89,77,145,146
565 DATA 134,74,133,157,44,41,44,35,40,36,89,0,0,0,88,36,36,0,0,0,28,138,28,35,93
566 DATA 139,27,161,157,138,29,35,157,139,29,161,0,0,0,41,25,174,105,168,25,55,36
567 DATA 83,27,35,36,83,25,161,0,0,0,26,91,91,165,105,36,36,174,174,168,173,41,0
568 DATA 0,0,124,0,0,0,21,156,109,156,165,105,41,83,132,19,52,17,165,105,35,160,216
569 DATA 98,90,72,38,98,148,136,84,68,200,84,104,68,232,148,0,0,0,180,8,132,116,180
570 DATA 40,110,116,244,204,74,114,242,164,138,0,0,0,162,162,116,116,116,114
571 DATA 68,104,178,50,178,0,0,0,34,0,0,0,26,26,38,38,114,114,136,200,196,202,38
572 DATA 72,68,68,162,200,4,34,16,32,45,47,51,84,70,72,68,67,44,65,73,78,0,0,0,250
573 DATA 232,0,251,60,0,251,106,0,251,221,0,252,253,0,253,48,0,253,218,0,253,84,0
574 DATA 85,253,253,132,0,250,93,0,250,70,0
```

ASCII CODE

0	00	NUL	– NULL CHARACTER	32	20	SPACE	64	40	@	96	60	`
1	01	SOH	– START HEADING	33	21	!	65	41	A	97	61	a
2	02	STX	– START TEXT	34	22	"	66	42	B	98	62	b
3	03	ETX	– END TEXT	35	23	#	67	43	C	99	63	c
4	04	EOT	– END TRANSMISSION	36	24	$	68	44	D	100	64	d
5	05	ENQ	– ENQUIRY	37	25	%	69	45	E	101	65	e
6	06	ACK	– ACKNOWLEDGE	38	26	&	70	46	F	102	66	f
7	07	BEL	– RING BELL	39	27	'	71	47	G	103	67	g
8	08	BS	– BACKSPACE	40	28	(	72	48	H	104	68	h
9	09	HT	– HORIZONTAL TABULATION	41	29	)	73	49	I	105	69	i
10	0A	LF	– LINE FEED	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	– VERTICAL TABULATION	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	– FORM FEED	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	– CARRIAGE RETURN	45	2D	–	77	4D	M	109	6D	m
14	0E	SO	– SHIFT OUT	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	– SHIFT IN	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	– DATA LINK ESCAPE	48	30	0	80	50	P	112	70	p
17	11	DC1	– DEVICE CONTROL #1	49	31	1	81	51	Q	113	71	q
18	12	DC2	– DEVICE CONTROL #2	50	32	2	82	52	R	114	72	r
19	13	DC3	– DEVICE CONTROL #3	51	33	3	83	53	S	115	73	s
20	14	DC4	– DEVICE CONTROL #4	52	34	4	84	54	T	116	74	t
21	15	NAK	– NEGATIVE ACKNOWLEDGE	53	35	5	85	55	U	117	75	u
22	16	SYN	– SYNCHRONOUS IDLE	54	36	6	86	56	V	118	76	v
23	17	ETB	– END TRANSMISSION BLOCK	55	37	7	87	57	W	119	77	w
24	18	CAN	– CANCEL	56	38	8	88	58	X	120	78	x
25	19	EM	– END MEDIUM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	– SUBSTITUTE	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	– ESCAPE	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	– FILE SEPARATOR	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	– GROUP SEPARATOR	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	– RECORD SEPARATOR	62	3E	>	94	5E	↑	126	7E	~
31	1F	US	– UNIT SEPARATOR	63	3F	?	95	5F	—	127	7F	DEL

ASCII characters. The American Standard Code on Information Interchange (ASCII) is largely followed by CBM equipment. One major difference is its use of the high bit as a parity bit, making the number of '1's in the complete byte even. CBM's version of 'ASCII' has no parity bit.

BASIC is by far the most popular microcomputer language at present. It is impossible to know whether this will remain true, but brief comments on currently available alternatives follow. Many interesting languages have turned out to be comparative failures, usually for reasons of commercial pressure; ALGOL, FOCAL and PL/1 illustrate the pattern, effectively being dominated by FORTRAN, BASIC, and COBOL respectively. Although there is no reason why the entire ROM set of a CBM couldn't be replaced by a new language in ROM, in practice most replacements use RAM and also make use of BASIC subroutines, which (a) reduces the maximum size of program, and (b) tends to make the language non-transferrable between versions of BASIC.

COMAL is 24K of machine-code, which is loaded from disk into CBM RAM. Error messages are read from disk, so this language is practicably of use only with a CBM unit with disk drives. It is a public domain program; versions for BASIC 4 and (perhaps) BASIC 2 exist. This language was developed in Denmark, where it is reported to be in widespread use in education. It permits long variable names (i.e. not distinguished by the two initial characters only) and has a number of structured features, for example an IF ELSE construction of this form: IF .. THEN .. ELIF .. THEN .. ELIF .. THEN .. ENDIF where ELIF means ELSE IF. It also formats its listings as the short example program (right) illustrates. The CBM version is interpreted and is not particularly fast. Its principal purpose is to make fairly simple programs readable: it is

```
0010 FOR J:=1 TO 10 DO
0020   PRINT J
0030 NEXT J
0040 END
```

therefore quite suitable for examiners who wish to mark large numbers of beginners' programs. The name stands for 'COmmon ALgorithmic language'.

COMPILERS which turn BASIC into machine-code are now being marketed. (As an introduction, see M Zimmermann's article on 'Floptran', in BYTE, Oct.'80, and a followup article in July '81). BASIC 'source code' is converted into 'object code' by the compiler, which is in RAM. The result might be stored on disk, and LIST as SYS1037, which when RUN executes machine-code exactly similar to the original BASIC. Speed increases of about ten times are commonly claimed; the improvement occurs because much of BASIC's housekeeping, notably of variables, is eliminated. Typically, a line or so of BASIC is converted per second when the compiler runs. The resulting code is much more difficult to modify than BASIC, which provides some security in the case of commercial programs. Usually the code has a standard library of routines, occupying 4K say; together with the program, which is shorter than its BASIC form by a factor of about a half. A longish program may therefore be stored as a shorter amount of equivalent machine-code, especially if it contains many REMs, which the compiler ignores. There may be problems however: check up on (a) the maximum size of BASIC which the program can handle; it may not be enough for your programs. (b) The compatibility with ordinary BASIC: the compiler may not allow integers, or arrays, or variable-length strings. It may not cope with commands using wedges, and these may be important. It may not stop when the stop key is pressed, or print accurate error messages.

PASCAL (named after Blaise Pascal) is an academic language, which seems to have remained quite unpopular. Tiny Pascals (i.e. small implementations, lacking many of the features of the full-scale language) can be bought comparatively cheaply, i.e. for 30 to 40 pounds or dollars.

PILOT is a language used for educational programs of the question-and-answer type as the demonstration program attempts to illustrate. ('T' is text, 'A' inputs answer, 'M' searches for a match, and so on). It is a relatively easy language to write (and makes an interesting exercise). The pro-

```
*RETRY T: WHAT IS THE CAPITAL OF FRANCE?
       A: NAME$
       M: PARIS
       Y: CORRECT!
       N: NO... TRY AGAIN
      JN: *RETRY
```

cess of matching is usually unsophisticated, which, in view of the difficulties, is not surprising. A reply of 'NOT PARIS' or 'LONDON OR PARIS' would probably be judged correct when running the sample short program. Nevertheless PILOT is easy to use, and quick to produce results. The chief drawbacks may be (a) RAM space is likely to be used rapidly by the verbose style of such programs, so disk or tape loading is a useful feature of the language. (b) If interpreted, it may be slow.

PROGRAM GENERATORS at present can write BASIC of a simple file-handling sort. Knowledge of the behaviour of files, and systems analysis, are both required, to avoid logical errors. Screen and output formatting are not available.

address memory location (from 0-65535)

addressing mode one of several ways in which machine-code processes its data

algorithm series of unambiguous rules

ASCII American Standard Code on Information Interchange

baud rate rate of data transmission

batch processing system where preparation of data is separate from processing

binary expressed in two forms only

buffer RAM locations used for temporary storage before processing

bug mistake in software

chain call one program from another

chip integrated circuit, often on silicon

clock crystal oscillator timing the MPU

cold start start of a program from scratch with all variables zero or null

compiler program to convert a high-level language (e.g. BASIC) into exactly equivalent machine-code

complement reverse bits of bytes

conditional dependent on a result

crash unwanted program stop

crlf carriage return and line feed

diagnostic software or hardware which tests an aspect of system functioning

diskette circular magnetic-coated disk in a protective envelope for data storage

dump complete list, usually on paper, of a system aspect, for examination

enhancements improvements to a system

EAROM, EPROM electrically alterable/ erasable-programmable read-only memory

floating-point storage system for numbers using scientific notation, i.e. an exponent system, rather than 'fixed point'

function routine which performs conversions according to some formula

garbage RAM data or pointers, etc., left from previous processing

graphics any pictorial or diagrammatic output from a computer

hardcopy output by a printer on paper

hardware computer machinery

hashtotal meaningless but repeatable total providing check on data accuracy

hexadecimal number notation with base 16, using 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

high-level language any language needing processing to be usable by the computer, especially one with powerful instructions

infinite loop loop with no exit

integer whole number

interactive processing system where data is continually updated on file

interface hardware connections between devices

interrupt temporary program stop to process other data

iteration solution by progressive approximation with repetition

jumper wire connecting two parts of a printed-circuit board (noun or verb).

K 1024 (or, loosely, 1000)

link join two or more programs into one

listing readable form of a program

literal defined sequence of characters, not a variable

LSB least significant byte (i.e. of two)

MHz megahertz = million cycles per second

machine-code program in a form which the computer can directly run

MSB most significant byte (i.e. of two)

mode type of processing a computer is currently engaged on; e.g. edit mode

monitor (i) TV-like display hardware, (ii) software for machine-code work

MPU micro version of central processing unit (CPU); in the PET/CBM, its 6502

octal number notation using base 8

opcode operation code; mnemonic for each machine-code command, e.g. LDA

package standard software

parallel data bits sent simultaneously

peripheral device attached to the computer (noun)

pointer location(s) holding address of data, variables, RAM limits, etc

poke put a value into RAM

PROM programmable read-only memory

RAM random-access memory; may be read from, or written to, freely

recursion technique in which a program calls itself. See 'recursion'.

return (i) carriage return, (ii) end of a subroutine

ROM read-only memory; can be read from, but not written to

serial data bits sent consecutively

software programs held within hardware

string set of consecutive characters as processed by a computer as a unit, without (e.g.) attempting calculations

syntax rules for correct operation of a high-level computer language

teletype (trade mark) early model of computer printer with keyboard

terminal more-or-less remote connection to a computer; usually keyboard & screen

token single byte coded form of keyword

translater, -or program which runs a language by scanning it as it is stored, without pre-processing

transparent usable without being noticed

TTL transistor-transistor logic (not chip)

validation checking that data is possibly correct, i.e. not nonsense

VDU visual display unit

warm start restart of program, retaining some or all previous results

wedge interposed piece of software

word unit of processing of a computer; identical to the 8-bit byte in the CBM

ADDENDUM

The **FAT-40** *(twelve inch screen, 40-column CBM) is somewhat underrepresented in this book. Its BASIC, sometimes called BASIC 4.41 to distinguish it from the eight inch screen's BASIC 4.40, is similar to other BASIC 4s except for the EOOO ROM, which mainly affects keyboard and screen facilities. The jump tables on p.424 in Chapter 15 have some different destinations, for example, the HTAB and VTAB use SYS 57457. Also a FAT-40 EOOO ROM or EPROM in an eight inch screen computer with BASIC 4 equips the machine with FAT-40 facilities such as repeat keys, but not of course those associated with the CRT controller or bell.*

The **8096** *is an 8032 fitted with 32K extra RAM. Briefly, 32K is added to the top half of memory, from $8000 to $FFFF. Each of the two sets of 32K in high memory is divided into two sets of 16K, so that at any time two of the 16K blocks are in use, making four permutations. So for example a BASIC program might reside in low memory, with BASIC itself occupying the high end of memory. The extra memory can be accessed by poking location $FFFO; bits 2 and 3 select the current pair of 16K blocks which are to be active. BASIC itself will be temporarily lost, so SYS commands and machine-code have to be used, and the technique is not elementary. Machine-code is therefore usually used, as with VisiCalc; however, since BASIC can be soft-loaded (i.e. held in RAM, not ROM) any version of Commodore BASIC, including user-modified versions, can be run, if the screen is configured to the correct column width by the CRT chip. Alternative BASIC's are available on disk, including some for 50 and 60Hz displays, and also including BASIC 5. The effect is similar, but more far-reaching, to that obtained by relocating ROM routines into RAM as outlined in Chapter 13.*

The **'Super-PET'** *or series 9000 has 64K extra RAM, in 16 blocks all stored in $9000 to $9FFF and selectable by software. The machine has various hardware modifications and several languages, all developed at Waterloo University in Canada. At the time of writing, translated versions of BASIC, COBOL, Fortan, Pascal and APL have been written. The computer was partly intended for computer programming students.*

Commodore have tried a number of experiments with cheaper computers, notably the **VIC-20** *and* **VIC-40**, *which have 23 and 40 column screen outputs respectively. The evolution of these products has not been free of problems. New features include the RS232 interface, the custom video chips, high resolution graphics using RAM, and eight extra keys generating CHR$ (133) to CHR$ (140). The* **Commodore 64** *uses a new chip, the 6509, and is likely to be an improvement on its predecessors.*

Notes on the text   *Some of the periodicals mentioned are no longer published; and there is an inevitable tendency for hardware to be improved or superseded.*
Typographical errors and omissions include:
p.7:   RESTORE is abbreviated by RE shift-S.
p.14:   S=O: E=O before DIM keeps the array pointers correct.
p.58:   DIM sets up the variable or variables following DIM so that DIM, A,B %,CS for example sets A, B %, and CS in that order in RAM.
p.126:   CHR$ (18), not CHR$ (8), produces a reversed program name.
p.137:   The POKE location is 3, 4, or 16 depending on the version of BASIC in use.
p.250:   The lines are horizontal, not vertical. The diagram omits dots produced by the special character when values of 128 are included.
p.265:   "S" or "s" homes the cursor; the character as printed clears the screen.
p.346:   BASIC 4's keyboard interrupt doesn't work correctly when the decimal flag is set, so SEI/ SED/ perform processing/ CLD/ CLI may by necessary.

With direct access to Commodore disks, early versions of disk ROM will not set the buffer pointer to zero, so B-P may give errors because of this. Note that B-W stores the buffer pointer in byte zero before writing the block to disk, which will wreck the chaining of disk files if it is unexpected. B-R sets up end-of-file detection based on the same pointer in byte zero. For these reasons, U2 and U1 are preferred.
LOCKSMITH (p.177), renamed LOCKDISK, is a program by Jim Butterfield.

INDEX