Copyright © Acorn Computers Limited 1987
All rights reserved

This BSC implementation of C was written and designed by
Tim Bell, David Christensen, James Mansion and Jim
Warwick.


British Broadcasting Corporation has been abbreviated to
BBC in this publication.

No part of this book may be reproduced by any means without
the prior permission of the copyright holder. The only
exceptions are as provided for by the Copyright
(photocopying) Act or for the purpose of review or in order for
the software herein to be entered into a computer for the sole
use of the owner of this book.

FIRST EDITION
ISBN 1 85250 037 9
Part no 0479,544
Published by Acorn Computers Limited, Fulbourn Road,
Cherry Hinton, Cambridge CB1 4JN.

<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>i</td>
</tr>
<tr>
<td>THIS MANUAL</td>
<td>i</td>
</tr>
<tr>
<td>CONVENTIONS USED</td>
<td>ii</td>
</tr>
<tr>
<td>THE C LANGUAGE</td>
<td>1</td>
</tr>
<tr>
<td>RESTRICTIONS</td>
<td>2</td>
</tr>
<tr>
<td>EXTENSIONS</td>
<td>4</td>
</tr>
<tr>
<td>C SYSTEM OVERVIEW</td>
<td>7</td>
</tr>
<tr>
<td>THE C SYSTEM COMMAND LINE INTERPRETER</td>
<td>7</td>
</tr>
<tr>
<td>PRODUCING AND RUNNING A C PROGRAM</td>
<td>10</td>
</tr>
<tr>
<td>THE C EDITOR</td>
<td>13</td>
</tr>
<tr>
<td>LOADING THE EDITOR</td>
<td>13</td>
</tr>
<tr>
<td>THE EDIT SCREEN</td>
<td>14</td>
</tr>
<tr>
<td>THE EDIT KEYBOARD</td>
<td>16</td>
</tr>
<tr>
<td>USING THE EDITOR</td>
<td>19</td>
</tr>
<tr>
<td>COMPILATION</td>
<td>47</td>
</tr>
<tr>
<td>ENTERING THE COMPILER</td>
<td>47</td>
</tr>
<tr>
<td>PROCESSING SOURCE TEXT</td>
<td>48</td>
</tr>
<tr>
<td>COMPILER OPTIONS</td>
<td>59</td>
</tr>
<tr>
<td>LINKING FILES</td>
<td>63</td>
</tr>
<tr>
<td>INVOKING THE LINKER</td>
<td>63</td>
</tr>
<tr>
<td>LINKING SPECIFIED LIBRARIES</td>
<td>64</td>
</tr>
<tr>
<td>LINKER OPTIONS</td>
<td>65</td>
</tr>
<tr>
<td>EXECUTING THE OBJECT CODE</td>
<td>69</td>
</tr>
<tr>
<td>HEADER FILES</td>
<td>73</td>
</tr>
<tr>
<td>FUNCTION DECLARATIONS</td>
<td>73</td>
</tr>
<tr>
<td>CONSTANT DEFINITIONS</td>
<td>73</td>
</tr>
</tbody>
</table>
INTRODUCTION

The purpose of this manual is to illustrate how to use the Acomsoft C system. It does not teach the C language, but explains the steps necessary to create and compile C programs using the Acomsoft C system.

The manual assumes that you have a knowledge of programming in general and a basic knowledge of the C programming language and its terminology in particular. If you require tutorial information about the C language itself, look at the list of recommended books in Appendix E.

THIS MANUAL

The chapters in this manual are organised as follows:

- **The C language** deals with the differences between Acomsoft C and the version defined by Kernighan and Ritchie.
- **C system overview** introduces you to the use of the Acomsoft C system and its editor.
- **The C editor** discusses the C editor in detail and explains how to use it.
- **Compilation** discusses the compilation process.
- **Linking files and Executing the object code** tell you how to invoke the linker and execute object code.
- **Header files** describes the header files found in C.
- **Library routines** lists all library routines and gives details on each.
- **Run-time memory allocation** discusses the memory map and run-time system workspace and variables.
• Appendix A is an editor command summary.

• Appendix B is a summary of all commands.

• Appendix C alphabetically lists all library routines, giving a page number for each.

• Appendix D contains error messages.

• Appendix E is a bibliography containing a list of further reading material on the C programming language.

CONVENTIONS USED

Throughout this manual, the following conventions are used:

• Text that is entered by you, text that appears on the screen, C keywords, library routines, macros and commands are printed in the following style:

  #Save [current filename]

• Keytops are shown as you see them on the keyboard, surrounded by a box. For example:

  [RETURN]

• Combinations of keys are referred to by the key names printed side by side. For example:

  [SHIFT] [f1]

  means you must hold down the shift key while simultaneously pressing the f1 key.

THE C LANGUAGE

The Acornsoft C system is based on the version of C described in Kernighan and Ritchie's book *The C Programming Language*. This chapter documents the minor differences between Kernighan and Ritchie's version of C and the Acornsoft C system.

Throughout this chapter, Kernighan and Ritchie's definition is referred to as standard C, although it is not a formal national or international standard.

Because many other implementations of C are also based on Kernighan and Ritchie's definition, you can move C programs freely between different computers, as long as you avoid extensions to the standard peculiar to individual machines. This portability applies between the Acornsoft C system and the C system provided with the Acorn Cambridge Workstation.

Note, however, that standard C does not define a full set of library routines which all systems must provide. The routines that are mentioned are not fully defined.

Acornsoft C provides a comprehensive set of library functions. These routines conform as far as possible to the X/OPEN Standards Group proposals. (See the X/OPEN Portability Guide for information on the X/OPEN Standards Group. It is listed in Appendix E.) The most significant differences occur in the low-level I/O routines which are constrained by the operating system and filing systems. The higher-level libraries contain routines which are common to almost all implementations of C. However, it is not guaranteed that these routines will have exactly the same effect as their counterparts in other versions of C, particularly in their handling of errors.
REstrictions

This section discusses Acornsoft C restrictions. They include restrictions on:

- type specifiers
- storage classes
- anachronisms
- the implementation of \n
Type specifiers

The following combinations of type specifiers are allowed. Optional clauses are given in square brackets:

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>0 to 255</td>
</tr>
<tr>
<td>int</td>
<td>2 bytes</td>
<td>-32768 to 32767 (-2 to 2 -1)</td>
</tr>
<tr>
<td>short [int]</td>
<td>2 bytes</td>
<td>-32768 to 32767 (-2 to 2 -1)</td>
</tr>
<tr>
<td>long [int]</td>
<td>4 bytes</td>
<td>-2147483648 to 2147483647 (-2 to 2 -1)</td>
</tr>
<tr>
<td>unsigned [int]</td>
<td>2 bytes</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>[long] float</td>
<td>5 bytes</td>
<td>-1e38 to 1e38 (approx)</td>
</tr>
<tr>
<td>double</td>
<td>5 bytes</td>
<td>-1e38 to 1e38 (approx)</td>
</tr>
<tr>
<td>struct</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>union</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

User-defined types are allowed; `typedef` is fully implemented.

Enumerated types are not allowed. The `enum` data type defined in standard C allows you to construct new data types by enumerating the values which variables of that type may take. Similarly, bitfields are not implemented. Bitfields are groups of bits, located next to each other within a single integer.

Storage classes

No use is made of the register storage class. It is defined in standard C as being used to advise the compiler that the variable being declared will be heavily used and, where possible, should be put in a machine register to increase execution speed. In Acornsoft C, it is treated as being equivalent to an auto declaration, since all reasonable optimisation of that nature, whatever the declaration, is performed automatically by the compiler.

Anachronisms

Note the following points regarding anachronistic forms:

- `=<operator>` is illegal. It is used in earlier versions of C instead of `<operator>=` for assignment operators.
- `int x 3` is illegal. It is used in earlier versions of C to initialise a variable. In Acornsoft C it must be written as `int x = 3;`.

\n
In 32016 C and Unix the implementation of `\n` is ASCII 10. In Acornsoft C `\n` is ASCII 13.
The # directive

When typing preprocessor directives, do not insert a space between the # symbol and the directive. For example, the directive:

```
# define
```

is illegal. It must be entered as:

```
#define
```

EXTENSIONS

This section discusses Acornsoft C extensions. They include the data type `void`, the removal of restrictions on structure member names and the use of `floats`.

Data type `void`

Acornsoft C has an extra data type, `void`, which is included in many modern compilers but is not a part of standard C. `void` is a special data type having no value, and is used to indicate that a function returns no value. This allows expressions to be cast to type `void` in order to discard their value explicitly. The (non-existent) value of a `void` expression may not be used in any way and neither explicit nor implicit conversions may be applied to such a value. A `void` expression may be used, therefore, only as an expression statement or as an operand of a comma operator.

Structure member names

Acornsoft C makes no restrictions on the use of the same member name in different structures. This feature occurs in many modern compilers. However, it is not found in standard C. In standard C, the same member name may occur in different structures only if the fields identified by the member name and all preceding fields are the same.

`Floats`

Acornsoft C lets you use `float` in statements which require an integer (e.g. `switch`). When a `float` is given, the integer part of the `float` is used. For example, `switch (1.0)`, `switch (1.4)` and `switch (1.84320)` are all equal to `switch (1)`. 

Note: Use of this feature is not recommended. It should only be used with caution.
The Acornsoft C installation and startup procedures are described in the configuration leaflet accompanying this manual. Make sure C is installed on your system before proceeding. Once C is installed and entered you are ready to type in C commands and operating system commands.

This chapter introduces you to the Acornsoft C system command line interpreter and takes you briefly through the steps required to produce and run an Acornsoft C program.

THE C SYSTEM COMMAND LINE INTERPRETER

C is a compiled language. This means that you cannot type sections of C and have them executed immediately. In this respect, C differs from BASIC, which is ‘interactive’ and executes immediate statements as soon as they are entered.

In C, only a small set of commands is required. They can be typed in upper or lower case and include the following:

• CLOSE: This command closes all open files on the current filing system. It is useful in situations where BREAK is pressed in the middle of a compilation, resulting in the source, object and/or temporary files being left open. The file(s) are inaccessible until closed.

• COMPILE filename: This command compiles a C source text file and generates a linkable file. It may take the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-dmacroname</td>
<td>define macroname</td>
</tr>
<tr>
<td>-mnumber</td>
<td>define the maximum number of errors allowed</td>
</tr>
<tr>
<td>-f</td>
<td>turn function tracing on</td>
</tr>
</tbody>
</table>
Option | Description
--- | ---
-q | do not print the compiler title and summary information
-l | invoke the linker automatically
-r<restofline> | run after linking

These options are discussed in greater detail in the chapter entitled **Compilation**.

**EDIT filename (sideways RAM)/**EDIT filename (Second Processor): This command activates the screen editor used to create and amend a C source file. If you specify a filename after EDIT, the file specified is loaded before the editor is entered. If a file is already loaded in memory and no filename is specified when the editor starts up, the file currently in memory appears on the screen. If there is no file currently in memory, the editor contains no text.

**HELP:** This command lists all Acornsoft C commands.

**LINK filename:** This command links two or more link files together. One of the files is, by default, the standard library. LINK generates an object program which may subsequently be executed. The following options are possible:

Option | Description
--- | ---
-daddress | generate the code to end at a given memory address (in hexadecimal)
-l | do not link against the standard library, stdlib
-n | give function information

The system returns to default values whenever [BREAK] is pressed or the C system is re-entered. For example, when it looks for include files, it will look in directory `<current path>.h`.

**MODE number:** This command sets the current screen mode.

**REPORT:** This command causes the last system message encountered to be printed.

**SETPATH element[, element]:** This command specifies where C looks for the various components of the system, ie C commands, header files, etc. It is generally referred to as 'the current path'.

The default path values depend on the filing system you are using:

- DFS | setpath :0
- ADFS or NFS | setpath $.C

**SHOWPATH:** This command prints the current path.

**TRACE:** This is a toggle command relating to the function trace facility. When it is on, information about the arguments passed to certain routines and the value returned by them is sent to the standard error file. This information is given for any routines defined in source text files which
were compiled with the -f option. When it is off no information is displayed.

• (full stop): This command specifies that the rest of a line should be treated as a comment and hence ignored. For example:

  . This is a command line

Any other (ie unrecognised) command, such as hello, is interpreted as the name of a code file. C attempts to find it using the current path. If no file is found, C generates the Not Found error message.

PRODUCING AND RUNNING A C PROGRAM

This section includes a brief description of how to produce and run a C program. It uses a sample source text file included on the Acornsoft C disc, C.HelloW. The description is intended to give you an overall view of the procedures you need to follow. More detailed information is found in the following chapters of this manual.

Producing and running a program is done in four stages:

1 Prepare the source text.
2 Compile the program.
3 Link with the libraries.
4 Execute the program image.

Preparing the source text

The source text for the sample text file C.HelloW is as follows:

```c
#include <stdio.h>

main()
{
    puts("Hello World!");
}
```

Compiling the program

To compile a program from this C source text, type:

```
COMPILE HelloW [RETURN]
```

Do not give the directory name since the compiler automatically searches in directory C for the source text file.

The compiler generates a corresponding code in a form suitable for the linker and saves it to disc (or the storage device used by your filing system) in the file C.HelloW. If the compilation is successful, the following message appears on the screen (the numbers on the last line may vary depending on the system you are using):

```
Acornsoft-BSC C Compiler V1.00
Including hstdio
code: 0x0042 data: 0x000E total: 0x0050
```

If it is not successful, an error message is displayed.
Linking with the libraries

The next step is to link in the standard library functions. This is necessary to resolve references to external functions (in this example puts). To invoke the linker, type:

```
LINK HelloW [RETURN]
```

An object program, `HelloW`, is generated and the following message is displayed (the numbers on the last line may vary depending on the system you are using):

```
Acornsoft-BSC C Linker V1.00
code: 0x205F data: 0x0218 total: 0x2277
```

Executing the program image

If the previous steps have been followed correctly there is now a file in directory `O` (the letter `O`) called `HelloW`. This file contains the compiled and linked code produced from the original source text. To execute this code, type:

```
HelloW [RETURN]
```

The message `Hello World!` is printed.

---

**THE C EDITOR**

This chapter discusses the Acornsoft C editor and describes the functions assigned to keys.

You use the editor to create or amend a C source file. Using the editor to type text in a file that then becomes a source file is the first step in writing a program in C. Once you have written the program and saved it on disc as a source file, it can be input to the C compiler. The compiler then checks it for errors (see the chapter entitled **Compilation**).

Before you begin, note the following points:

- You can use another editor (such as VIEW) with Acornsoft C. However it is recommended that you use the C editor as it is written specifically for programming with C and contains features not found in other editors. It is also easy to use.
- If you are using a BBC Master 128, you can use the editor which is resident in the machine.
- If you are using a BBC Master Turbo, you must unplug the ROM-based editor before using the C editor. Press `CTRL` `BREAK` after issuing the `*unplug` command.

**LOADING THE EDITOR**

Once you have installed and loaded C, the editor can be activated in any of the 40-column or 80-column display modes using the `EDIT` command.

How you enter the `EDIT` command depends on whether you are using a computer with sideways RAM or a Second Processor.
• Sideways RAM: To activate the editor with sideways RAM, at the command line type:

```
EDIT
```

and press [RETURN].

• Second Processor: To activate the editor with a Second Processor, at the command line type:

```
*EDIT
```

and press [RETURN].

If a file is already in memory when you give the EDIT command, that file is displayed on the screen ready for editing. If no file is present in memory when the command is given, a blank screen is displayed on which you can enter text.

**THE EDIT SCREEN**

When you load the editor and press [RETURN] the edit screen is displayed. At the top is an asterisk (or a white square if you are using display mode 7). At the bottom is the status line. How the rest of the screen appears depends on whether or not you indicated a filename when you gave the EDIT command.

• Filename indicated: The screen displays the file contents with the asterisk at the end of the text and the cursor below it. The cursor marks the place where you can begin entering new text.

• Filename not indicated: The cursor appears just below the asterisk at the top of the screen, marking the place where you can begin entering text.

The status line at the bottom of the screen contains user information telling you whether the editor is in insert or overtype mode, whether cursor editing is activated and if markers are set. It is always displayed during editing.

When you first enter the editor, the default status line appears as follows:

```
#Insert 0 mark(s)
```

The parameters are:

- **#Insert**
- A blank space
- 0 mark(s)

The editor is in insert mode
Cursor editing is inactive
No markers are set

If you change the editor, for example if you turn overtype on ( [SHIFT] [I] ), activate cursor editing ( [SHIFT] [COPY] ) and set markers ( [SHIFT] [1R] ), the status line appears as follows:

```
#Over * 2 mark(s)
```

The parameters are:

- **#Over**
- *
- 2 mark(s)

The editor is in overtype mode
Cursor editing is active
Two markers are set

If you are in the editor and use the global REPLACE command, the status line also shows the number of replacements made:

```
#insert 0 mark(s)  found n
```
THE EDIT KEYBOARD

Keys and combinations of keys are assigned certain functions by the editor. This section discusses the use of the keyboard with the editor. It is divided into three parts:

- cursor movement keys
- editing control keys
- function keys

Cursor movement keys

The cursor movement (arrow) keys are used to move the cursor up, down, left and right on the screen. They are also used in conjunction with [SHIFT] and [CTRL] to move the cursor quickly within the text.

Cursor movement key assignments are as follows:

- [↑] Moves the cursor up one line
- [↓] Moves the cursor down one line
- [←] Moves the cursor one character to the left
- [→] Moves the cursor one character to the right
- [SHIFT] [↑] Moves the cursor to the top of the screen
- [SHIFT] [↓] Moves the cursor to the bottom of the screen
- [SHIFT] [←] Moves the cursor to the beginning of the line
- [SHIFT] [→] Moves the cursor to the end of the line
- [CTRL] [↑] Moves the cursor to the beginning of the text file
- [CTRL] [↓] Moves the cursor to the end of the text file
- [CTRL] [↑] Moves the cursor past the top of the text at the beginning of the file or below the asterisk (or white square in mode 7) marking the end of the file.

Editing control keys

The editing control keys are [COPY], [CTRL], [DELETE], [ESCAPE], [RETURN], [SHIFT] and [TAB].

- [COPY] performs two functions:
  - it is used to copy strings or blocks of text
  - it deletes a character at the cursor position when the editor is in insert mode.
- [CTRL] is always used in conjunction with another key or keys. It does not perform an editing function on its own.
- [DELETE] removes a character to the left of the cursor position.
- [ESCAPE] terminates copy mode and cancels functions and commands entered by mistake.
- [RETURN] is pressed when a command is entered and is ready for implementation. It is also used to activate copy mode after [SHIFT] [COPY] is pressed.
- [SHIFT] is always used in conjunction with another key or keys. It does not perform an editing function on its own.
**Function keys**

The function keys are the red keys located at the top of the keyboard. They are used to manipulate program text. The specific functions assigned to each key and key combination are as follows:

- To tab below words
- To tab in columns of eight

<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tab</td>
<td>Moves the cursor without creating spaces. You can use it in one of two ways to position the cursor:</td>
</tr>
<tr>
<td>[Search for a line]</td>
<td>Display the hard carriage return symbol (toggle)</td>
</tr>
<tr>
<td>[Enter an operating system command]</td>
<td>Choose insert/overtype mode (toggle)</td>
</tr>
<tr>
<td>[Write text from one file to another file]</td>
<td>Read text from one file into another file</td>
</tr>
<tr>
<td>[Save text to another file]</td>
<td>Remove the scroll margins</td>
</tr>
<tr>
<td>[Find and replace a string]</td>
<td>Quit the editor</td>
</tr>
</tbody>
</table>

**Note:** Using **[Shift]** + to quit the editor is only effective in sideways RAM. When using a Second Processor follow the procedure detailed in the sub-section **Quitting the editor.**

**USING THE EDITOR**

This section discusses the Acornsoft C editing functions as follows:

- How to set markers
- How to use 'hard' characters
- Character display in relation to ASCII codes
- Scroll margins
- How to load text
- How to save text
- How to use insert/overtype
- How to use the tab function
- How to delete text
- How to copy blocks of text
- How to move blocks of text
- How to read text from one file into another
- How to write text from one file into another
- How to find/replace text
- How to use command macros
- How to quit the editor
Setting markers

Some of the functions (copying, deleting, reading and writing) require the setting of markers within text. Note the following:

- If markers are set, the number of markers (1 or 2) is displayed on the status line.
- If you set markers and then decide not to use them, you can remove them by pressing [SHIFT] [8].
- You cannot enter text, insert a file in a document or use interactive find and replace when markers are set.
- The area of text defined within markers begins with the first marker placed on the first character you want to include in the block and ends with the second marker placed just after the last character you want to include in the block.

Take the following steps to set markers:

1. Position the cursor at the beginning of the section you wish to mark.
3. Move the cursor to the end of the section you wish to mark.

A block of text is now defined.

‘Hard’ characters

‘Hard’ characters are spaces that appear blank on the screen but are recognised as characters by the editor. They are made by pressing the Space Bar or [RETURN].

When you are entering text, if you reach the end of a line on the far right-hand side of the screen and wish to continue on the next line, you do not have to press [RETURN]. The text automatically wraps around and continues on the line below the first. If you do press [RETURN], however, a hard character carriage return is inserted.

Hard characters made by pressing [RETURN] are displayed by pressing [SHIFT] [8]. They appear in the form of a reverse M. Pressing [SHIFT] [8] again causes the screen display to toggle back to invisible hard carriage returns.

By making hard carriage returns visible you can see if there are any unnecessary trailing blanks on a line.

Character display and ASCII codes

You can type control characters (characters whose ASCII value is below 32) by pressing [CTRL] and the relevant character. For example, you can enter a hard carriage return by pressing [CTRL] [8]; you can enter the bell character by pressing [CTRL] [G]. The characters (in this case M and G) appear on the screen in reverse video (except in mode 7 where they appear as a white mark).

The delete character, whose ASCII code is 127, is shown as a small white mark in all modes.

Characters whose ASCII codes are greater than 127 are displayed as normal. You cannot type characters with codes between 127 and 255.
Scroll margins

Scroll margins are top and bottom screen margins. They are automatically set four lines from the top and four lines from the bottom of the screen. This means that when you move the cursor up or down, the text scrolls and the cursor stays on the fourth line from the top (or bottom).

The bottom scroll margin, the space occupied by the last four lines on the screen, is used by the find/replace function to display target strings.

When [CTRL] is used to move to the end of text, the end of text marker is displayed on the bottom scroll margin.

Loading text

Text can be loaded for editing at one of the following times:

- before the editor is loaded
- when the editor is loaded
- after the editor has been loaded

Refer to the beginning of this chapter for information on how to load the C editor. Loading text is discussed below.

- Before the editor is loaded: To load a file into memory before activating the editor, take the following steps:

  1. Load the file you wish to edit by following the procedure required by the software you are using.

  2. At the command line, type EDIT if you are using sideways RAM or *EDIT filename if you are using a Second Processor. Press [RETURN].

  3. Begin editing. The file is displayed on the screen.

Saving text

You can save an entire text file or part of a text file.

- Saving an entire text file: To save an entire text file when you finish editing it, take the following steps:

  1. Press []. The following prompt appears on the screen:

     #Save [current filename]:

- When the editor is loaded: To load a file for editing at the same time you load the editor, take the following steps:

  1. Make sure you are at the command screen.

  2. Type EDIT filename if you are using sideways RAM or *EDIT filename if you are using a Second Processor. Press [RETURN].

  3. Begin editing. The file designated on the command line is displayed on the screen.

- After the editor has been loaded: To load a file after the editor is activated, take the following steps:

  1. Load the editor as described above.

  2. Press[]. The following prompt appears on the screen:

     #Load []:

  3. Enter the name of the file you wish to edit and press [RETURN].

  4. Begin editing. The file designated at the prompt is displayed on the screen.
2 If current filename is the name of the file you wish to save, press \[RETURN]. If you wish to save the file under a new filename, enter the new filename you require and press \[RETURN].

The current filename is the name last used in a load command. It is always included with the save prompt, enclosed in brackets. It is updated every time a load command is executed.

Caution: If you press \[RETURN] without designating a filename, the file you are saving overtypes the file with the current filename, deleting its contents.

- **Saving part of a text file:** To save part of a text file, take the following steps:

1. Set a marker at one end of the text you wish to save and position the cursor at the other end.
2. Press \[]. The prompt:

   \#SAVE [current filename]:

   appears on the screen.

3. If current filename is the name of the file you wish to save, press \[RETURN]. If you wish to save the file under a new filename, enter the new filename you require and press \[RETURN].

**Inserting and overtyping text**

When the editor is activated, the system is in insert mode. This means that as you type in text everything on the screen to the right of the text you are inserting is moved along. For example, to add the character `=` next to the `<` character in the following string:

```
if (x<10)
```

position the cursor on the 1 character and type the `=` character. The characters `10` move to the right and the string appears as follows:

```
if (x<10)
```

To change to overtype mode, press \[SHIFT][]. In overtype mode, when you enter text it overtypes previously inserted text, replacing it with the text you are typing.

Note the following points regarding insert/overtype:

- \[SHIFT][] is a toggle switch, changing from insert to overtype and back again when pressed.
- When the editor is in insert mode, the status line displays the word insert. When it is in overtype mode, the status line displays the word over.
- A hard carriage return cannot be deleted in overtype mode. You must be in insert mode and use \[DELETE] or \[COPY] (see the section below on how to delete text) to remove a hard carriage return.

**Tabulating**

Tabulating is used to move the cursor without inserting hard spaces in the text. This enables you to tab below words and is chosen by pressing \[SHIFT][TAB].

To use the tab facility, position the cursor below a line of text and press \[SHIFT][TAB] to activate tabulation. You can then cause the cursor to jump to the space below the beginning of the next word in the text line by pressing \[TAB]. For example, if the cursor is positioned under the `i` of `int` in the following line:
int, lower, upper, step;

pressing [tab] successively causes the cursor to move to the space below the l of lower, the u of upper and the s of step before returning to the space below the i of int.

Deleting text

There are four ways to delete text:

• by pressing [delete] (deletes a character)
• by pressing [copy] (deletes a character)
• by marking text and pressing [shift] [insert] (deletes a block of text)
• by pressing [shift] [fs] (deletes all text)

If you delete text by mistake using any of the above methods, pressing [fs] restores the deleted text. However, if you delete text then delete more text, only the text deleted the second time can be restored.

• Pressing [delete]: To remove text using [delete], take the following steps:

  1. Position the cursor to the right of the character you wish to delete.
  2. Press [delete]. The character to the left is removed.

If you press [delete] when the cursor is at the beginning of a line, the current line joins up with the one above it.

Note that a hard carriage return is removed only by pressing [delete] or [copy] (see Pressing [copy] below) when the editor is in insert mode.

Pressing [copy]: To remove text using [copy], take the following steps:

  1. Position the cursor on the character you wish to delete.
  2. Press [copy]. The character is deleted.

[copy] only deletes text when the editor is in insert mode. Along with [delete], it is the only other key that can remove a hard carriage return.

• Marking text and pressing [shift] [insert]: To delete a block of text, take the following steps:

  1. Set a marker at one end of the text you wish to delete and position the cursor at the other end.
  2. Press [shift] [fs].

Text contained between the two markers is deleted.

• Pressing [shift] [fs]: [shift] [fs] deletes the entire contents of a file. The cursor can be positioned anywhere within the file. Use it with caution. This can be reversed, however, by immediately pressing [fs] if you accidentally delete a file.

Copying text

Text is copied either one character at a time or in blocks.

• Copying characters: Strings of characters can be copied by taking the following steps:

  1. Position the cursor where you want the text being copied to appear.
2 Press [SHIFT] [COPY] to activate the copy cursor. (The copy
cursor is separate from the text cursor. It only appears
when you press [SHIFT] [COPY].)

3 Use the arrow keys to move the copy cursor to the text
you want to copy. The text cursor remains at the place
you want the copied text to appear.

4 Press [COPY] until all the characters you are copying
appear at the position indicated in step 1.

5 Press [ESCAPE] to terminate copy mode.

- Copying a block of text: Take the following steps to copy a
block of text:

1 Set markers at the beginning and end of the text you
wish to copy.

2 Position the cursor at the point you wish the copied text
to be inserted. You cannot copy a block onto itself; do
not place the cursor between the two markers or a
Bad Marking error occurs.

3 Press [F7].

Text enclosed in the markers is copied to the position
indicated by the cursor.

Moving text

Take the following steps to move text within a file:

1 Set markers at the beginning and end of the text you wish
to move.

2 Position the cursor at the point at which you wish to insert
the text you are moving.

3 Press [SHIFT] [F7].

Text enclosed in the two markers is moved to the position
indicated by the cursor. The place from which it is moved
closes up.

Reading text

If you wish to read a block of text from one file and insert it in
the current file, take the following steps:

1 In a separate file save the block of text you wish to read into
the current file.

2 Position the cursor at the point in the file you are working
on where you want the block to be placed.

3 Press [SHIFT] [F7]. The following prompt appears:

#Insert [current filename]:

4 Press [RETURN] if the file given in the current filename is the
one you wish to read into the file you are working on. If the
current filename is not correct, type the name of the file
you wish to read into the current file and press [RETURN].

The file is read into the text at the current cursor position.
Text in front of the cursor is unaffected. Text at and after the
cursor is moved in order to make room for the file being read
into the text. The cursor is placed at the start of the text just
read in.

After a file has been read into the current file, the current
filename becomes that of the file inserted.
Writing text

To write text from the file you are working on into another file, take the following steps:

1. Set a marker at one end of the text you wish to write into another file and position the cursor at the other end.
2. Save the marked block of text as a separate file.
3. Save and exit the file you are working on and load the file in which you wish to write the block of text you have saved.
4. Follow the steps for reading one file into another.

Finding and replacing text

Finding and replacing text is one of the most useful features of the editor. You can:

- Find a line
- Selectively find a string
- Selectively find and replace a string
- Selectively find and delete a string
- Globally find and replace a string
- Globally find and delete a string
- Count the number of times a string appears in a document
- Finding and replacing with patterns

Note that when you are finding and/or replacing text selectively the search for a string begins at the cursor position. You must therefore move the cursor to the position at which you wish the search to begin.

When you are finding and replacing text globally, the search for a string automatically starts from the top of a file. It is therefore not necessary to move the cursor to the position at which you wish the search to begin. However, it is possible to limit the scope of the global search and replace by setting a marker before issuing the search command ([F]), then moving the cursor to the end of the section. When the search takes place, only text between the marker and the cursor is affected.

- Finding a line: To find a given line, take the following steps:
  1. Position the cursor at the beginning of the document.
  2. Press [F]. The following prompt appears at the bottom of the screen:
     #line:
  3. Type the number of the line you are looking for. The top line is number one. If the line specified is greater than the number of lines in the document a Not Found error message occurs.
  4. Press [RETURN].

The line you indicated appears on the bottom scroll margin.

- Selectively finding a string only: To find a string, take the following steps:
  1. Position the cursor at the beginning of the document.
  2. Press [F]. The following prompt appears:
     #Find and replace:
  3. Enter the string you are searching for, for example:
     #Find and replace: string1
4 Press \[RETURN\]. The cursor moves to the first occurrence of string1 and the following prompt appears at the bottom of the screen:

\#R(Replace), C(ontinue) or ESCAPE

5 Choose an option:

- \[R\] prompts for a replacement string
- \[C\] moves to the next occurrence of string1
- \[ESCAPE\] exits the find/replace function

If you choose \[R\] the screen prompts:

\#Replace by:

Enter a replacement string and press \[RETURN\]. The replacement is made automatically and the cursor moves to the next occurrence of string1.

If you choose \[C\], the cursor moves to the next occurrence of string1 and the find/replace function ends.

When the search is complete, the message:

Not found

*********Escape to continue*********

is displayed. Press \[ESCAPE\] to return to the text.

- \[Selectively finding and replacing a string\]: To find and replace a string, take the following steps:

1 Position the cursor at the beginning of the document.
• Selectively finding and deleting a string: To find and delete a string, take the following steps:

1. Position the cursor at the beginning of the document.
2. Press [HOME]. The following prompt appears:
   
   #Find and replace:

3. Enter the string you wish to delete as shown in the following example:

   #Find and replace: string2/

4. Press [RETURN]. The cursor moves to the first occurrence of string2 and the following prompt appears at the bottom of the screen:

   #R(eplace), C(ontinue) or ESCAPE

5. Choose an option:

   - R deletes string2
   - C moves to the next occurrence of string2
   - ESCAPE exits the find/replace function

   If you choose R, string2 is automatically deleted and the cursor moves to the next occurrence of string2.

   If you choose C, the cursor moves to the next occurrence of string2 without deleting the first occurrence of string2.

When the search is complete, the message:

Not found

**********Escape to continue********** is displayed.

6. Press [ESCAPE] to return to the text.

• Finding and replacing a string globally: To perform this function, take the following steps:

1. If you are setting a marker, insert it in the text and position the cursor.
2. Press [HOME]. The following prompt appears:

   #Global replace:

3. Enter the string you wish to replace as shown in the following example:

   #Global replace: oldstring/newstring

4. Press [RETURN]

   All occurrences of oldstring are automatically replaced with newstring. The number of times the replacement occurs is shown on the status line.

• Finding and deleting a string globally: To perform this function, take the following steps:

1. If you are setting a marker, insert it in the text and position the cursor.
2 Press [R]. The following prompt appears:

#Global replace:

3 Enter the string you wish to delete as shown in the following example:

#Global replace: string1/

4 Press [RETURN].

All occurrences of string1 are automatically deleted. The number of times the deletion occurs is shown on the status line.

- Counting occurrences of a string: To count the number of times a string occurs in a file, take the following steps:

1 If you are setting a marker, insert it in the text and position the cursor.

2 Press [R]. The following prompt appears:

#Global replace:

3 Enter the string you wish to count as shown in the following example:

#Global replace: string2

4 Press [RETURN].

The number of times string2 occurs is shown on the status line.

- Finding and replacing with patterns: In the examples of find and replace discussed so far, strings of straightforward text have been used. However, by adding special characters to a string it is possible to specify variables that perform certain find and replace functions. These strings, made up of constant parts (literal text) and variable parts (wildcards, ranges, choices, repeats and inversions) are called patterns. The special characters used in patterns are as follows:

- matches any single ASCII character between 0 and 255. All wildcards may be duplicated, so two .. symbols match any two characters, etc. For example, to find the number of characters in a C program, press [R]. At the prompt type:

#Global replace: -

and press [RETURN].

The characters are counted and the total is displayed on the status line.

- matches any alphanumeric, upper case, lower case or underline character (0-9, A-Z, a-z, _).

- matches any digit (0-9). For example, to find all numbers containing three digits, press [M]. At the prompt type:

#Find and replace: # # #

and press [RETURN].

- [xyz] matches any of the characters specified in the brackets, in this case x or y or z. For example, to replace all occurrences of x or y or z by , press [R]. At the prompt type:

#Global replace: [xyz]/

and press [RETURN].
a-z matches any valid hexadecimal letter between the character on the left and the character on the right, in this case a and z.

$ matches the hard carriage return character.

\[ matches [\text{CTRL}][c] (VDU 3).

\[ makes the next character a control character.

\[a matches ASCII 0, \[A means ASCII 1, etc.

\[? matches the delete character (ASCII 127).

\[ is the escape character.

\[c means ASCII c + 128. For example, \[!@ matches chr(128) and \[?! matches chr(255).

\[ matches anything but the symbol that follows (which can be a wildcard). Thus "A matches anything but A," # matches any non-digit and "A-Z matches anything that is not an upper case letter.

\[ removes any special meaning from the symbol that follows it. Thus \[ stands for $, not carriage return, \[ prevents the character after the bar from being interpreted as a control character, \[ means ., not 'any character', etc.

\[ matches zero or more of a pattern (a 'multiple match' pattern). Thus \[ matches any sequence of characters. It also always matches the shortest string possible, which is why it can match nothing at all. For example, \[ (an asterisk followed by a space) matches zero or more spaces and \[\[A-Z] matches zero or more upper case letters. The pattern printf matches exactly the same strings as printf because the \[* on the end does as little work as possible (ie it matches the null string).

To force * patterns to do more work, they must end with a 'solid' character, for example $ or $. See the third, fourth, fifth and sixth examples below.

\& means 'whatever was matched by the pattern'. It has special meaning only in a replacement string (although it still needs a \ before it if it is to be read literally in a pattern). The replacement string can be either global or selective. For example, if the pattern is a literal such as 'while', the & symbol stands for 'while'.

When wildcards and repeats (*) are used in a pattern it is not possible to know exactly what is matched. For example, to duplicate all digits so that 1 becomes 11 and 123 becomes 112233, use the following global replace:

```
#Global replace: #/&& [RETURN]
```

The ampersand becomes whatever is matched by the pattern.

\%n means a certain field in a pattern designated by the number n, where n must be a digit between 0 and 9.

A field is any non-constant item in a pattern, i.e. a wildcard character, a multiple match (a symbol preceded by *), an inverted match (a symbol preceded by ^), a range (a-z) or a choice ([13579]). The fields are numbered from zero on the left. See examples thirteen and fourteen below.

You can combine ranges and choices using the above characters in patterns. Note the following examples:
• Example 1: To match any hexadecimal character use:

```
[0-9A-F]
```

or:

```
[#A-F]
```

• Example 2: To match any character in the range a to z or A to Z or the character _, use:

```
[a-zA-Z_]
```

This mixes single characters and ranges so that a pattern to match characters at the start of a C variable name is possible.

• Example 3: To delete the trailing spaces from all lines in the text use:

```
#Global replace: * $/$ RETURN
```

This means that all occurrences of zero or more spaces followed by a hard carriage return are replaced with a hard carriage return.

• Example 4: To find all occurrences of printf which output a new line use:

```
#Find and replace: printf("\n") RETURN
```

This finds all occurrences of printf( the " character followed by a sequence of any ASCII characters other than another quote, followed by \n. Two backslashes are required as \n itself is a special character and therefore needs a preceding backslash to cancel its special meaning.

• Example 5: To match one or more occurrences of a character (as opposed to zero or more) use:

```
#Find and replace: c* RETURN
```

• Example 6: To find all blank lines use:

```
#Find and replace: $* $ RETURN
```

A blank line means a hard carriage return followed by another hard carriage return with only zero or more spaces in between. The first $ matches a hard carriage return; the * (an asterisk followed by a space) matches zero or more spaces; the second $ matches the hard carriage return of the blank line.

• Example 7: To match any control character use:

```
#Find and replace: [\@-\_] RETURN
```

\@ is CTRL@, the lowest valued control character (ASCII 0) and _ is CTRL_, the highest control character (ASCII 31).

• Example 8: To match a hexadecimal constant use:

```
#Find and replace: 0x[\#A-F]*[\#A-F] RETURN
```

This means 0x must be followed by one or more hexadecimal characters.

• Example 9: To count the number of lines in a file use:

```
#Global replace:$ RETURN
```

This displays on the status line the number of lines found in the file.
• **Example 10:** To match two fields (field 0 and field 1), use:

```
#Find and replace: A*@. [RETURN]
```

The characters matched by the @ are field 0 and the character matched by the . is field 1.

• **Example 11:** To match four fields use:

```
#Find and replace: ###* [RETURN]
```

This matches four fields: #, #, ~ and * respectively.

• **Example 12:** To match any number of non-digits followed by one or more digits use:

```
#Find and replace: *"#### [RETURN]
```

Field 0 is *"#; field 1 is # and field 2 is *#.

• **Example 13:** To reverse the order of alternate characters use:

```
#Global replace: ../%1%0 [RETURN]
```

Field 0 is the character matched by the first . (full stop); field 1 is the character matched by the second . (full stop).

• **Example 14:** To change all variables which end in X (upper case) so they begin with x (lower case), use:

```
#Global replace:*&X~@/x_%0%1 [RETURN]
```

Using command macros

Command macros give you the capability of assigning repetitive tasks to any of the [60]-[67] and cursor movement keys. All the commands in EDIT are ASCII characters above 127, making it easy to redefine the keys to issue several commands at a single keystroke.

Each [60]-[67] and cursor movement key has a string assigned to it. This string must be entered when the key is redefined. The table below lists the keys that can be redefined and their assigned strings:

<table>
<thead>
<tr>
<th>Key</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>[60]</td>
<td>! ! @</td>
</tr>
<tr>
<td>[SHIFT][60]</td>
<td>! ! P</td>
</tr>
<tr>
<td>[CTRL][60]</td>
<td>! !</td>
</tr>
<tr>
<td>[61]</td>
<td>! ! A</td>
</tr>
<tr>
<td>[SHIFT][61]</td>
<td>! ! Q</td>
</tr>
<tr>
<td>[CTRL][61]</td>
<td>! !</td>
</tr>
<tr>
<td>[62]</td>
<td>! ! B</td>
</tr>
<tr>
<td>[SHIFT][62]</td>
<td>! ! R</td>
</tr>
<tr>
<td>[CTRL][62]</td>
<td>! !</td>
</tr>
<tr>
<td>[63]</td>
<td>! ! C</td>
</tr>
<tr>
<td>[SHIFT][63]</td>
<td>! ! S</td>
</tr>
<tr>
<td>[CTRL][63]</td>
<td>! !</td>
</tr>
</tbody>
</table>

 retain the X at the end it must be explicitly given in the replacement:

```
#Global replace:*àX^@/x_%0%1 [RETURN]
```

Each [60]-[67] and cursor movement key has a string assigned to it. This string must be entered when the key is redefined. The table below lists the keys that can be redefined and their assigned strings:
Command macros are defined by pressing \[F1\] to allow you to enter an operating system command, followed by the word KEY and the number of the key you wish to reassign.

For example, one of the most useful applications of this command macro capability is to execute several global replacements in quick succession. This is done by pressing \[F1\] once the editor is loaded and entering the following:

\[KEY01MI!1E<from>/<to>1M \text{RETURN}10\]

The parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY0</td>
<td>assigns the replacement function.</td>
</tr>
<tr>
<td>1M</td>
<td>is the [RETURN] that is normally pressed to return from the star prompt.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>E</td>
<td>issues the command (as if [E] is pressed).</td>
</tr>
<tr>
<td>&lt;from&gt;/&lt;to&gt;</td>
<td>designates the search and replacement patterns.</td>
</tr>
<tr>
<td>RETURN</td>
<td>is the [RETURN] that is normally pressed after a global replace command.</td>
</tr>
<tr>
<td>RETURN</td>
<td>on its own ends the key definition.</td>
</tr>
<tr>
<td>G</td>
<td>executes the global replacement.</td>
</tr>
</tbody>
</table>

**Quitting the editor**

There are two methods for quitting the editor, dependent upon whether you are using sideways RAM or a Second Processor.

- When using sideways RAM press [SHIFT] [E]
- When using a Second Processor [SHIFT] [E] has no effect. Therefore take the following steps:

1. Press [E]. The following prompt appears on screen:
   
   *   

2. Type TC.

3. Press [RETURN].

**ENTERING THE COMPILER**

To run a source file through the compiler, at the prompt type:

```plaintext
COMPILE filename
```

and press [RETURN].

The compiler automatically looks for the source file in the current directory, sub-directory C (unless you designate a specific pathname). If it does not find the file, it searches each of the places specified in the current path. For example, if you are using an ADFS system, the current directory is `$.develop`, and the `setpath` is `$.C` (the default value), the compiler searches the directories `$.develop.C` and `$.C.C`.

If it finds the file, the compiler automatically starts to process the text in that file. If not, it displays an appropriate error message. For example:

```plaintext
Acornsoft-BSC C Compiler V1.00
Cannot open file filename: Not found
Total 1 error(s)
```
Processing the source text through the compiler involves three passes:

- preprocessing
- syntax checking and code generation
- postprocessing

**Preprocessing**

During preprocessing, the compiler carries out the following four tasks:

- macro substitution
- file inclusion
- conditional compilation
- line control

**Macro substitution:** A macro is a section of C source text identified by a name or token and defined within the source text using the directive `#define`. (Do not confuse these macros with command macros which redefine keystrokes when using the C editor—see the chapter entitled *The Acornsoft C Editor* for a discussion on command macros.)

Note the following points regarding macro substitution:

- You define macros within the C source text using the `#define` directive. For example:

  ```c
  #define MAXVAL 12
  ```

  tells the preprocessor to replace any occurrence of `MAXVAL` with 12.

- Exchanges take place for all occurrences between the definition and the end of the source text, unless you use a `#undef` directive (see below) or if the name occurs within a quoted string. If the name occurs within a quoted string, it is not changed.

  For example, the C source code:

  ```c
  for (number=1 ; number <= MAXVAL ; ++number) { 
    if (high < result(number)) high = result(number); 
    printf("CURRENT MAXVAL = %d\n",high);
  }
  ```

  becomes:

  ```c
  for (number=1 ; number <= 12 ; ++number) { 
    if (high < result(number)) high = result(number);
    printf("CURRENT MAXVAL = %d\n",high);
  }
  ```

  when run through the preprocessor.

- Macros do not need to be single statements and can take up to ten arguments. They can consist of several statements and may be split over several lines. If they are split over several lines, you must insert a backslash (`\`) at the end of each line that is continued. For example:

  ```c
  #define SUM(x,count,result)\ 
  (result) = 0;\ 
  for ((count)=1 ; (count)<=(x) ; ++(count))\ 
  (result) += (count);
  ```

  In this case, the preprocessor replaces:

  ```c
  SUM(10,temp,result)
  ```
with:

(result) = 0;
for ((temp)=1 ; (temp)<=(10) ; ++(temp))
(result) += (tmp);

- A replacement string may itself reference other macros. For example:

#define SQUARE(x) (x)*(x)
#define SIDESQ(hyp,x) (SQUARE(hyp) - SQUARE(x))
#define PSIDESQ(h,s) printf("THE VALUE IS %d\n",SIDESQ(h,s))

In this case,

PSIDESQ(5,4);

is replaced by:

printf("THE VALUE IS %d\n",5*5-4*4);

This example involves three levels of macro definition:
SIDESQ which references SIDESQ, SIDESQ which references SQUARE, and SQUARE. Up to ten levels can be used.

- You can redefine a particular macro identifier at a later stage in the source text and can even return it to a previous definition using the #undef directive. For example:

#define MAXVAL 10
w = MAXVAL;
#define MAXVAL 20
x = MAXVAL;
#define MAXVAL 30
y = MAXVAL;
#undef MAXVAL
z = MAXVAL;

is replaced by:

w = 10;
x = 20;
y = 30;
z = 20;

#undef clears the most recent definition of MAXVAL, ie #define MAXVAL 30, and restores the definition which was current at the time that definition was used, ie #define MAXVAL 20.

- File inclusion: You can include the contents of different files in a source file by using the directive #include.

Note the following points regarding file inclusion:

- You can include the contents of a file in a source file by typing:

#include filename

The compiler replaces the #include directive with the entire contents of the file called filename.

- You can include macro definitions in a source file. For example, by typing:

#include "h.math"

inserts the standard maths definitions which are provided on the C system disc, so that macros like M_PI may be used.
- You can tell the compiler to search for a source file only along the current path and not in the directory of the original source file by typing:

```
#include <filename>
```
in the source file.

- Files to be inserted may also contain the `#include` directive. For example, the directives:

```
#include "h.math"
define PISQ (Pl*Pl)
```
can be placed in another file (called `MATHS2`). Then the following directive:

```
#include "MATHS2"
```
is replaced by all the definitions in `h.math` plus the definition of `PISQ`.

- **Conditional compilation**: Conditional compilation allows you to designate which parts of the source text are compiled. This is done by using conditional compiler directives. The directives are as follows:

  - `#if constant expression` checks to see if the constant expression evaluates to true (non-zero).
  - `#ifdef identifier` checks whether the identifier is not currently defined. It is the opposite of `#ifndef`.
  - `#else` makes checks which may produce the value true or false. If the result of the check is true, then the lines from the point of the check down to either an `#else`, if one is present, or a `#endif`, are compiled. If there is a `#else`, any text between it and the `#endif` is ignored. If the result is false, then the opposite happens. The lines from the point of the check down to either the `#else` or the `#endif` are ignored, and any between the `#else` and `#endif` are included.
  - `#ifndef identifier` makes checks which may produce the value true or false. If the result of the check is true, then the lines from the point of the check down to either an `#else`, if one is present, or a `#endif`, are compiled. If there is a `#else`, any text between it and the `#endif` is ignored. If the result is false, then the opposite happens. The lines from the point of the check down to either the `#else` or the `#endif` are ignored, and any between the `#else` and `#endif` are included.

`#if` and `#else` are sufficient if the condition being tested has only two distinct states, ie whether a flag is set or not. However, different actions may be required if more than two distinct states are tested. For example, if the value of an `int` is positive, negative or zero `#elif` may be used:

```
#if (code<0)
  ...code section 1...
#else (code>0)
  ...code section 2...
```
#assert constant
expression

- #assert constant
expression

Conditional compilation has many different uses, however
the most common application is to use it for including
debugging code. For example:

```c
#if DEBUG
printf ("current value of x = \%d\n",x);
printf ("current value of y = \%d\n",y);
#endif
```

The two messages are printed out only if `DEBUG` is true
(set to non-zero).

- **Line numbering:** Line numbering assigns a number to each
line in a program. Should an error be detected during
compilation, the line on which the error occurs is
identified by the number assigned to it. Along with the line
number, the name of the source file in which the error
occurs is also displayed.

Lines are counted from the beginning of the source text,
starting with line 1. The filename is the name under which
the source text file is saved. However, you can instruct the
compiler to redefine both the line number and the filename
for the purposes of error diagnostics using the `#line`
directive:

```c
#line constant identifier
```

This defines the line number of the next source line to be
the value of the constant and the current input file to have
the name of the identifier. For example:

```c
#line 12 errtrap
```

The identifier is optional. If it is not there, the filename
remains unchanged.

If the preprocessor pass is successful, no messages are
produced. The compiler moves on to the second pass,
syntax checking and code generation. However, if any
errors occur during the preprocessor stage, the compilation
is terminated at the end of the preprocessor pass and details
of the error(s) are output.

Error messages include the following:

- the name of the file
- the number of the line which contained the error
- a message saying what the fault is
Possible sources of errors are:

- badly defined or called macros
- #include files which are not found or have illegal names

The following example is a program containing a badly defined macro:

```c
/* error1 */
main()
{
#define MIN 4
#define MAX 12
#define AV(x,y) (x+y)/2

    AV(MIN);
}
```

When compiled, the following error message is displayed:

```
Acornsoft-BSC C compiler V1.00
File error1 line 10 Macro with wrong number of parameters
Exit(-1)
```

A list of possible compiler error messages is given in Appendix D.

Syntax checking and code generation

When preprocessing is complete, the compiler begins the second pass: syntax checking and code generation.

During this pass, the syntax of the fully expanded C text is checked and the corresponding code is generated in a temporary data file. Any questionable program or syntax fault found causes either a warning message or an error message to be displayed.

- **Warning messages:** Warning messages are given for dubious programs which are valid C programs but contain undesirable expressions. When the compiler encounters such a situation, a warning message is displayed but code is still produced.

An example of a valid but undesirable C program that contains a conditional expression where the expression is constant is as follows:

```c
/* warn1 */
main()
{
    int x;

    x = (3 > 2) ? 1 : 2;
}
```

It produces the following message (the display on the last line may vary depending on the system you are using):

```
Acornsoft-BSC C Compiler V1.00
File warn1 line 8 Warning: Condition is constant in x?y:z
code: 0x000C data: 0x0000 total: 0x000C
```

The three values given are, respectively:

- the size of the code section in bytes
- the size of the data section in bytes
- the total size in bytes
• Error messages: Error messages are given for incorrect programs. They occur when the compiler is unable to make sense of the source text and as a result has to halt code generation.

When an incorrect program is encountered, the compiler continues parsing the syntax of the program so that as many errors as possible are reported together. It does this by scanning through the text until it finds a structure or statement delimiter which it recognises and starts checking again from that point. This means that you can only rely on the first error message, since others reported later may be the result of the compiler either having missed out a vital section of code or starting to check again from an unsuitable position.

Error messages have a similar format to warnings. The file below contains two errors:

```c
/* error2 */
main()
{
  int min,max,av;
  float av;
  av = (min+max)/2
}
```

it produces the following message:

```
Acornsoft-BSC C Compiler V1.00
File error2 line 7 Multiply defined local identifier av
File error2 line 10 Semi-colon expected
Total 2 error(s)
```

**COMPILER OPTIONS**

The options listed below may be given to the compiler. During compilation they may be passed in any order as long as `-l` (if present) is last:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d</code></td>
<td>This option must be followed by a macro definition. It is equivalent to having defined (using the directive <code>#define</code>) the given name in the program being compiled. More than one <code>-d</code> option may be given on a line. For example: COMPIL test -dNDEBUG -dNSCANF</td>
</tr>
<tr>
<td><code>-dNDEBUG</code></td>
<td>takes out any assertions made about the program using <code>assert()</code>.</td>
</tr>
<tr>
<td><code>-dNSCANF</code></td>
<td>is used if <code>&lt;stdio&gt;</code> is included and <code>scanf()</code> code is not wanted. An example is linking with <code>stdlib</code>.</td>
</tr>
<tr>
<td><code>-f</code></td>
<td>This option provides function tracing. Any functions defined in the source text being compiled are 'marked' for later use. Once the functions are marked, the <code>TRACE</code> command is</td>
</tr>
</tbody>
</table>
used to turn tracing on. When the resulting object program is run, each call to one of these functions produces debugging information which is sent to the standard error stream. This information consists of the values of all the arguments being passed to the function, and the value that it returns (if the function is not of type void). For example:

```c
/* > c.debug3 */

main(argc,argv)
int argc;
char *argv[];
{
    int argno;
    for (argno=1;argno<argc;++argno)
        (void)myputs(argno,argv[argno]);
    return 42;
}

int myputs(argno,arg)
int argno;
char *arg;
{
    printf("Arg %d is %s\n",argno,arg);
    return strlen(arg);
}
```

If this is compiled, linked and the resulting object program run in debug mode using:

```
COMPILE debug3 -f
LINK debug3
TRACE
Trace On
d debug3 fred jim sheila
```

the following output is produced:

```
main(argc=0004,argv=0641)
myputs(argcno=0001,argv=05A8)
Arg 1 is fred
myputs returned 0004
myputs(argcno=0002,argv=05AD)
Arg 2 is jim
myputs returned 0003
myputs(argcno=0003,argv=05B1)
Arg 3 is sheila
myputs returned 0006
main returned 002A
Exit(42)
```

If the compilation is successful, this option immediately invokes the linker, passing the remainder of the command line to it. Therefore,

```
COMPILE test -f -l -r
```

can be thought of as an abbreviation for:

```
COMPILE test -f
LINK test -r
```

This option must be followed by a number. It sets the maximum number of errors which the compiler reports before it stops parsing the code.
and returns to C command mode. The default value is 20. For example:

```bash
COMPILE test -m1
```

stops as soon as the first error is found.

This option turns off the compiler's header and summary information. Normally, when the compiler is invoked successfully, it produces the following information:

```
Acornsoft-BSC C Compiler V1.00
code: 0x0016 data: 0x0000 total: 0x0016
```

which is sent to the standard output stream. By default the standard output stream is the screen (the chapter entitled Executing the Object Code tells you how to redirect the output stream) Using the `-q` option stops the production of these messages. However, any error messages are still displayed.

---

**LINKING FILES**

After the C compiler finishes examining the source file and all errors are corrected, it outputs to an intermediate file. The intermediate file is in a form suitable for the linker, as it becomes the input file for the linker.

The linker's main task is to take the information produced by the compiler and fit it all together to form an executable C program. This chapter looks at how this is done.

All but the most trivial C programs access functions not defined in the main source text file. Many of these are library functions that perform operations such as reading and writing data and evaluating mathematical functions, for example `SIN` and `COS`. Others may be user-defined functions, written separately for ease of maintenance. During compilation, the compiler makes a note of all functions which are referenced. The linker provides information about where these are found.

**INVOKING THE LINKER**

To invoke the linker, type:

```bash
LINK filename
```

and press [RETURN].

This takes the file `L.filename` produced by the compiler, adds to it the link code of the standard library routines in `stdlib`, and ensures that all the functions called are present. Providing they are, it produces the following message (information on the last line may vary depending on the system you are using):

```
Acornsoft-BSC C Linker V1.00
code: 0x207F data: 0x0225 total: 0x22A4
```

A file called `o.filename`, containing the executable code, is generated.
If a function which is referenced is not found, the linker gives an appropriate error message. For example:

```plaintext
Acornsoft-BSC C Linker V1.00
*** ERROR pust not found while resolving file L.HelloW1
Total 1 error(s)
```

**LINKING SPECIFIED LIBRARIES**

You can also link libraries explicitly. For example:

```plaintext
link testfile mylib
```

allows the linker to resolve all references to the user-defined functions in the library `L.mylib`.

A file called `stdlib` contains the code for all the library functions. If you link against `stdlib`, all references to library functions are included. However, `stdlib` is large and if used it increases the size of the object code by approximately 8k bytes.

If the source text needs to access only a small subset of the total number of library functions, it is better to link against a library containing fewer functions.

Three other libraries which are subsets of `stdlib` are included in Acornsoft C:

- **smallib**: contains all but `printf` and `scanf` (approx 5k).
- **system**: contains the system minus the high level i/o routines (approx 3k).
- **minlib**: contains workspace definitions, and a very minimal set of routines (approx 1/4k).

---

You can use any of these instead of `stdlib` by telling the linker not to include `stdlib` as follows:

```plaintext
LINK testfile system -l
```

The `-l` is an option which tells the linker you do not want to use `stdlib`.

Note the following points regarding the creation of libraries of user defined functions:

- All files are compiled independently.
- Only one file must contain a main program.
- All other files must not contain a main program.

For example:

```plaintext
LINK testfile tmod2 tmod3
```

**LINKER OPTIONS**

As shown above, the linker can be given options. These can be in any order so long as `-r` (if present) is last. A full list of the options available follows:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-d</code></td>
<td>An address is required</td>
</tr>
<tr>
<td></td>
<td>C object code is not relocatable, ie it contains references to absolute locations in memory. Hence, it can only be executed at one address. This option allows you to specify that address.</td>
</tr>
</tbody>
</table>

Specifying an address is useful if you are producing the code on one system for use on a different...
Option  Description

system. You should produce the code so that it lies immediately below the bottom of the run-time system you are going to run it on. The default (if -d is not specified) is that of the current run-time system. The address given is at the bottom of the run-time system. It is in hexadecimal and is one more than the higher location to be used by the object code. For example, a program which is to be run on a standard Master 128 appears as follows:

```
LINK testfile -d8000
```
A program which is to be run on a machine with a Second Processor attached appears as follows:

```
LINK testfile -d0000
```

-l Exclude stdlib
This option instructs the linker not to include stdlib. It should be used when one of the smaller libraries is to replace stdlib. For example:

```
LINK testfile minlib -l
```

Note: To use smallib and include <hstdio>, you must first use:

```
#define NPRINTF
#define NSCANF
```
before including <hstdio>. If these are omitted hstdio declares printf, scanf etc and chokes the linker.

-n List functions
This option gives you information about all the values and addresses currently being linked, including both user-defined functions and library functions. It presents the information in an alphabetical list giving the identifier and the following information:

- what sort of value it is:
  A = absolute value
  C = C function
  D = data

- addresses at which object codes are located or the value

- files in which identifiers are defined

-q Regulates information sent to the standard output stream
This option prevents the title line and summary information from being sent to the standard output stream. Normally, a command such as:

```
LINK testfile
```
produces the following output (this information may vary depending on the system you are using):

```
Acornsoft-BSC C Linker V1.00
code: 0x2080 data: 0x0225 total: 0x22a5
```
**EXECUTING THE OBJECT CODE**

The simplest way to execute the object code produced by the linker is to type its filename and any arguments it requires to the C command line interpreter. For example:

```bash
testfile arg1 arg2 .... argn
```

However, you may give several options which are of relevance when input and output routines are being used. These redirect the standard input, output and error streams to given files:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; filename</td>
<td>redirects stdout to a file called filename.</td>
</tr>
<tr>
<td>&gt;&gt; filename</td>
<td>redirects stdout to a file called filename, creating filename if necessary, or, if filename already exists, appending any output to the end of it.</td>
</tr>
<tr>
<td>#filename</td>
<td>redirects stderr to a file called filename.</td>
</tr>
<tr>
<td>#&gt; filename</td>
<td>redirects stderr to a file called filename, creating filename if neccessary, or if filename already exists, appending any output to the end of it.</td>
</tr>
<tr>
<td>&lt;filename</td>
<td>redirects stdin to come from a file called filename. In this case, filename must exist.</td>
</tr>
</tbody>
</table>

Examples of the above options are as follows.

To open files from, to and errs as stdio, enter:

```bash
testprg arg1 <from arg2 >to #errs
```
The object program is then run as:

```
testprg arg1 arg2
```

ie with the file indirections stripped out. However, if you attempt to redirect a stream twice, such as:

```
testprg <from1 <from2 arg
```

only the first attempt, the `<from1`, is taken as a file direction. The other file, `<from2` is passed as an argument. Thus, in this example, `<from1` is opened as stdin and the command line becomes:

```
testprg <from2 arg
```

Similarly:

```
COMPILE testfile >stdout -1 >>stdout
```

becomes:

```
COMPILE testfile -1 >>stdout
```

and then:

```
LINK testfile >>stdout
```

Note that the main program is really an integer function. A non-zero return from `main()` prints, on exit, its return value (which, if not set up by a return statement, is garbage).

The following is an example of exiting when the return value is zero:

```
main()
{
    return 0;
}
```

The following is an example of exiting when the return value is non-zero:

```
main()
{
    return 42;
}
Exit(42)
```

For the definition of `exit` refer to `exit` under System Calls in the chapter entitled Library Routines.
HEADER FILES

The Acornsoft C system contains two types of header files which declare variables and functions available in the standard library files.

FUNCTION DECLARATIONS

h.stdio and h.string declare the following functions:

- h.stdio: contains the declarations of all the high-level file i/o routines.
- h.string: contains the declarations of the memory/string operations.

CONSTANT DEFINITIONS

There are four header files containing a range of useful constants: h.osdefs, h.errno, h.fcntl and h.math.

- h.osdefs: contains operating system entry points.
- h.errno: contains error numbers.
- h.fcntl: contains filing system/low-level file constants.
- h.math: contains mathematical constants.

The constants found in these header files are as follows:

- h.osdefs
  
  OSFIND  open or close a file for byte access
  OSGBP  read or write a group of bytes
  OSBPUT  write a single byte to an open file
  OSBGET  get one byte from an open file
  OSARGS  read or write an open file's arguments
OSFILE  read or write a whole file or its attributes
OSRDCH read character from currently selected input stream
OSASCI write character routine
OSNEWL write a newline to selected output stream
OSWRCH write character to currently selected output stream
OSWORD OS call specified by A taking parameters pointed to by X & Y
OSBYTE OS call specified by A taking parameters in X & Y
OSCLI pass text to the command line interpreter

• h_errno

ENOERR no error
EINVAL not a valid operation
EPATH path too long
EINTR escape
EACCESS access violation (ADFS)/insufficient access (NFS)
ENFILE cat full (DFS)
EFBIG cannot extend (DFS)/Who are you? (NFS)
EMFILE too many open files (ADFS, DFS, NFS)

ELOCK locked (ADFS, DFS, NFS)
.EXIST exists (ADFS, DFS)
ENOSPC disc full (ADFS, DFS, NFS)
EROFS disc protected (ADFS, DFS, NFS)
EBNAME bad name (ADFS, DFS, NFS)
ENODEV drive not ready (ADFS)/faulty drive (DFS)
ENOTDIR bad directory (DFS)
ENOENT file not found (ADFS, DFS, NFS)
EBADF channel (ADFS, DFS, NFS)
EEOF eof (ADFS, DFS, NFS)
EWILD wildcards (ADFS)/Bad string (NFS)
EBCOM bad command (ADFS, DFS, NFS)

• h.fcntl

O_WRONLY open for writing only (only one of these is allowed)
O_RDONLY open for reading only (only one of these is allowed)
O_CREAT create a new file for reading or writing
O_APPEND all output is appended to the end of a file
O_TRUNC if the file exists, its length is truncated to zero
if \texttt{O\_EXCL} and \texttt{O\_CREAT} are set, open fails if the file exists

\texttt{O\_NPATH} \quad open, but do not look down the system path for the file

\texttt{O\_NCSD} \quad open, but do not use currently selected directory for the file

\texttt{O\_RDWR} \quad open for reading and writing (only one of these is allowed)

\texttt{F\_OK} \quad test file existence

\texttt{R\_OK} \quad test owner read permission

\texttt{W\_OK} \quad test owner write permission

\texttt{X\_OK} \quad test owner execute permission

\texttt{L\_OK} \quad test locked for owner – BBC specific

\texttt{PR\_OK} \quad test public read permission – BBC specific

\texttt{PW\_OK} \quad test public write permission – BBC specific

\texttt{PX\_OK} \quad test public execute permission – BBC specific

\texttt{PL\_OK} \quad test locked for public – BBC specific

\texttt{SEEK\_SET} \quad set file pointer to offset from start of file

\texttt{SEEK\_CUR} \quad set file pointer to current plus offset

\texttt{SEEK\_END} \quad set file pointer to filelength plus offset

\texttt{h\_math} \quad the largest floating point number that is represented by the Acornsoft C system

\texttt{MAXFLOAT} \quad the value that is returned by a routine when the value that should be returned is too big to be represented

The following are all math constants contained in \texttt{h\_math}:

\texttt{M\_E} \quad \texttt{M\_LOG2E} \quad \texttt{M\_LOG10E} \quad \texttt{M\_LN2} \quad \texttt{M\_LN10} \quad \texttt{M\_PI} \quad \texttt{M\_PI\_2} \quad \texttt{M\_PI\_4} \quad \texttt{M\_1\_PI} \quad \texttt{M\_2\_PI} \quad \texttt{M\_2\_SQRTPI} \quad \texttt{M\_SWRT2} \quad \texttt{M\_1\_SQRT2}
LIBRARY ROUTINES

All the library routines, both functions and macros, are listed below, grouped according to the type of operation they perform:

- standard i/o routines
- low level i/o routines
- system calls
- heap allocation
- memory operations
- string operations
- character classification
- conversions

The routines are listed alphabetically, along with the page number on which the explanation of each is found, in Appendix C.

The explanation provided for each routine discussed is divided into four parts:

- the name of the routine
- a synopsis which includes header file(s), type returned and arguments
- a description
- an example

STANDARD INPUT/OUTPUT ROUTINES

The standard i/o functions provide a portable i/o interface for C programs.

Before you can read or write the data in a file, the file must be opened and a 'stream' associated with it. This task is performed by `fopen` which returns a pointer to the stream for use in subsequent i/o operations. Once a stream is assigned, close the file using `fclose`. 
Note: It is advisable to avoid having more than one stream open to the same file at any point in a program.

Close files when they are no longer in use, since there is a limit to the number of files which may be open at once. Closed files can be reopened.

The precise number of open files allowed depends on the filing system being used. For example:

<table>
<thead>
<tr>
<th>System</th>
<th>Number of Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>5 files</td>
</tr>
<tr>
<td>ADFS</td>
<td>10 files</td>
</tr>
<tr>
<td>NFS</td>
<td>5 files</td>
</tr>
</tbody>
</table>

On a Master 128, you can have files on more than one filing system open simultaneously.

For convenience, three streams are always automatically opened: stdin, stdout and stderr. These are declared in hstdio as follows:

- **FILE *stdin**

  This is the standard input stream. On most systems, including Acornsoft C, stdin is connected to a terminal keyboard by default. The current input device as set by *FX2 is used.

- **FILE *stdout**

  This is the standard output stream. On this and most systems stdout is the display device (VDU or printer) of a terminal. The current output device(s) as set by *FX3 are used.

- **FILE *stderr**

  This is the standard error stream, used by programs for outputting error messages. It, too, is normally the terminal output device.

The standard i/o routines are listed on the following pages.
NAME

fclose – close a stream (smallib)

SYNOPSIS

#include <stdio.h>

int fclose(stream)
FILE *stream;

DESCRIPTION

If the stream is an output stream, then fclose causes any buffered data for the specified stream to be written out and the stream to be closed.

close is performed automatically for all open streams upon calling exit().

close returns 0 if successful. Otherwise it returns EOF indicating that the stream is not associated with a file or there is an error.

EXAMPLE

close(stdout);
NAME

fgetc - get a character from a stream (smallib)

SYNOPSIS

#include <stdio>

int fgetc(stream) FILE *stream;

DESCRIPTION

fgetc returns the next character from the specified stream. It also moves the file pointer (if defined) ahead one character such that successive calls to fgetc return successive characters.

fgetc is identical to getc in the Acornsoft C system.

fgetc returns EOF when attempting to read past the end-of-file or a read error, should one occur. Use feof and ferror to find out whether it is EOF or a read error.

EXAMPLE

int ch; ch = fgetc(stdin);

NAME

fgets - get a string from a stream (smallib)

SYNOPSIS

#include <stdio>

char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;

DESCRIPTION

fgets reads characters from the stream into the array pointed to by s, until:
• n-1 characters are read
• a newline character is read and transferred to s
• an end-of-file condition is encountered.

The last character read into s is followed by a null character. However, if end-of-file is encountered and no characters are read, no characters are transferred to s.

fgets returns s if successful. If, however, no characters are transferred to s (due to the end-of-file condition described above), or a read error occurs, a NULL pointer is returned.

EXAMPLE

#define LEN 20
char string[LEN];
if (fgets(string, LEN, stdin)==NULL)
    puts("Error in reading string");
else
    puts(string);
NAME
fopen — open a stream (smallib)

SYNOPSIS
#include <h.stdio>

FILE *fopen(filename, type)
char *filename, *type;

DESCRIPTION
fopen opens the file named by filename and associates a stream with it.

type is a character string that indicates the type of access for which the file is opened. The following values are possible:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Open for reading in the currently selected directory or in standard places.</td>
</tr>
<tr>
<td>w</td>
<td>Truncate or create for writing.</td>
</tr>
<tr>
<td>a</td>
<td>Append; open for writing at end of file, or create for writing.</td>
</tr>
<tr>
<td>p</td>
<td>Open for reading only in the given path.</td>
</tr>
<tr>
<td>r+</td>
<td>Open for update (reading and writing); a file is created if it does not already exist.</td>
</tr>
<tr>
<td>w+</td>
<td>Truncate or create for update.</td>
</tr>
<tr>
<td>a+</td>
<td>Append; open or create for update at end-of-file.</td>
</tr>
</tbody>
</table>

Passing a NULL filename results in either the keyboard or screen being used depending on whether it is for reading or writing.

fopen returns a pointer used to identify the stream in subsequent operations if successful. Otherwise it returns a NULL pointer and errno is set to indicate the error.

EXAMPLE

FILE *newinput;
newinput=fopen(":/C.HelloW","r");
NAME

fprintf – formatted printing to a stream (stdlib)

SYNOPSIS

#include <stdio.h>

int fprintf(FILE *stream, char *format {, argument} ... )

DESCRIPTION

fprintf writes its output to the specified stream.

fprintf converts, formats and outputs the arguments controlled by its format argument.

A format argument is a character string which contains two types of objects:

- plain characters, which are copied to the output stream
- conversion specifications, each of which causes zero or more arguments to be fetched

If there are insufficient arguments for the format the results are undefined. If the format is exhausted while arguments remain, the excess arguments are ignored.

Each conversion specification is introduced by the character %. Following the % character there may be (in the order given):

- An optional plus sign: The plus sign (+), specifies that the result of a signed conversion always begins with a sign (+ or -).
- An optional digit string: The digit string specifies a minimum field width. If the converted value has fewer characters than the field width it is blank padded on the left (or right if a left justification indicator, the - character, is given) to make up the field width.
- A precision: The precision specifies:
  - the number of digits to appear after the decimal point for e and f format conversion
  - the maximum number of significant digits for g conversion
  - the maximum number of characters to be output from a string in s conversion

The precision takes the form of a full stop (.) followed by a decimal digit string; a NULL digit string is treated as zero.

- An optional l: The l (lower case L), specifies that a following d, o, u, x or X corresponds to a long integer argument. An l before any other conversion character is ignored since in this system a float is the same length as a double. Thus the conversion has no effect.
- A character: The character indicates the type of conversion to be applied.

A field width or precision may be specified as an asterisk (*) instead of a digit string, in which case a corresponding integer argument is used as either the field width or precision.
The conversion characters and their meanings are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, o, u, x, X</td>
<td>The integer argument is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (x and X) respectively. The letters abcdef are used for x conversion and ABCDEF for X conversion. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a NULL string.</td>
</tr>
<tr>
<td>f</td>
<td>The float or double argument is converted to decimal notation in the form (-Jddd, ddd) where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly zero, no decimal point appears.</td>
</tr>
<tr>
<td>e, E</td>
<td>The float or double argument is converted in the style (-Jd, dde \ d), where there is one digit before the decimal point and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced. If the precision is zero, no decimal point appears. The E format code produces a number with E instead of e introducing the exponent. The exponent always contains two digits in this system.</td>
</tr>
<tr>
<td>g, G</td>
<td>The float or double argument is output in style f or e (or in style E in the case of a G format code), whichever gives full precision in minimum space. Trailing zeroes are removed from the result and a decimal point appears only if it is followed by a digit.</td>
</tr>
<tr>
<td>c</td>
<td>The character argument is printed.</td>
</tr>
<tr>
<td>s</td>
<td>The argument is taken to be a string (character pointer) and characters from the string are printed until a NULL character is reached or until the number of characters indicated by the precision specification is reached. However if the precision is zero or missing, all characters up to a NULL are printed. A NULL value for the argument yields undefined results.</td>
</tr>
<tr>
<td>u</td>
<td>The unsigned integer argument is converted to an unsigned decimal number and output.</td>
</tr>
<tr>
<td>%</td>
<td>Prints a % character. No argument is converted.</td>
</tr>
</tbody>
</table>

If the character after the % is not a valid conversion character the results of the conversion are not predictable.

In no case does a non-existent or small field width cause truncation of a field. Padding takes place only if the specified field width exceeds the actual width. Characters generated by fprintf are printed as if by printf.

fprintf returns the number of characters transmitted if successful, or a negative value if an output error is encountered.

**EXAMPLE**

```c
char *errmsg = "multiple definition";
int val;
int *addr = &val;
fprintf(stderr, "***ERROR %s at %.4X", errmsg, addr);
```
NAME

fputc – put a character to a stream (smallib)

SYNOPSIS

#include <h.stdio>

fputc(c, stream)
int c;
FILE *stream;

DESCRIPTION

fputc writes the character c to the specified output stream at
the position where the file pointer, if defined, is pointing. It is
identical to putc in this system.

fputc returns the character written if successful, or EOF if an
error is encountered.

EXAMPLE

FILE *newoutput;
fputc('A', newoutput);

NAME

fputs – put a string to a stream (smallib)

SYNOPSIS

#include <h.stdio>

int fputs(s, stream)
char *s;
FILE *stream;

DESCRIPTION

fputs copies the null terminated string s to the specified
output stream. The null character which terminates the string
is not written to the stream.

fputs returns EOF if an error is encountered.

EXAMPLE

fputs("A string", stdout);
NAME

fread – read n items from a stream (smallib)

SYNOPSIS

#include <stdio.h>
int fread(pointer, size, nitems, stream)
char *pointer;
int size, nitems;
FILE *stream;

DESCRIPTION

Note the following information regarding fread:

- fread copies up to nitems of data from the named input stream into an array pointed to by pointer. Each item of data is a sequence of bytes of length size. The maximum number of items in Acornsoft C is 32767.

- fread stops appending bytes if an end-of-file or error condition is encountered while reading a stream, or if nitems are read.

- fread changes the file pointer (if defined) to point to the byte following the last byte read (if one was read) but does not change the contents of stream.

- fread returns the number of items actually read. If size or nitems is non-positive, no characters are read and zero is returned.

EXAMPLE

FILE *data;
#define BUFLEN 40
char buffer[BUFLEN];
printf("%d items read\n", fread(buffer, sizeof(char), BUFLEN, data));
NAME
freopen – close and reopen a stream (smallib)

SYNOPSIS
#include <stdio>
FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;

DESCRIPTION
freopen closes the current open stream and substitutes the
named file in its place. Type is a character string given as in
fopen.

Passing a NULL filename results in either the keyboard or
screen being used depending on whether it is for reading or
writing.

freopen returns a pointer to the file associated with the
stream if successful. Otherwise it returns a NULL pointer and
errno is set to indicate the error.

EXAMPLE
if (freopen("stdio", "r", stdin) == NULL)
perror("Can't reopen stdin");

LIBRARY ROUTINES
NAME
fscanf – formatted input from a stream (stdlib)

SYNOPSIS
#include <stdio>
int fscanf(stream, format (, pointer) ... )
FILE *stream;
char *format;

DESCRIPTION
fscanf reads characters from the specified input stream,
interprets them according to a format string and stores the
results in the variables pointed to by its arguments. The
format string usually contains conversion specifications which
are used to direct interpretation of input sequences. The
format string may contain:

• white space, which causes white space to be skipped on
input
• an ordinary character other than % which must match the
next character of the input stream
• conversion specifications (following a %), consisting of:
  * an optional assignment suppressing character
  % an optional numerical maximum field width
  l optional (lower case L – see printf)
  h indicating the size of the receiving variable, and a
  conversion code.
  ☐ specify scan set

A conversion specification directs the conversion of the
next input field. The result is placed in the variable
pointed to by the corresponding pointer argument, unless assignment suppression is indicated by ∗. For all descriptors except %, % and e, white space leading an input is ignored. An input field is defined as a string of non-space characters. It extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>A single % character is expected in the input at this point; no assignment is done.</td>
</tr>
<tr>
<td>n</td>
<td>Returns the total number of characters so far which have been scanned from the beginning of the scan. The corresponding argument should be an integer pointer.</td>
</tr>
<tr>
<td>d</td>
<td>A decimal integer is expected. The corresponding argument should be an integer pointer.</td>
</tr>
<tr>
<td>u</td>
<td>An unsigned decimal integer is expected. The corresponding argument should be an unsigned integer pointer.</td>
</tr>
<tr>
<td>o</td>
<td>An octal integer is expected. The corresponding argument should be an integer pointer.</td>
</tr>
<tr>
<td>x</td>
<td>A hexadecimal integer is expected. The corresponding argument should be an integer pointer.</td>
</tr>
<tr>
<td>e, f, g</td>
<td>A character string is expected. The corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating NULL character which is added. The input field is terminated by white space.</td>
</tr>
<tr>
<td>i</td>
<td>It is used for general integer conversion. It reads the input as an integer using C rules for conversion, e.g. 12 is read as 12 (decimal), 012 is read as 10 (decimal) and 0x12 as 18 (decimal). The corresponding argument should be an integer pointer.</td>
</tr>
<tr>
<td>s</td>
<td>A character is expected. The corresponding argument should be a character pointer. The normal skip over space character is suppressed in this case. To read the next non-space character, use %Is. If a field width is given, the corresponding argument should point to a character array, and the indicated number of characters is read.</td>
</tr>
<tr>
<td>e, f, g</td>
<td>A floating point number is expected. The next field is converted accordingly and stored through the corresponding argument which should be a pointer to a float. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e followed by an optionally signed integer exponent.</td>
</tr>
<tr>
<td>Character</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| [ | Indicates a string not to be delimited by space characters. The left bracket is followed in the format string by a set of characters and a right bracket. The characters between the brackets define a set of characters called the scanset. If the first character is not a circumflex (\^), the scanset consists of all the characters until the first character not in the set between the brackets. If the first character after the left bracket is ^, the scanset consists of all characters not contained in the remainder of the scanset string. The corresponding argument must point to a character array large enough to hold the data field and the terminating NULL character which is added automatically. Note that if the first character after the [ (or ^ if present) is a J, this forms part of the scanset; eg [^JJ matches J.

The conversion characters d, o, u and x may be preceded by l (lower case L) or h to indicate that a pointer to long or short rather than to int is in the argument list. Similarly, the conversion characters e, f and g may be preceded by l (lower case L) to indicate a pointer to double rather than float. This has no effect in this system. The L or h modifier is ignored for other conversion characters.

fscanf conversion terminates at end-of-file at the end of the control string, or when an input character conflicts with the control string. In the latter case the offending character is left unread.

fscanf returns the number of successfully matched and assigned input items. This is used to decide how many input items were found. The constant EOF is returned on end of input if EOF occurs while reading the input. Note that this is different from zero, which means that no conversion is done. If conversion is intended, it is frustrated by an inappropriate character in the input.

**EXAMPLE**

```c
int decnum;
long hexval;
char string[40];
fscanf(stdin, "%d %lx %s", &decnum, &hexval, string);
```
NAME

fseek – set the pointer on a stream (smallib)

SYNOPSIS

#include <h.stdio>
#include <h.fcntl>

int fseek(stream, offset, ptrname)
FILE *stream;
Long offset;
int ptrname;

DESCRIPTION

fseek sets the position of the next input or output operation on the stream. The new position is at the signed distance offset bytes:

• from the beginning
• from the current position
• from the end of the file

The position chosen depends on whether ptrname has the value SEEK_SET, SEEK_CUR or SEEK_END.

fseek undoes any effects of ungetc.

fseek returns non-zero for improper seeks. Otherwise it returns zero. An improper seek can be, for example, an fseek done on a stream that is not opened using fopen.

EXAMPLE

fseek(stream, offset, SEEK_CUR);

NAME

ftell – read the pointer of a stream (smallib)

SYNOPSIS

#include <h.stdio>

long ftell(stream)
FILE *stream;

DESCRIPTION

ftell returns the current value of the offset (in bytes) relative to the beginning of the file associated with the named stream.

EXAMPLE

FILE *data;
long offset;
offset=ftell(data);
NAME

fwrite — write items to a stream (smallib)

SYNOPSIS

#include <stdio.h>

int fwrite (pointer, size, nitems, stream)
  char *pointer;
  int size, nitems;
  FILE *stream;

DESCRIPTION

fwrite appends up to nitems of data to a specified output stream from an array pointed to by pointer. An item of data is a sequence of bytes of length size. fwrite stops appending when nitems of data are appended or if an error condition is encountered on stream. fwrite increments the file pointer on stream by the number of bytes written but does not change the contents of the array pointed to by pointer.

fwrite returns the number of items written, if size or nitems is non-positive, no characters are written and zero is returned.

EXAMPLE

char array[] = "ABCDEFGH";
fwrite(array, 1, strlen(array), stdout);

NAME

gets — get a string from the standard input stream (smallib)

SYNOPSIS

#include <stdio.h>

char *gets(s)
  char *s;

DESCRIPTION

gets reads a string into the array pointed to by s from the standard input stream stdin.

gets reads characters until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

gets returns its argument s if successful. If end-of-file is encountered with no characters read, no characters are transferred to s and a NULL pointer is returned. If a read error occurs a NULL pointer is returned.

EXAMPLE

#define LEN 20
char string[LEN];
if (gets(string)==NULL)
  perror("Error in reading string");
else
  puts(string);
NAME

getw - get a word from a stream (smallib)

SYNOPSIS

#include <stdio.h>

int getw(FILE *stream);

DESCRIPTION

getw returns the next word (integer) from the named input stream.

getw increments the associated file pointer, if defined, to point to the next word so that successive calls to getw return successive words from the stream.

Note that getw is machine-dependent, as the size of a word varies from one machine to another.

EOF is returned at end-of-file or when a read error is detected. However, check for feof and perror error conditions since EOF is a valid integer.

EXAMPLE

int fred;
fred = getw(stdin);

NAME

perror - print message and last system error (stdlib)

SYNOPSIS

void perror(char *s);

extern int errno;

DESCRIPTION

perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function. The argument string s is printed first, then a colon and a blank, followed by the message and a newline. The error number is taken from the external variable errno, which is set when errors occur but is not cleared when non-erroneous calls are made.

If a NULL pointer is given, no output takes place and a pointer to the error string is returned.

EXAMPLE

if ((newinput=fopen(":0.C.HelloW","r"))==NULL)
perror("Open failed");
NAME
printf — formatted printing to the standard output stream (stdlib)

SYNOPSIS
#include <stdio>

int printf(format {, argument} ...)
char *format;

DESCRIPTION
printf performs like fprintf with one exception. It writes output to the standard output stream, stdout.

EXAMPLE
int hexval= 0xff;
char *string= "Hello World"
printf("Results: %.4X, '%s', \n", hexval, string);

NAME
puts — put a string to the standard output stream (smallib)

SYNOPSIS
#include <stdio>

int puts(s)
char *s;

DESCRIPTION
puts copies the null terminated string pointed to by s to the standard output stream, stdout, followed by a newline character. The terminating null character is not copied.

EXAMPLE
puts("Hello World");
NAME

putw – put a word to a stream (smallib)

SYNOPSIS

#include <h.stdio>

int putw(w, stream)
int w;
FILE *stream;

DESCRIPTION

putw outputs a word (ie integer) to the output stream at the position at which the file pointer, if defined, is pointing.

putw neither assumes nor causes special alignment in the file.

Note that putw is machine-dependent, as the size of a word varies from one machine to another.

putw returns the word written if successful, or EOF if a write error is encountered.

EXAMPLE

putw(0xFFEE, stdout);

rewind – reset the file pointer on a stream (milib)

SYNOPSIS

#include <h.stdio>

void rewind(stream)
FILE *stream;

DESCRIPTION

rewind repositions the stream to the first byte of the associated file (byte 0). It is equivalent to fseek (stream, 0L, SEEK_SET), except that no value is returned.

EXAMPLE

rewind(stderr);
NAME
scanf - formatted input from the standard input stream (stdlib)

SYNOPSIS
#include <hstdio>
scanf(format {, pointer} ...)
char *format;

DESCRIPTION
scanf reads input from the standard input stream stdin. The characters it reads are interpreted according to the given format and the resulting values stored in the locations pointed to by the pointer arguments.

The meaning of the arguments to scanf is the same as fscanf.

scanf returns the number of successfully matched and assigned input items or EOF on end of input.

EXAMPLE
int var, var2;
char *formstr = "%d %d %s"
printf("%d items converted\n", scanf(formstr, &var, &var2, str));

NAME
sprintf - formatted output to a string (stdlib)

SYNOPSIS
#include <hstdio>
int sprintf(s, format {, argument} ...)
char *s, *format;

DESCRIPTION
sprintf writes formatted output into a character array via a pointer s supplied by the caller. The format meaning and the argument values are as for fprintf. The output string is automatically terminated by a NULL character.

sprintf returns the number of characters output (not including the NULL character) if successful, or EOF if an output error is encountered.

EXAMPLE
#define LEN 20
char *filename = "Cprogram"
char *buffer [LEN];
sprintf(buffer, "c.%s\n", filename);
NAME

scanf – formatted input from a string (stdlib)

SYNOPSIS

#include <stdio>

int sscanf(s, format {, pointer} ...)
char *s, *format;

DESCRIPTION

scanf reads input from the strings. It interprets the characters it reads according to the given format string and stores the resulting values in the locations pointed to by the pointer arguments. The meaning of the arguments to sscanf is the same as for fscanf.

scanf returns the number of successfully matched and assigned input items or EOF on end of string.

EXAMPLE

char c1; char c2;
char*read_string = "ab cdefg";
char str[412];
scanf(read_string, "%c%c%s", &ch1, &ch2, str);

NAME

ungetc – push a character back onto a stream

SYNOPSIS

#include <stdio>

int ungetc(c, stream)
int c;
FILE *stream;

DESCRIPTION

ungetc pushes the character c back onto an input stream. That character is returned by the next getc call on that stream. One character of push-back is guaranteed provided something was read from the stream. Attempts to push EOF are rejected. Pushed back characters are lost on fseek.

ungetc returns c if successful, or EOF if it cannot push a character back.

EXAMPLE

char nextch;
FILE *input = stdin;
ungetc(nextch,input);
NAME

clearerr - reset the error and end-of-stream conditions
(macro)

SYNOPSIS

#include <stdio>
#define clearerr(p)

DESCRIPTION

clearerr resets any error or EOF indication on the named
stream.

EXAMPLE

clearerr(stderr);

NAME

creat - create a new file / rewrite existing file (macro)

SYNOPSIS

#include <stdio>
#include <fcntl>
#define creat(path,mode) open(path,(mode):0_CREAT:0_TRUNC)

DESCRIPTION

creat creates a new file or deletes the contents of an existing
one, preparing it to be written to.

creat returns a file descriptor if successful, and opens the file
for writing. Otherwise a value of -1 is returned and errno is set
to indicate the error.

The values for mode are as follows:

#define 0_RDONLY
#define 0_RDWR
#define 0_APPEND
#define 0_TRUNC
#define 0_EXCL
#define 0_NPATH
#define 0_NCSOD
#define 0_RDONLY

These are described under the library routine open.

EXAMPLE

int fildes;
fildes = creat("maketmp", 0_WRONLY);
NAME

feof - test end-of-stream condition (macro)

SYNOPSIS

#include <stdio>
#define feof(p)

DESCRIPTION

feof returns non-zero if end-of-file has been detected reading the named input stream (note it may not be currently at the end). Otherwise it returns zero.

EXAMPLE

FILE *stream; int ch;
while (feof(stream)==0) ch=getc(stream);
NAME

fileno — file descriptor for stream (macro)

SYNOPSIS

#include <stdio>
#define fileno(p)

DESCRIPTION

fileno returns the Acornsoft C filing system's handle which is associated with the named stream.

EXAMPLE

int han = fileno(stdin);
NAME

getchar – get a character from the standard input stream (macro)

SYNOPSIS

#include <h.stdio>
#define getchar()

DESCRIPTION

getchar returns the next character from the standard input stream stdin. It is equivalent to getc(stdin).

getchar returns EOF at end-of-file or when a read error is detected. Use feof or ferror to test for this.

EXAMPLE

int ch = getchar();
NAME

putchar — put a character to the standard output stream
(macro)

SYNOPSIS

#include <h_stdio>
#define putchar(c)

DESCRIPTION

putchar writes the character c to the standard output stream, stdout, at the position where the file pointer, if defined, is pointing. It is equivalent to putc(c, stdout).

putchar returns the character written if successful or EOF if an error is encountered.

EXAMPLE

putchar(\n);
NAME
close – close a file (minlib)

SYNOPSIS
int close(int fd);

DESCRIPTION
close breaks the connection between a file descriptor and the
file itself, i.e., given a file descriptor, for example fd as
returned by open or creat, close closes the associated file.

close returns zero if successful. Otherwise, a value of -1 is
returned and errno is set to indicate the error.

The argument corresponds to the BBC OS file handle passed
to OSFILE. This means that close (0) closes all currently open
BBC files.

EXAMPLE
int close(int fd);

NAME
lseek – set pointer for a file (minlib)

SYNOPSIS
#include <fcntl>

long lseek(int fd, off_t off, int whence);

DESCRIPTION
The file descriptor refers to a file open for reading and/or
writing. The read/write pointer for the file is set as follows:

• If whence is SEEK_SET, the pointer is set to offset bytes from
  start of file.

• If whence is SEEK_CUR, the pointer is set to its current
  location plus offset.

• If whence is SEEK_END, the pointer is set to the size of the
  file plus offset.

• lseek returns the resulting pointer value if successful, if not
  -1 is returned and errno is set to indicate the error.

EXAMPLE
lseek(int fd, off_t off, int whence);
NAME
open – open a file (minlib)

SYNOPSIS
#include <h.fcntl>
int open(path, oflag)
char *path;
int oflag;

DESCRIPTION
path points to a pathname identifying a file. open opens a file descriptor for the identified file and sets the file status flags according to the value of oflag.

oflag values are constructed by “or-ing” together flags from the following list:

Flag   Description
O_WRONLY open for writing only (only one of these is allowed).
O_RDONLY open for reading only (only one of these is allowed).
O_CREAT if the file exists this flag has no effect otherwise a new file will be created.
O_APPEND if set the file pointer will be set to the end of file prior to each write.
O_TRUNC if the file exists, its length is truncated to zero.
O_EXCL if 0_EXCL and 0_CREAT are set, open fails if the file exists.
O_NPATH open, but do not look down the system path for the file.
O_NCSĐ open, but do not use currently selected directory for the file.

The file pointer used to mark the current position within the file is set to the beginning of the file.

open returns the file descriptor if successful, otherwise a value of –1 is returned and errno is set to indicate the error.

EXAMPLE
int han = open("h.fcntl", O_RDONLY | O_NCSĐ);
NAME

read – read bytes from a file (minlib)

SYNOPSIS

```c
int read(int fildes, char *buf, int nbyte);
```

DESCRIPTION

`fildes` is a file descriptor obtained from a `create` or `open` system call. `read` attempts to read `nbyte` bytes from the file associated with `fildes` into the buffer pointed to by `buf`.

Note the following points regarding `read` and seeking devices:

- **Devices capable of seeking**: These devices start reading at a position in the file given by the file pointer associated with `fildes`. When `read` finishes, the file pointer is incremented by the number of bytes actually read.

- **Devices incapable of seeking**: These devices always read from the current position.

`read` returns the number of bytes actually read and placed in the buffer. This number may be less than `nbyte` if the number left in the file is less than `nbyte` bytes.

A value of zero is returned if the input is at end-of-file at the call.

A value of -1 is returned if `read` fails and `errno` is set to indicate the error.

EXAMPLE

```c
#define BUFSIZE 128
char buffer[BUFSIZE];
switch(read(fileno(stdin), buffer, BUFSIZE))
{
    case -1: perror("Read failed"); break;
    case 0: puts("EOF in read"); break;
    default: puts("Read successful"); break;
}
```
NAME

unlink - delete a file (minlib)

SYNOPSIS

int unlink(path)
char *path;

DESCRIPTION

unlink deletes the file named by path.
unlink returns zero if successful, otherwise a value of -1 is
returned and errno is set to indicate the error.

EXAMPLE

unlink("tmpfile");

NAME

write - write bytes to a file (minlib)

SYNOPSIS

int write(fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

fildes is a file descriptor obtained from a creat or open
system call. write attempts to write nbyte bytes from the
buffer pointed to by buf to the file associated with fildes.

Note the following points regarding fildes and random
access:

• Filing systems capable of random access: These systems write
data to the position in the file indicated by the file pointer.
When write finishes, the file pointer is incremented by the
number of bytes actually written.

• Filing systems incapable of random access: These systems
always begin writing at the current position.

write returns the number of bytes actually written. If
unsuccessful (for example if there is not enough room for all
the bytes to be written), write returns a value of -1 and errno
is set to indicate the error.

EXAMPLE

write(fileno(stdout), buffer, BUFSIZE);
SYSTEM CALLS

System calls help perform various useful tasks, such as halting program execution, non-local jumps, issuing operating system commands etc.

Acornsoft C system calls are discussed on the following pages.

NAME

call – call machine code, returning AXYP (minlib)

SYNOPSIS

#include <osdefs>

long call(addr, a, x, y, arg1, arg2, ...)
int addr, a, x, y, arg1, arg2, ...

DESCRIPTION

call calls the 6502 machine code at machine address addr.
The 6502 registers are taken from the corresponding arguments on the stack, P being undefined. Not all the arguments need to be supplied and are undefined if omitted.

Arguments are pushed on backwards in this implementation, and the user's code is entered with _iacc pointing to arg1, assuming int arg1 (ie _iacc = &arg1).

When the call returns, the 6502 registers P, Y, X, A are put into the C result (as if USR is called in BASIC). Any BRK occurring in this code is trapped by the run time system. errno and _sysmsg are set and the result of the function is undefined.

EXAMPLE

printf("code returns %.8X\n", call(my_code, 0xFF, 0, 1, v, &w));
call(code2, 4);
NAME
oscalt – call machine code, with BRKs indicated (minlib)

SYNOPSIS
#include <h.osdefs>
void oscalt(blkptr, addr, a, x, y)
    int addr, a, x, y;
    struct _osret *blkptr;

DESCRIPTION
oscalt calls a block of 6502 machine code at the machine
address, addr. The 6502 registers are taken from the
corresponding arguments on the stack, P being undefined.
Not all the arguments need to be supplied and are undefined if
omitted.

When the call returns, the 6502 registers P,Y,X,A are put into
the _osret struct whose address is passed in blkptr. The error
flag in the struct is set (non-zero) if a BRK occurs in the call and
is clear (zero) otherwise. Any BRK occurring in this code is
trapped by the run time system, setting errno and _sysmsg[].

EXAMPLE
oscalt(&osblk, OSFILE, 5, fileblk, fileblk>>8);
if (osblk.error) perror("Error in OSFILE");

NAME
_exec – pass line to C system CLI (minlib)

SYNOPSIS
int _exec(string)
    char *string;

DESCRIPTION
_exec passes the (null terminated) string to the C system
command line interpreter as if it had been typed at the
command line by the user.

_exec never returns in this implementation. It should be
assumed, however, that _exec does return for future
expansion.

EXAMPLE
_exec("Mode 135");
NAME

_exit — return to C system (minlib)

SYNOPSIS

void _exit(status)
    int status;

DESCRIPTION

_exit returns control back to the C system. If status is non-zero the message Exit (nnn) is printed as a signed decimal integer to the console.

_exit never returns.

EXAMPLE

_exit(-1);

NAME

access — read the attributes of a file (system)

SYNOPSIS

#include <fcntl.h>

int access(path, amode)
    char *path;
    int amode;

DESCRIPTION

access checks a file for accessibility according to the bit pattern contained in amode. The file checked by access is defined by path. path points to a pathname naming the file. The value of amode is the sum of the access modes to be checked as defined in <fcntl.h>:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_OK</td>
<td>owner read</td>
</tr>
<tr>
<td>W_OK</td>
<td>owner write</td>
</tr>
<tr>
<td>PR_OK</td>
<td>public read (BBC specific)</td>
</tr>
<tr>
<td>PW_OK</td>
<td>public write (BBC specific)</td>
</tr>
<tr>
<td>L_OK</td>
<td>checks if file is locked for owner</td>
</tr>
<tr>
<td>PL_OK</td>
<td>checks if file is locked for public</td>
</tr>
<tr>
<td>F_OK</td>
<td>checks existence of file</td>
</tr>
</tbody>
</table>

If the access requested is permitted, a value of zero is returned. Otherwise a non-zero value is returned and errno is set to indicate the error.

EXAMPLE

int present = access(filename, F_OK);
int can_write = access(filename, W_OK);
NAME

cdir – change currently selected directory (system)

SYNOPSIS

int cdir(path)
char *path;

DESCRIPTION

cdir causes a directory pointed to by path to become the
current working directory. It is equivalent to *DIR <path>.

cdir returns zero if successful. Otherwise a non-zero value is
returned and errno is set to indicate the error.

EXAMPLE

if (cdir("$C")!=0) perror("Trouble");

NAME

chmod – set the attributes of a file (system)

SYNOPSIS

#include <fcntl>

int chmod(path, mode)
char *path;
int mode;

DESCRIPTION

cmod sets the access attributes of a file according to the bit
pattern contained in mode. path points to a pathname naming
the file chmod identifies.

Access permission bits are described in <fcntl>.

cmod returns zero if successful. Otherwise a non-zero value is
returned and errno is set to indicate the error.

EXAMPLE

chmod("MailBox", R_OK + W_OK);
NAME

escape_action – alter / check the escape status (minlib)

SYNOPSIS

#include <h.escape>

int escape_action(s)

int s;

DESCRIPTION

escape_action has the following effects on the escape function, depending on the value of s:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENA_ESC</td>
<td>Enable escape.</td>
</tr>
<tr>
<td>DIS_ESC</td>
<td>Disable escape.</td>
</tr>
<tr>
<td>TST_ESC</td>
<td>Return zero if the internal escape flag is clear and non-zero is set</td>
</tr>
<tr>
<td>CLR_ESC</td>
<td>Clear the internal escape flag.</td>
</tr>
</tbody>
</table>

If escape is enabled, pressing [ESCAPE] during execution of a C program interrupts execution of the program. Enable is the default state.

If escape is disabled, pressing [ESCAPE] during execution of a program causes an internal escape flag to be set without interrupting execution of the program.

EXAMPLE

escape_action(DIS_ESC);
if (escape_action(TST_ESC))
{
    puts("Escape pressed");
    escape_action(CLR_ESC);
}
NAME

exec - run another C program, passing arguments (system)

SYNOPSIS

#include <stdio>

int execl(path, arg0, arg1, ..., argv, NULL
char *path, *arg0, *arg1, ..., *argv;

DESCRIPTION

execl builds up a command string which it sends to the C command line interpreter.

execl never returns in this system, but for compatibility it should be assumed that it may.

EXAMPLE

execl ("link", "file1", NULL);

NAME

exit - quit, closing streams (minlib)

SYNOPSIS

void exit(status);
int status;

DESCRIPTION

exit is the normal means of terminating program execution.
Its functions include:

• closing all open streams
• performing various clean-up actions
• calling _exit

exit never returns.

EXAMPLE

exit(returncode);
NAME

longjmp - jump to previously saved position (minlib)

SYNOPSIS

#include <h.setjmp>
void longjmp(env, val)
jmp_buf env;
int val;

DESCRIPTION

longjmp restores the environment with the corresponding env argument saved by the last call of setjmp.

After longjmp is completed, program execution continues as if the corresponding call of setjmp returns the value val. If longjmp cannot cause setjmp to return the value zero. If longjmp is invoked with a second argument of zero, setjmp returns 1.

EXAMPLE

jmp_buf savedpos;
longjmp(savedpos, 42);

NAME

seterror - set the system error message and number (minlib)

SYNOPSIS

void seterror(errnum, errstr)
int errnum;
char *errstr;

DESCRIPTION

seterror sets errno equal to errnum and copies errstr (a null terminated string) into _sysmsg[].

seterror is especially useful for lower-level routines that want to set system errors for other routines to notice.

EXAMPLE

#include <h.errno>
seterror(EINTR, "Escape!");
NAME

setjmp – save a position for a later longjmp (minlib)

SYNOPSIS

#include <h.setjmp>

int setjmp(env)
jmp_buf env;

DESCRIPTION

setjmp saves its stack environment in env, whose type
jmp_buf is defined in the <h.setjmp> header file. It is then
available for later use by longjmp.

setjump returns the value zero when first called.

EXAMPLE

jmp_buf savedpos;
if (setjmp(savedpos)!=0) return;

NAME

setpath – set the system file search path (minlib)

SYNOPSIS

int setpath(path)
char *path;

DESCRIPTION

setpath sets the given path as the C system path:
• to use for file operations
• in the command line interpreter

Any spaces are stripped out.

setpath returns zero if successful. Otherwise:
• -1 is returned
• the previous path is unaffected
• errno is set to indicate the error

EXAMPLE

setpath("-net-$Clib,$C");
NAME

showpath - return a pointer to the system file search path
(minlib)

SYNOPSIS

char *showpath()

DESCRIPTION

showpath returns a pointer to the current C system path. Note
the following regarding individual path elements:

- they are delimited by commas
- they are terminated by null
- they may be obtained using strtok()

Examples of valid paths are:

```
".
"\$C"
"$-compiler,-net-$Library.$, -linker"
```

EXAMPLE

strcpy(my_path, showpath());
/* where my_path[160] has been declared */

NAME

system - pass a string to the machine OSCLI (minlib)

SYNOPSIS

int system(string)
char *string;

DESCRIPTION

system causes a string to be given to the machine command
line interpreter.

Errors, such as bad commands, cause -1 to be returned with
the error in errno. Otherwise zero is returned.

EXAMPLE

system("tv0 1");
NAME

trace – turn function tracing on/off (minlib)

SYNOPSIS

void trace(flag)
int flag;

DESCRIPTION

trace turns function tracing on and off as follows:

• Flag is non-zero: function tracing is turned on
• Flag is zero: function tracing is turned off

Turning tracing on and off in this way is equivalent to issuing
the command TRACE at command level.

EXAMPLE

trace(1);
**NAME**

malloc – allocate space on heap (system)

**SYNOPSIS**

```c
#include <h.malloc>

char *malloc(size)
unsigned size;
```

**DESCRIPTION**

malloc returns a pointer to a block of at least size bytes or zero if there is not enough available memory.

Undefined results occur if the space assigned by malloc is overrun.

**EXAMPLE**

```c
char *buffer;
if ((buffer=malloc(1200)) == NULL) puts("Ran out of memory");
```

**NAME**

calloc – allocate space on heap and clear to zero (system)

**SYNOPSIS**

```c
#include <h.malloc>

char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

**DESCRIPTION**

calloc allocates space for an array of nelem elements of size elsize. The space is initialised to zeros.

calloc returns a pointer to the block allocated, or NULL if there is not enough available memory.

**EXAMPLE**

```c
char *buffer = (char *)calloc(32, sizeof(long));
```
NAME
free – free an allocated area to the heap (system)

SYNOPSIS
#include <h.malloc>

void free(ptr)
char *ptr;

DESCRIPTION
The argument to free is a pointer to a block previously
allocated by malloc.

After free is performed this space is made available for further
allocation.

Undefined results occur if the pointer does not point to a
block of memory allocated using malloc/calloc.

EXAMPLE
free(memory);

NAME
mallinfo – return information on heap allocation (system)

SYNOPSIS
#include <h.malloc>

struct _mallinfo *mallinfo();

DESCRIPTION
mallinfo provides information describing space usage. It
returns the structure _mallinfo which includes the following
members:

• unsigned maxblk: The largest piece of heap memory
  available.

• unsigned free: Total free space on heap memory.

• unsigned nblocks: The number of blocks on the heap
  (excludes main memory block).

This routine is Acornsoft C specific and does not conform to
X/OPEN. This is because the heap allocator on the Acornsoft
C system does not support XOPEN allocation modes such as
fast blocks etc.
MEMORY OPERATIONS

The following functions operate on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memory functions are discussed on the following pages.

NAME

memcpy — move memory up to character (system)

SYNOPSIS

#include <string.h>

char *memcpy(s1, s2, c, n)
char *s1, *s2;
int c, n;

DESCRIPTION

memcpy copies characters from memory address s2 into memory address s1. It stops:
• after the first occurrence of character c is copied, or
• after n characters are copied

whichever comes first.

memcpy returns a pointer to the character after the copy of c in s1, or a NULL pointer if c is not found in the first n characters of s2.

EXAMPLE

memcpy(blk1, blk2, 'A', blklen);
NAME

`memchr` – find character in memory (system)

SYNOPSIS

```
#include <string>

char *memchr(s, c, n)
char *s;
int c, n;
```

DESCRIPTION

`memchr` returns a pointer to the first occurrence of character c in the first n characters of memory areas s, or a NULL pointer if c does not occur.

EXAMPLE

```
char *foundat = memchr(start, 'Z', (int)(end-start));
if (foundat!=NULL) printf("Char found at %d\n", foundat-start);
```

NAME

`memcmp` – compare memory (system)

SYNOPSIS

```
#include <string>

int memcmp(s1, s2, n)
char *s1, *s2;
int n;
```

DESCRIPTION

`memcmp` compares the character arrays pointed to by s1 and s2, looking at the first n characters only.

`memcmp` returns an integer less than, equal to, or greater than zero, depending on whether s1 is lexicographically less than, equal to, or greater than s2.

EXAMPLE

```
if (memcmp(blk1, blk2, blklen)==0) puts("Same");
```
NAME

memcpy - copy block of memory (system)

SYNOPSIS

#include <string>

char *memcpy(s1, s2, n)
char *s1, *s2;
int n;

DESCRIPTION

memcpy copies n characters from memory area s2 to memory area s1. It returns s1. 

memcpy works if the areas of memory are overlapping, although this should not be relied upon if programs are to be used with other C systems.

EXAMPLE

memcpy(blk1, blk2, blklen);

NAME

memset - set block of memory to character (system)

SYNOPSIS

#include <string>

char *memset(s, c, n)
char *s;
int c, n;

DESCRIPTION

memset sets the first n characters in memory area s to the value of character c. It returns s.

EXAMPLE

memset(start, 0xEA, end-start));
STRING OPERATIONS

The string functions take pointers to strings as arguments. A string is an array of characters terminated by a null character.

The functions `strcat`, `strncat`, `strcpy` and `strncpy` alter the array pointed at by `s1`. They do not check for overflow of the array pointed at by `s1`.

String functions are discussed on the following pages.

NAME

`strcat` – concatenate strings (system)

SYNOPSIS

```c
#include <string.h>

char *strcat(s1, s2)
char *s1, *s2;
```

DESCRIPTION

`strcat` appends a copy of string `s2` to the end of string `s1`. `s1` is returned.

EXAMPLE

```c
strcat(buffer, filename);
```
NAME

strchr — search for character in string (system)

SYNOPSIS

#include <string.h>

cchar *strchr(s, c)

cchar *s;

tint c;

DESCRIPTION

strchr returns a pointer to the first occurrence of character c in string s, or a NULL pointer if c does not occur in the string. The null character terminating a string is considered part of the string.

EXAMPLE

if (strchr(filename, ':') == NULL) puts("No drive specified");

NAME

strcmp — compare two strings (system)

SYNOPSIS

#include <string.h>

int strcmp(s1, s2)

cchar *s1, *s2;

DESCRIPTION

strcmp compares its arguments and returns an integer greater than, equal to, or less than zero, depending on whether s1 is lexicographically greater than, equal to or less than s2.

EXAMPLE

if (strcmp(srcname, dstname)==0)
    puts("ERROR — filenames identical");
NAME
strcpy – copy a string (system)

SYNOPSIS
#include <string>
char *strcpy(s1, s2)
char *s1, *s2

DESCRIPTION
strcpy copies string s2 to s1, stopping after the null character is copied. s1 is returned.

EXAMPLE
strcpy(runname, argv[0]);

NAME
strcspn – length of string not made of given characters (system)

SYNOPSIS
#include <string>
int strcspn(s1, s2)
char *s1, *s2;

DESCRIPTION
strcspn scans s1 character by character until a null or a character in s2 is encountered. The number of characters scanned is returned.

EXAMPLE
char delims[] = " ,;"
int firstlen = strcspn(string, delims);
NAME

strlen – calculate length of string (system)

SYNOPSIS

#include <string>

int strlen(s)
  char *s;

DESCRIPTION

strlen returns the number of characters in s, not including the terminating null character.

EXAMPLE

int length = strlen(string);

NAME

strncat – add (part of) string to string (system)

SYNOPSIS

#include <string>

char *strncat(s1, s2, n)
  char *s1, *s2;
  int n;

DESCRIPTION

strncat appends a copy of string s2 to the end of string s1. It copies at most n characters. A pointer to the null terminated result is returned.

EXAMPLE

strncat(hellostring, "World", 6);
NAME
strncpy — copy (parts of) strings (system)

SYNOPSIS
#include <string>
char *strncpy(s1, s2, n)
char *s1, *s2;
int n;

DESCRIPTION
strncpy copies n characters of string s2 to s1, truncating s2 or adding null characters to s1 if necessary. The result is not null terminated if the length of s2 is n or more. It returns s1.

EXAMPLE
strncpy(buffer, stringtocopy, buflen-strlen(buffer));

NAME
strpbrk — search for set of characters in string (system)

SYNOPSIS
#include <string>
char *strpbrk(s1, s2)
char *s1, *s2;

DESCRIPTION
strpbrk returns a pointer to the first occurrence in string s1 of any character from string s2, or a NULL pointer if no character from s2 exists in s1.

EXAMPLE
char delim[] = ",";
int ptr = strpbrk(showpath(), delim);
NAME

strrchr – search for character in string (system)

SYNOPSIS

#include <string>

char *strrchr(s, c)
char *s;
int c;

DESCRIPTION

strrchr returns a pointer to the last occurrence of character c in string s, or a NULL pointer if c does not occur in the string. The null character terminating a string is considered to be part of the string.

EXAMPLE

int obscurelength = (strrchr(string, '\0')-string);

NAME

strspn – length of string made of given characters (system)

SYNOPSIS

#include <string>

int strspn(s1, s2)
char *s1, *s2;

DESCRIPTION

strspn returns the length of the initial segment of string s1 which consists entirely of characters from string s2.

EXAMPLE

int hexlen = strspn(string, "ABCDEFabcdef0123456789");
NAME

strtok – break up string into ‘tokens’ (system)

SYNOPSIS

#include <h.string>

char *strtok(s1, s2)
char *s1, *s2;

DESCRIPTION

Note the following information regarding strtok:

- **strtok** considers strings s1 to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string s2.

- The first call (with pointer s1 not NULL) returns a pointer to the first character of the first token, and writes a NULL character into s1 immediately following the returned token.

- **strtok** keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with a NULL pointer as the first argument) work through the string s1 that immediately follows that token. In this way subsequent calls work through the string s1 until no tokens remain.

- The separator string s2 may be different from call to call.

- When no token remains in s1, a NULL pointer is returned.

EXAMPLE

char sentence[] = "I came; I saw; I conquered."
char delims[] = ";,;
char *temp, *sentence, *ptr;
int i = 0;

while ((ptr=strtok(temp, delims)) != NULL)
{
    printf("Word %d is '%s' \n", ++i, ptr);
    temp=NULL;
}
CHARACTER CLASSIFICATION

These are macros which classify character-coded integer values. Each macro is a predicate returning non-zero for true and zero for false. \texttt{isascii} is defined on all integer values. The rest are defined only where \texttt{isascii} is true and on the single non-ASCII value \texttt{EOF}.

Character classification functions are discussed on the following pages:

NAME

\texttt{isalpha} – test alphabetic (macro)

SYNOPSIS

\texttt{#include <ctype>}

\texttt{#define isalpha(c)}

DESCRIPTION

\texttt{isalpha} returns a non-zero value if the argument \texttt{c} is a letter. If the argument \texttt{c} is not a letter, it returns zero. If \texttt{c} is not an ASCII character or \texttt{EOF}, the result is undefined.
NAME
isupper – test upper case (macro)

SYNOPSIS
#include <ctype>
#define isupper(c)

DESCRIPTION
isupper returns a non-zero value if the argument c is a lower case letter. If the argument c is not a lower case letter, it returns zero. If c is not an ASCII character or EOF, the result is undefined.

NAME
islower – test lower case (macro)

SYNOPSIS
#include <ctype>
#define islower(c)

DESCRIPTION
islower returns a non-zero value if the argument c is a lower case letter. If the argument c is not a lower case letter, it returns zero. If c is not an ASCII character or EOF, the result is undefined.
NAME

isdigit – test digit 0..9 (macro)

SYNOPSIS

#include <ctype>

#define isdigit(c)

DESCRIPTION

isdigit returns a non-zero value if the argument c is a digit. If the argument c is not a digit, it returns zero. If c is not an ASCII character or EOF, the result is undefined.

NAME

isxdigit – test hexadecimal digit 0..9,a..f,A..F (macro)

SYNOPSIS

#include <ctype>

#define isxdigit(c)

DESCRIPTION

isxdigit returns a non-zero value if the argument c is a hexadecimal digit (0..9,A..F or a..f). If the argument is not a hexadecimal digit, it returns zero. If c is not an ASCII character or EOF, the result is undefined.
NAME

isalnum - test alphanumeric (macro)

SYNOPSIS

#include <ctype>

#define isalnum(c)

DESCRIPTION

isalnum returns a non-zero value if the argument c is a letter or a digit. If the argument c is not a letter or a digit, it returns zero. If c is not an ASCII character or EOF, the result is undefined.

DESCRIPTION

isspace returns a non-zero value if the argument c is a space, horizontal or vertical tab, newline, linefeed or formfeed character. Otherwise it returns zero. If c is not an ASCII character or EOF, the result is undefined.
NAME
ispunct - test not alphanumeric or white space (macro)

SYNOPSIS
#include <ctype>
#define ispunct(c)

DESCRIPTION
ispunct returns a non-zero value if the argument c is a punctuation character (neither control nor alphanumeric). Otherwise it returns zero. If c is not an ASCII character or EOF, the result is undefined.

NAME
isprint - test not control or delete (macro)

SYNOPSIS
#include <ctype>
#define isprint(c)

DESCRIPTION
isprint returns a non-zero value if the argument c is a printing character, having a code between 20 hexadecimal (space) and 7E (tilde). If it is not a printing character, argument c returns zero. If c is not an ASCII character or EOF, the result is undefined.
NAME

isgraph – test not control, delete or space (macro)

SYNOPSIS

#include <ctype.h>

#define isgraph(c)

DESCRIPTION

isgraph returns a non-zero value if the argument c is printing a character other than a space. If the argument c is a space, it returns zero. If c is not an ASCII character or EOF, the result is undefined.

NAME

iscntrl – test control or delete (macro)

SYNOPSIS

#include <ctype.h>

#define iscntrl(c)

DESCRIPTION

iscntrl returns a non-zero value if the argument c is an ASCII control character. If the argument c is not an ASCII control character, it returns zero. If c is not an ASCII character or EOF, the result is undefined.
NAME

isascii – test in 7-bit ASCII set (macro)

SYNOPSIS

#include <ctype>
#define isascii(c)

DESCRIPTION

isascii returns a non-zero value if the argument is an ASCII character (code less than 128 in decimal). If the argument is not an ASCII character, it returns zero.

CONVERSIONS

These routines provide conversion operations between various representations of numeric values and mapping routines between lower case and upper case characters.
NAME

atof — ASCII to float conversion (macro)

SYNOPSIS

#include <h.cvt>
#define atof(str)

DESCRIPTION

atof returns the value represented by the character string pointed to by str as a double-precision floating point number. The string is scanned up to the first unrecognised character.

In the order given, atof recognises:

- an optional string of white space characters
- an optional sign
- a string of digits optionally containing a decimal point
- an optional e or E
- an optional sign
- an integer

atof returns HUGE if the correct value causes overflow and zero if the correct value causes underflow. In both cases errno is set to [ERANGE].

atof(str) is equivalent to strtotime(str, (char **)0).

EXAMPLE

double fpnumber = atof("2.0e-3");
NAME

atol – ASCII to long conversion (macro)

SYNOPSIS

#include <cvt>

#define atol(str)

DESCRIPTION

atol returns as a long the decimal value represented by the character string pointed to by str. The string is scanned up to the first non-digit. Leading white space characters are ignored.

atol(str) is equivalent to strtol(str, (char **0, 10).

EXAMPLE

long longnum = atol(" \n87654\n");

NAME

ecvt – floating point to string conversion (smallib)

SYNOPSIS

#include <cvt>

char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

DESCRIPTION

Note the following information regarding ecvt:

• ecvt converts value to a null terminated string of ndigit digits and returns a pointer to it.

• The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded.

• The position of the decimal point relative to the beginning of the string is stored indirectly through decpt (negative means to the left of the returned digits). The decimal point is not included in the returned string.

• If the sign of the result is negative, the word pointed to by sign is non-zero. Otherwise it is zero.

EXAMPLE

int decplace, isneg;
puts(ecvt(4.567, 20, &decplace, &isneg));
NAME
fcvt – floating point to string conversion (smallib)

SYNOPSIS
#include <h.cvt>
char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

DESCRIPTION
Note the following points regarding fcvt:

- fcvt converts value to a NULL terminated string of ndigit digits and returns a pointer to it.
- The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded for printf '%f' (FORTRAN F-format) output of the number of digits specified by ndigit.
- The position of the decimal point relative to the beginning of the string is stored indirectly through decpt (negative means to the left of the returned digits).
- The decimal point is not included in the returned string. If the sign of the result is negative, the word pointed to by sign is non-zero. Otherwise it is zero.

EXAMPLE
int decplace, isneg;
puts(fcvt(-89.3, 26, &decplace, &isneg));
NAME

strtod – string to float conversion (system)

SYNOPSIS

#include <h.cvt>

double strtod(str, ptr)
char *str, **ptr;

DESCRIPTION

strtod returns as a double-precision floating point number the value represented by the character string pointed to by str. The string is scanned up to the first unrecognised character.

In the following order strtod recognises:

• an optional string of white space characters
• an optional sign
• a string of digits optionally containing a decimal point
• an optional e or E
• an optional sign
• an integer

If the value of ptr is not NULL, a pointer to the character terminating the scan is returned in the location pointed to by ptr. If no number can be formed and zero is returned, *ptr is set to str. This is only true, however, if 0 is non-NULL ptr.

EXAMPLE

double fpnumber = strtod(inputstring);
EXAMPLE

```c
long longval = strtol("\t0xEE000000", NULL, 0);
```

NAME
toupper – convert character to upper case (system)

SYNOPSIS

```c
#include <ctype>

int toupper(c)

int c;
```

DESCRIPTION

If c is a lower case letter, toupper returns the code for the corresponding upper case letter. Otherwise, the value of c is returned unchanged.
NAME
tolower – convert character to lower case (system)

SYNOPSIS
#include <ctype>

int tolower(c)
int c;

DESCRIPTION
If c is an upper case letter, tolower returns the code for the corresponding lower case letter. Otherwise, the value of c is returned unchanged.

NAME
toupper – convert lower case letter to upper case (macro)

SYNOPSIS
#include <ctype>

#define toupper(c)

DESCRIPTION
If c is a lower case letter, toupper returns the code for the corresponding upper case letter. Otherwise, the value is undefined.
NAME

_tolower — convert upper case letter to lower case (macro)

SYNOPSIS

#include <ctype>
#define _tolower(c)

DESCRIPTION

If c is an upper case letter, _tolower returns the code for the corresponding lower case letter. Otherwise, the value is undefined.
MISCELLANEOUS

The library routine assert does not fall into any of the previous or following categories.

NAME

assert – print error and stop if assertion failed (macro)

SYNOPSIS

#include <cassert>

#define assert(ex)

DESCRIPTION

If expression is false (zero), assert prints the following line on the standard error output:

"Assertion failed: file xyz, line nnn"

When printout is finished, assert exits.

xyz is the name of the source file and nnn the source line number of the assert statement.

To remove this feature, declare #define NDEBUG before #include <cassert>, or recompile with -DNDEBUG.

assert calls _assert which is in stdlib.

EXAMPLE

assert(BUFLEN==32*sizeof(int));
This chapter describes the memory map of the C system and lists the pointers to the various parts of the map. It is written for those of you who are expert users of C and wish to use C to call machine code routines, etc.

### THE MEMORY MAP

<table>
<thead>
<tr>
<th></th>
<th>0xC000 on host, 0xF800 on Tube</th>
<th>0x8000 on host, 0xD000 on Tube</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>_hb initially</td>
<td>_hb after some calls to malloc()</td>
</tr>
<tr>
<td>C program</td>
<td>_sp while running program</td>
<td>_sp initially (= PAGE)</td>
</tr>
<tr>
<td>heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS workspace</td>
<td>0x0800</td>
<td>0x0400</td>
</tr>
<tr>
<td>C workspace</td>
<td>0x0200</td>
<td>0x0100</td>
</tr>
<tr>
<td>OS workspace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6502 stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS zeropage ws</td>
<td></td>
<td></td>
</tr>
<tr>
<td>user zeropage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lib zeropage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C zeropage</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RUN-TIME SYSTEM WORKSPACE AND VARIABLES

The values of several variables relevant to the system workspace are available. They are stored in minlib.

The values are as follows:

<table>
<thead>
<tr>
<th>Location</th>
<th>Description and library</th>
</tr>
</thead>
<tbody>
<tr>
<td>._facc</td>
<td>floating point accumulator</td>
</tr>
<tr>
<td>._ftmp</td>
<td>floating point temporary mantissa</td>
</tr>
<tr>
<td>._fwrk</td>
<td>floating point temporary accumulator</td>
</tr>
<tr>
<td>._fwsa</td>
<td>temporary floating point store</td>
</tr>
<tr>
<td>._fwsb</td>
<td>temporary floating point store</td>
</tr>
<tr>
<td>._fwsc</td>
<td>temporary floating point store</td>
</tr>
<tr>
<td>._fwsd</td>
<td>temporary floating point store</td>
</tr>
<tr>
<td>._hb</td>
<td>lowest point currently used by heap</td>
</tr>
<tr>
<td>._iacc</td>
<td>integer accumulator</td>
</tr>
<tr>
<td>._iob</td>
<td>array of file control blocks</td>
</tr>
<tr>
<td>._seterr</td>
<td>set system error from machine code</td>
</tr>
<tr>
<td>._sp</td>
<td>C stack pointer</td>
</tr>
<tr>
<td>._sysmsg</td>
<td>last error message reported to system</td>
</tr>
<tr>
<td>errno</td>
<td>last error in system or library function (declared in h_errno)</td>
</tr>
<tr>
<td>LibZP</td>
<td>16 bytes (long) of library zero page (declared in h.osdefs)</td>
</tr>
<tr>
<td>usrZP</td>
<td>base of user zero page (long) (declared in h.osdefs)</td>
</tr>
</tbody>
</table>

APPENDIX A

EDITOR COMMAND SUMMARY

Below is a summary of the cursor movements and function key commands available in the editor.

CURSOR MOVEMENT KEY ASSIGNMENTS

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>Moves the cursor up one line</td>
</tr>
<tr>
<td>[2]</td>
<td>Moves the cursor down one line</td>
</tr>
<tr>
<td>[3]</td>
<td>Moves the cursor one character to the left</td>
</tr>
<tr>
<td>[4]</td>
<td>Moves the cursor one character to the right</td>
</tr>
<tr>
<td>[SHIFT]</td>
<td>Moves the cursor to the top of the screen</td>
</tr>
<tr>
<td>[SHIFT]</td>
<td>Moves the cursor to the bottom of the screen</td>
</tr>
<tr>
<td>[SHIFT]</td>
<td>Moves the cursor to the beginning of the line</td>
</tr>
<tr>
<td>[SHIFT]</td>
<td>Moves the cursor to the end of the line</td>
</tr>
<tr>
<td>[CTRL]</td>
<td>Moves the cursor to the beginning of the text file</td>
</tr>
<tr>
<td>[CTRL]</td>
<td>Moves the cursor to the end of the text file</td>
</tr>
</tbody>
</table>

FUNCTION KEY ASSIGNMENTS

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[F0]</td>
<td>Search for a line</td>
</tr>
<tr>
<td>[SHIFT]</td>
<td>Display the hard (carriage) return symbol</td>
</tr>
<tr>
<td>[F1]</td>
<td>Enter an operating system command</td>
</tr>
<tr>
<td>[F2]</td>
<td>Choose insert/overwrite mode</td>
</tr>
<tr>
<td>[F3]</td>
<td>Write text from one file to another file</td>
</tr>
<tr>
<td>[SHIFT]</td>
<td>Read text from one file into another file</td>
</tr>
<tr>
<td>[F5]</td>
<td>Save text to another file</td>
</tr>
<tr>
<td>[SHIFT]</td>
<td>Remove the scroll margins</td>
</tr>
</tbody>
</table>
APPENDIX B
SYSTEM COMMAND SUMMARY

Listed below are the commands available in the Acornsoft C System.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE</td>
<td>close all open files</td>
</tr>
<tr>
<td>COMPILE filename</td>
<td>compile C source text file</td>
</tr>
</tbody>
</table>

The optional arguments are:
- `-d macro name` #define macro name
- `-f` compile function trace information into code
- `-l` invoke the linker automatically
- `-m number` set maximum number of errors
- `-q` do not print title and summary information

EDIT filename     enter screen editor (sideways RAM version)
#EDIT filename    enter screen editor (Second Processor version)
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HELP</td>
<td>list C commands</td>
</tr>
<tr>
<td>LINK filename {file} [options]</td>
<td>link one or more link files to each other or to the library</td>
</tr>
<tr>
<td></td>
<td>The optional arguments are:</td>
</tr>
<tr>
<td></td>
<td>• (-d) address define address for code</td>
</tr>
<tr>
<td></td>
<td>• (-l) exclude stdlib</td>
</tr>
<tr>
<td></td>
<td>• (-n) give function information</td>
</tr>
<tr>
<td></td>
<td>• (-q) do not print title and summary information</td>
</tr>
<tr>
<td></td>
<td>• (-r) run the object program automatically</td>
</tr>
<tr>
<td>MODE number</td>
<td>set the current screen mode</td>
</tr>
<tr>
<td>REPORT</td>
<td>print last error encountered</td>
</tr>
<tr>
<td>SETPATH element {, element}</td>
<td>specify search path for commands etc</td>
</tr>
<tr>
<td>SHOWPATH</td>
<td>print the current path</td>
</tr>
</tbody>
</table>

**Description**

Any other (ie unrecognised) command, such as `hello`, is interpreted as the name of a code file which C attempts to find, using the current path, and execute.

**Description**

Toggles the debug function trace mode.

**Description**

Ignore rest of line.
<table>
<thead>
<tr>
<th>Function/Macro</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>access</code></td>
<td>read the attributes of a file</td>
<td>139</td>
</tr>
<tr>
<td><code>assert</code></td>
<td>print error and stop if assertion failed</td>
<td>207</td>
</tr>
<tr>
<td><code>assert</code></td>
<td>print error when assert fails</td>
<td>207</td>
</tr>
<tr>
<td><code>atof</code></td>
<td>ASCII to float conversion</td>
<td>192</td>
</tr>
<tr>
<td><code>atoi</code></td>
<td>ASCII to int conversion</td>
<td>193</td>
</tr>
<tr>
<td><code>atol</code></td>
<td>ASCII to long conversion</td>
<td>194</td>
</tr>
<tr>
<td><code>call</code></td>
<td>call machine code, returning AXYP</td>
<td>135</td>
</tr>
<tr>
<td><code>calloc</code></td>
<td>allocate space on heap and clear to zero</td>
<td>155</td>
</tr>
<tr>
<td><code>chdir</code></td>
<td>change currently selected directory</td>
<td>140</td>
</tr>
<tr>
<td><code>chmod</code></td>
<td>set the attributes of a file</td>
<td>141</td>
</tr>
<tr>
<td><code>clearerr</code></td>
<td>clear error and end-of-stream conditions</td>
<td>116</td>
</tr>
<tr>
<td><code>close</code></td>
<td>close a file</td>
<td>126</td>
</tr>
<tr>
<td><code>creat</code></td>
<td>create a new file/rewrite existing file</td>
<td>117</td>
</tr>
<tr>
<td>Function/Macro</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>ecvt</td>
<td>floating point to string conversion</td>
<td>195</td>
</tr>
<tr>
<td>escape</td>
<td>alter/check the ESCAPE status</td>
<td>142</td>
</tr>
<tr>
<td>_exec</td>
<td>pass line to C system CLI</td>
<td>137</td>
</tr>
<tr>
<td>execl</td>
<td>run another C program, passing arguments</td>
<td>144</td>
</tr>
<tr>
<td>exit</td>
<td>quit, closing streams</td>
<td>145</td>
</tr>
<tr>
<td>_exit</td>
<td>return to C system</td>
<td>138</td>
</tr>
<tr>
<td>fclose</td>
<td>close a stream</td>
<td>82</td>
</tr>
<tr>
<td>fcvt</td>
<td>floating point to string conversion</td>
<td>196</td>
</tr>
<tr>
<td>feof</td>
<td>test end-of-stream condition</td>
<td>118</td>
</tr>
<tr>
<td>ferror</td>
<td>test stream error condition</td>
<td>119</td>
</tr>
<tr>
<td>fflush</td>
<td>flush a stream's buffers</td>
<td>83</td>
</tr>
<tr>
<td>fgetc</td>
<td>get a character from a stream</td>
<td>84</td>
</tr>
<tr>
<td>fgets</td>
<td>get a string from a stream</td>
<td>85</td>
</tr>
<tr>
<td>fileno</td>
<td>file descriptor for a stream</td>
<td>120</td>
</tr>
<tr>
<td>fopen</td>
<td>opens a stream</td>
<td>86</td>
</tr>
<tr>
<td>fprintf</td>
<td>formatted printing to a stream</td>
<td>88</td>
</tr>
<tr>
<td>fputc</td>
<td>put a character to a stream</td>
<td>92</td>
</tr>
<tr>
<td>fputs</td>
<td>put a string to a stream</td>
<td>93</td>
</tr>
<tr>
<td>fread</td>
<td>read n items from a stream</td>
<td>94</td>
</tr>
<tr>
<td>free</td>
<td>free previously allocated block of memory</td>
<td>156</td>
</tr>
<tr>
<td>freopen</td>
<td>close and reopen a stream</td>
<td>96</td>
</tr>
<tr>
<td>fscanf</td>
<td>formatted input from a stream</td>
<td>97</td>
</tr>
<tr>
<td>fseek</td>
<td>set the pointer on a stream</td>
<td>102</td>
</tr>
<tr>
<td>ftell</td>
<td>read the pointer of a stream</td>
<td>103</td>
</tr>
<tr>
<td>fwrite</td>
<td>write n items to a stream</td>
<td>104</td>
</tr>
<tr>
<td>gcvt</td>
<td>floating point to string conversion</td>
<td>197</td>
</tr>
<tr>
<td>getc</td>
<td>get a character from a stream</td>
<td>121</td>
</tr>
<tr>
<td>getchar</td>
<td>get a character from the standard input stream</td>
<td>122</td>
</tr>
<tr>
<td>gets</td>
<td>get a string from the standard input stream</td>
<td>105</td>
</tr>
<tr>
<td>getw</td>
<td>get a word from a stream</td>
<td>106</td>
</tr>
<tr>
<td>isalnum</td>
<td>test alphanumeric</td>
<td>184</td>
</tr>
<tr>
<td>isalpha</td>
<td>test alphabetic</td>
<td>179</td>
</tr>
<tr>
<td>isascii</td>
<td>test in 7-bit ASCII set</td>
<td>190</td>
</tr>
<tr>
<td>iscntrl</td>
<td>test control or delete</td>
<td>189</td>
</tr>
<tr>
<td>isdigit</td>
<td>test digit 0..9</td>
<td>182</td>
</tr>
<tr>
<td>Function/Macro</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>isgraph</td>
<td>test not control, delete or space character</td>
<td>188</td>
</tr>
<tr>
<td>islower</td>
<td>test lower case</td>
<td>181</td>
</tr>
<tr>
<td>isprint</td>
<td>test not control or delete</td>
<td>187</td>
</tr>
<tr>
<td>ispunct</td>
<td>test not alphanumeric or white space</td>
<td>186</td>
</tr>
<tr>
<td>isspace</td>
<td>test white space: space character, TAB, etc.</td>
<td>185</td>
</tr>
<tr>
<td>isupper</td>
<td>test upper case</td>
<td>180</td>
</tr>
<tr>
<td>isxdigit</td>
<td>test hexadecimal digit (0..9,a..f,A..F)</td>
<td>183</td>
</tr>
<tr>
<td>longjmp</td>
<td>jump to previously saved position</td>
<td>146</td>
</tr>
<tr>
<td>lseek</td>
<td>set pointer for a file</td>
<td>127</td>
</tr>
<tr>
<td>malloc</td>
<td>allocate space on heap</td>
<td>154</td>
</tr>
<tr>
<td>memccpy</td>
<td>move memory up to character</td>
<td>159</td>
</tr>
<tr>
<td>memchr</td>
<td>find character in memory</td>
<td>160</td>
</tr>
<tr>
<td>memcmp</td>
<td>compare memory</td>
<td>161</td>
</tr>
<tr>
<td>memcpy</td>
<td>move block of memory</td>
<td>162</td>
</tr>
<tr>
<td>memset</td>
<td>set block of memory to character</td>
<td>163</td>
</tr>
<tr>
<td>open</td>
<td>open a file</td>
<td>128</td>
</tr>
<tr>
<td>oscall</td>
<td>call machine code, with BRKs indicated</td>
<td>136</td>
</tr>
<tr>
<td>_osret</td>
<td>structure for oscall return block</td>
<td>136</td>
</tr>
<tr>
<td>perror</td>
<td>print message and last system error</td>
<td>107</td>
</tr>
<tr>
<td>printf</td>
<td>formatted output to the standard output stream</td>
<td>108</td>
</tr>
<tr>
<td>putc</td>
<td>put a character to a stream</td>
<td>123</td>
</tr>
<tr>
<td>putchar</td>
<td>put a character on the standard output stream</td>
<td>124</td>
</tr>
<tr>
<td>puts</td>
<td>put a string to the standard output stream</td>
<td>109</td>
</tr>
<tr>
<td>putw</td>
<td>put a word to a stream</td>
<td>110</td>
</tr>
<tr>
<td>read</td>
<td>read bytes from a file</td>
<td>130</td>
</tr>
<tr>
<td>rewind</td>
<td>reset the file pointer on a stream</td>
<td>111</td>
</tr>
<tr>
<td>scanf</td>
<td>formatted input from the standard input stream</td>
<td>112</td>
</tr>
<tr>
<td>seterror</td>
<td>set the system error message and number</td>
<td>147</td>
</tr>
<tr>
<td>setjmp</td>
<td>save a position for a later longjmp</td>
<td>148</td>
</tr>
<tr>
<td>Function/Macro</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>setpath</td>
<td>set the system file search path</td>
<td>149</td>
</tr>
<tr>
<td>showpath</td>
<td>return a pointer to the system file search path</td>
<td>150</td>
</tr>
<tr>
<td>sprintf</td>
<td>formatted output to a string</td>
<td>113</td>
</tr>
<tr>
<td>sscanf</td>
<td>formatted input from a string</td>
<td>114</td>
</tr>
<tr>
<td>strcat</td>
<td>concatenate strings</td>
<td>165</td>
</tr>
<tr>
<td>strchr</td>
<td>search for character in string</td>
<td>166</td>
</tr>
<tr>
<td>strcmp</td>
<td>compare two strings</td>
<td>167</td>
</tr>
<tr>
<td>strcpy</td>
<td>copy a string</td>
<td>168</td>
</tr>
<tr>
<td>strcspn</td>
<td>length of string not made of given characters</td>
<td>169</td>
</tr>
<tr>
<td>strlen</td>
<td>calculate length of string</td>
<td>170</td>
</tr>
<tr>
<td>strncpy</td>
<td>add (part of) string to string</td>
<td>171</td>
</tr>
<tr>
<td>strcpy</td>
<td>copy (parts of) strings</td>
<td>172</td>
</tr>
<tr>
<td>strpbrk</td>
<td>search for set of characters in string</td>
<td>173</td>
</tr>
<tr>
<td>strrchr</td>
<td>search for character in string</td>
<td>174</td>
</tr>
<tr>
<td>strspn</td>
<td>length of string made of given chars</td>
<td>175</td>
</tr>
<tr>
<td>strtok</td>
<td>string to float conversion</td>
<td>198</td>
</tr>
<tr>
<td>strtock</td>
<td>break up string into 'tokens'</td>
<td>176</td>
</tr>
<tr>
<td>strtol</td>
<td>string to long conversion</td>
<td>199</td>
</tr>
<tr>
<td>system</td>
<td>pass a string to the machine</td>
<td>151</td>
</tr>
<tr>
<td>toascii</td>
<td>ensure character is valid ASCII</td>
<td>205</td>
</tr>
<tr>
<td>tolower</td>
<td>convert character to lower case</td>
<td>202</td>
</tr>
<tr>
<td>_tolower</td>
<td>convert upper case letter to lower case</td>
<td>204</td>
</tr>
<tr>
<td>toupper</td>
<td>convert character to upper case</td>
<td>201</td>
</tr>
<tr>
<td>_toupper</td>
<td>convert lower case letter to upper case</td>
<td>203</td>
</tr>
<tr>
<td>trace</td>
<td>turn function tracing on/off</td>
<td>152</td>
</tr>
<tr>
<td>ungetc</td>
<td>push character back onto a stream</td>
<td>115</td>
</tr>
<tr>
<td>unlink</td>
<td>delete a file</td>
<td>132</td>
</tr>
<tr>
<td>write</td>
<td>write bytes to a file</td>
<td>133</td>
</tr>
</tbody>
</table>
# APPENDIX D

## ERROR MESSAGES

<table>
<thead>
<tr>
<th>Message</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual parameter required</td>
<td>72</td>
</tr>
<tr>
<td>No parameter appeared between a comma (,) and a closing bracket (}) in a procedure call.</td>
<td></td>
</tr>
<tr>
<td>Argument for * must be a pointer</td>
<td>74</td>
</tr>
<tr>
<td>Square brackets ([]) are applied to an item other than an array or pointer.</td>
<td></td>
</tr>
<tr>
<td>Attempted access to undefined structure/union</td>
<td>43</td>
</tr>
<tr>
<td>An attempt was made to access a structure/union via a previously declared pointer.</td>
<td></td>
</tr>
<tr>
<td>Bad arg for unary &amp;</td>
<td>54</td>
</tr>
<tr>
<td>An illegal address operator application was made. The address operator only applies to a variable or memory location.</td>
<td></td>
</tr>
<tr>
<td>Bad argument for cast</td>
<td>61</td>
</tr>
<tr>
<td>The requested type coercion is impossible.</td>
<td></td>
</tr>
<tr>
<td>Bad argument for unary –</td>
<td>57</td>
</tr>
<tr>
<td>A unary minus (–) was applied to a non-numeric argument.</td>
<td></td>
</tr>
<tr>
<td>Bad argument for unary *</td>
<td>60</td>
</tr>
<tr>
<td>An illegal indirection operator application was made. The indirection operator is only applied to a pointer or array address.</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>Number</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Bad argument for ++</td>
<td>56</td>
</tr>
<tr>
<td>An illegal pre- and post-increment was made. A pre- and</td>
<td></td>
</tr>
<tr>
<td>post-increment is only applied to variables or memory</td>
<td></td>
</tr>
<tr>
<td>locations.</td>
<td></td>
</tr>
<tr>
<td>Bad argument for -</td>
<td>58</td>
</tr>
<tr>
<td>The complement operator was applied to a non-integral</td>
<td></td>
</tr>
<tr>
<td>argument.</td>
<td></td>
</tr>
<tr>
<td>Bad argument for !</td>
<td>59</td>
</tr>
<tr>
<td>The logical negation operator was applied to a non-numeric</td>
<td></td>
</tr>
<tr>
<td>node.</td>
<td></td>
</tr>
<tr>
<td>Bad #if or #assert expression (must reduce to constant)</td>
<td>15</td>
</tr>
<tr>
<td>An error occurred in evaluating the expression after an #if or</td>
<td></td>
</tr>
<tr>
<td>#assert directive. The expression must reduce to a constant</td>
<td></td>
</tr>
<tr>
<td>at compile time.</td>
<td></td>
</tr>
<tr>
<td>Badly defined declarator</td>
<td>36</td>
</tr>
<tr>
<td>A syntax error was detected in a declarator.</td>
<td></td>
</tr>
<tr>
<td>Bitfields not supported</td>
<td>87</td>
</tr>
<tr>
<td>An illegal attempt was made to implement bitfields.</td>
<td></td>
</tr>
<tr>
<td>Cannot define a function as external</td>
<td>103</td>
</tr>
<tr>
<td>An illegal attempt was made to define a function with an</td>
<td></td>
</tr>
<tr>
<td>extern attribute.</td>
<td></td>
</tr>
<tr>
<td>Cannot declare a function here</td>
<td>88</td>
</tr>
<tr>
<td>A function declaration was encountered inside a structure/union.</td>
<td></td>
</tr>
<tr>
<td>Cannot define a function here</td>
<td>31</td>
</tr>
<tr>
<td>The compiler found a function definition where only a function</td>
<td></td>
</tr>
<tr>
<td>declaration is allowed.</td>
<td></td>
</tr>
<tr>
<td>Cannot have an array of functions</td>
<td>38</td>
</tr>
<tr>
<td>The compiler detected an illegal declarator comprising an array of</td>
<td></td>
</tr>
<tr>
<td>functions.</td>
<td></td>
</tr>
<tr>
<td>Cannot have an identifier in an abstract declarator</td>
<td>34</td>
</tr>
<tr>
<td>The compiler found an identifier while parsing an abstract declarator</td>
<td></td>
</tr>
<tr>
<td>(for example, the sort of declarator that follows a sizeof(type)</td>
<td></td>
</tr>
<tr>
<td>expression). An identifier is not allowed in an abstract declarator.</td>
<td></td>
</tr>
<tr>
<td>Cannot initialise unions</td>
<td>23</td>
</tr>
<tr>
<td>An illegal attempt was made to initialise a union.</td>
<td></td>
</tr>
<tr>
<td>Cannot open file file-name : error-message</td>
<td>2</td>
</tr>
<tr>
<td>The compiler is unable to open the designated file. The error message</td>
<td></td>
</tr>
<tr>
<td>is the text returned by the filing system.</td>
<td></td>
</tr>
<tr>
<td>Character character not understood by lexical analyser</td>
<td>94</td>
</tr>
<tr>
<td>The given character is not recognised by the pre-processor pass.</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>Number</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Code generation stack overflow (function too complex)</td>
<td>35</td>
</tr>
<tr>
<td>The compiler ran out of room when generating code for a function. Try splitting the function up into smaller units.</td>
<td></td>
</tr>
<tr>
<td>Colon expected after case</td>
<td>106</td>
</tr>
<tr>
<td>A case clause is not followed by a colon.</td>
<td></td>
</tr>
<tr>
<td>Colon expected after default</td>
<td>107</td>
</tr>
<tr>
<td>A default clause is not followed by a colon.</td>
<td></td>
</tr>
<tr>
<td>Conflicting type declarations</td>
<td>29</td>
</tr>
<tr>
<td>The compiler encountered a declaration and a definition for the same name which are of different types. This means that a variable/function was declared earlier in the source with a different type.</td>
<td></td>
</tr>
<tr>
<td>Declarator buffer overflow</td>
<td>30</td>
</tr>
<tr>
<td>The compiler ran out of space when parsing a declarator. This should only happen with very large structures/unions declarators. Try splitting up the declaration into several smaller ones.</td>
<td></td>
</tr>
<tr>
<td>Doubly defined label</td>
<td>95</td>
</tr>
<tr>
<td>The compiler encountered a label which was previously declared in the current function.</td>
<td></td>
</tr>
<tr>
<td>End of comment before start of comment</td>
<td>5</td>
</tr>
<tr>
<td>An end of comment (<em>/ ) was encountered before a start of comment (/</em>).</td>
<td></td>
</tr>
<tr>
<td>End of file before end of compound statement</td>
<td>110</td>
</tr>
<tr>
<td>The compiler ran out of file space in the middle of a compound statement.</td>
<td></td>
</tr>
<tr>
<td>End of file before end of structure/union definition</td>
<td>86</td>
</tr>
<tr>
<td>The end of file is found before the end of a structure/union definition.</td>
<td></td>
</tr>
<tr>
<td>End of file encountered before #endif</td>
<td>4</td>
</tr>
<tr>
<td>During the pre-processor pass of the compiler, an end of file is encountered before #endif.</td>
<td></td>
</tr>
<tr>
<td>End of file found before end of comment</td>
<td>6</td>
</tr>
<tr>
<td>An end of comment (*/ ) is missing. It is expected to terminate a comment.</td>
<td></td>
</tr>
<tr>
<td>End of file found unexpectedly</td>
<td>108</td>
</tr>
<tr>
<td>The pre-processor unexpectedly ran out of file to parse.</td>
<td></td>
</tr>
<tr>
<td>Error in x?y:z</td>
<td>51</td>
</tr>
<tr>
<td>The wrong number of arguments was provided for the ternary conditional operator.</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>Number</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Error error-number in link output file: error-message</td>
<td>105</td>
</tr>
<tr>
<td>While the linkage output file was being generated, an error occurred.</td>
<td></td>
</tr>
<tr>
<td>Error error-number in temporary data file: error-message</td>
<td>109</td>
</tr>
<tr>
<td>While the linkage output file was being generated, an error occurred.</td>
<td></td>
</tr>
<tr>
<td>Expected a formal parameter in function declarator</td>
<td>42</td>
</tr>
<tr>
<td>While parsing a formal parameter list the compiler found something (ie a formal parameter) which was not an identifier.</td>
<td></td>
</tr>
<tr>
<td>Expression cannot be converted to boolean</td>
<td>76</td>
</tr>
<tr>
<td>An inapt expression was used as an argument for if or one of &amp;&amp; or 11.</td>
<td></td>
</tr>
<tr>
<td>Expression does not reduce to constant</td>
<td>22</td>
</tr>
<tr>
<td>The compiler expected to reduce an expression to a constant value and could not do so. For example, a constant expression is expected in array bounds in array declarations.</td>
<td></td>
</tr>
<tr>
<td>Expression required</td>
<td>46</td>
</tr>
<tr>
<td>The compiler expected an expression but could not identify one.</td>
<td></td>
</tr>
<tr>
<td>Expression workspace overflow</td>
<td>69</td>
</tr>
<tr>
<td>The expression was too complex. Try splitting the file into sections with each section containing fewer variables.</td>
<td></td>
</tr>
<tr>
<td>Filename already given on command line</td>
<td>37</td>
</tr>
<tr>
<td>More than one file name was entered on the command line; the compiler only accepts one. Flags must be prefixed by a dash (-).</td>
<td></td>
</tr>
<tr>
<td>Filename too long</td>
<td>55</td>
</tr>
<tr>
<td>The compiler was given a filename that is too long.</td>
<td></td>
</tr>
<tr>
<td>Global table full</td>
<td>115</td>
</tr>
<tr>
<td>The compiler ran out of room to internally represent the destination of a goto. Try splitting up the file so that there are less global variable/function definitions.</td>
<td></td>
</tr>
<tr>
<td>Goto label label-name not resolved at end of function</td>
<td>27</td>
</tr>
<tr>
<td>A goto was found without a corresponding label declaration.</td>
<td></td>
</tr>
<tr>
<td>Identifier expected after a goto</td>
<td>96</td>
</tr>
<tr>
<td>The compiler encountered a goto without an identifier (label).</td>
<td></td>
</tr>
<tr>
<td>Illegal control character inside string or character constant</td>
<td>104</td>
</tr>
<tr>
<td>The compiler found a carriage return inside a character/constant because there is a missing closing quote mark ('').</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>Number</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td><strong>Illegal formal parameter in macro definition</strong></td>
<td>67</td>
</tr>
<tr>
<td>A macro formal parameter has an illegal identifier. It must be a simple identifier.</td>
<td></td>
</tr>
<tr>
<td><strong>Illegal return type for function</strong></td>
<td>40</td>
</tr>
<tr>
<td>A function has returned an illegal type. A function cannot return an array, another function, or a user defined type.</td>
<td></td>
</tr>
<tr>
<td><strong>Illegal pointer operation</strong></td>
<td>48</td>
</tr>
<tr>
<td>An illegal operation was performed involving one or more pointers. For example, <code>&lt;pointer&gt; &lt;pointer&gt;</code>.</td>
<td></td>
</tr>
<tr>
<td><strong>Illegal type coercion</strong></td>
<td>78</td>
</tr>
<tr>
<td>An expression requires an illegal implicit type conversion.</td>
<td></td>
</tr>
<tr>
<td><strong>Illegal type for operand</strong></td>
<td>77</td>
</tr>
<tr>
<td>The operands for the operand are an illegal type.</td>
<td></td>
</tr>
<tr>
<td><strong>Illegal type of actual parameter</strong></td>
<td>44</td>
</tr>
<tr>
<td>An illegal parameter was detected (ie a structure/union).</td>
<td></td>
</tr>
<tr>
<td><strong>Illegal #directive</strong></td>
<td>13</td>
</tr>
<tr>
<td>An unrecognised pre-processor directive was encountered.</td>
<td></td>
</tr>
<tr>
<td><strong>Illegal #include (use &quot;filename&quot; or &lt;filename&gt;)</strong></td>
<td>9</td>
</tr>
<tr>
<td>The filename specified after a <code>#include</code> is not enclosed in quotes or brackets.</td>
<td></td>
</tr>
<tr>
<td><strong>Initialisation string is too long</strong></td>
<td>24</td>
</tr>
<tr>
<td>A string supplied for initialising a character array is bigger than the array.</td>
<td></td>
</tr>
<tr>
<td><strong>Lexical analyser out of memory</strong></td>
<td>14</td>
</tr>
<tr>
<td>The program is too big for the pre-processor. Try splitting up the program.</td>
<td></td>
</tr>
<tr>
<td><strong>l-value required</strong></td>
<td>75</td>
</tr>
<tr>
<td>The left hand side of an assignment operator is lacking a variable or the contents of a memory address.</td>
<td></td>
</tr>
<tr>
<td><strong>Macro parameters not closed with a ')'</strong></td>
<td>17</td>
</tr>
<tr>
<td>The parameters of a macro are not enclosed in parentheses.</td>
<td></td>
</tr>
<tr>
<td><strong>Macro with wrong number of parameters</strong></td>
<td>8</td>
</tr>
<tr>
<td>A macro parameter containing the incorrect number of parameters is invoked.</td>
<td></td>
</tr>
<tr>
<td><strong>Mismatched pointers</strong></td>
<td>50</td>
</tr>
<tr>
<td>An attempt was made to take the difference between pointers which point at different size objects.</td>
<td></td>
</tr>
<tr>
<td><strong>Missing ')' in declarator</strong></td>
<td>33</td>
</tr>
<tr>
<td>A close bracket in a parenthesised declarator (`) is missing, detected by the compiler.</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>Number</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Missing '[' in array bound</td>
<td>41</td>
</tr>
<tr>
<td>A close square bracket (]) needed to terminate an array bound is missing. This was detected by the compiler.</td>
<td></td>
</tr>
<tr>
<td>Multiply defined global identifier variable</td>
<td>25</td>
</tr>
<tr>
<td>A name has been given that was previously defined in the source.</td>
<td></td>
</tr>
<tr>
<td>Multiply defined local identifier variable</td>
<td>28</td>
</tr>
<tr>
<td>A name has been given that was previously defined in the compound statement.</td>
<td></td>
</tr>
<tr>
<td>Need '(' to precede parameters in a macro</td>
<td>16</td>
</tr>
<tr>
<td>The parameters of a macro are not enclosed in parentheses.</td>
<td></td>
</tr>
<tr>
<td>No ')' in function declarator or declaration</td>
<td>39</td>
</tr>
<tr>
<td>The compiler parsed a function declarator/declaration whose parameter list was terminated without an expected close bracket (').</td>
<td></td>
</tr>
<tr>
<td>Option option not understood in command line</td>
<td>84</td>
</tr>
<tr>
<td>The compiler does not recognise the displayed option.</td>
<td></td>
</tr>
<tr>
<td>Out of memory</td>
<td>3</td>
</tr>
<tr>
<td>The compiler ran out of memory. This usually means that your source file is too big.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter list can only be applied to a function</td>
<td>113</td>
</tr>
<tr>
<td>The compiler encountered a parameter list that is associated with an object which does not evaluate to a function.</td>
<td></td>
</tr>
<tr>
<td>Parameter list too complex</td>
<td>66</td>
</tr>
<tr>
<td>A parameter list was used which exceeded the compiler’s capabilities.</td>
<td></td>
</tr>
<tr>
<td>Prefix for '[' must be array or pointer</td>
<td>71</td>
</tr>
<tr>
<td>An attempt was made to subscript an item other than an array or pointer.</td>
<td></td>
</tr>
<tr>
<td>RHS of operator expected</td>
<td>47</td>
</tr>
<tr>
<td>A binary operator without a second argument was encountered.</td>
<td></td>
</tr>
<tr>
<td>Semicolon expected</td>
<td>89</td>
</tr>
<tr>
<td>The compiler expected it was at the end of a statement and expected a semi-colon to terminate it. This is a common error.</td>
<td></td>
</tr>
<tr>
<td>Size of structure/union not known</td>
<td>53</td>
</tr>
<tr>
<td>An attempt was made to take the size of a structure/union which was defined only by a previously declared pointer.</td>
<td></td>
</tr>
<tr>
<td>Sizeof argument required</td>
<td>52</td>
</tr>
<tr>
<td>The sizeof unary operator without an argument is encountered.</td>
<td></td>
</tr>
</tbody>
</table>
Message | Number
--- | ---
Statement or declarator in invalid context | 100
The compiler does not recognise the statement it is trying to parse. This is a common error.
Structure/union specifier expected | 83
The compiler has encountered a struct without a recognised structure/union specifier/declaration following it.
Structure/union tag tag-name already declared | 82
The tag-name given to a structure/union tag was used previously.
Subscript required after ']' | 68
An array access without a subscript was encountered.
Symbol table overflow | 93
The compiler has run out of symbol table space. Try splitting up the program.
Typedef tag name already exists | 45
An attempt was made to declare a typedef with a previously used name.
Type specifier expected | 81
The compiler expected a type specifier such as int, char *....

Message | Number
--- | ---
Unexpected value in temporary data file | 79
The compiler detected an error in its temporary data file, therefore the media has failed.
Unexpected ')' in expression | 116
The compiler encountered a bracket (') and is unable to parse the following expression.
Unknown LHS for . or -> | 64
The structure/union access operators are incorrectly applied. They can only be applied to structures/unions and pointers to them, respectively.
Unknown offset into structure/union | 65
The tag field given in a structure access is not known in the accessed structure.
Warning: compiler has performed float conversion | 70
The compiler has truncated a float/double into an integral type in a way that may not be totally portable.
Warning: compiler has performed pointer conversion | 80
The compiler has converted a pointer expression into a non-pointer type that may not be portable.
Warning: Condition is constant in x?y:z | 112
A condition evaluated to a constant, therefore it was unnecessary.
<table>
<thead>
<tr>
<th>Message</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warning: LHS of comma ignored</td>
<td>49</td>
</tr>
<tr>
<td>The left hand side of a comma operator was ignored because it was a constant or a simple variable access.</td>
<td></td>
</tr>
<tr>
<td>Warning: Size of object is zero/undefined</td>
<td>32</td>
</tr>
<tr>
<td>An attempt was made to reference the size of an object which was zero. For example, taking the <code>sizeof</code> an array which was declared without specifying any bounds.</td>
<td></td>
</tr>
<tr>
<td>While expected to close do loop</td>
<td>97</td>
</tr>
<tr>
<td>The compiler has not found the expected <code>while</code> construct to close a do loop.</td>
<td></td>
</tr>
<tr>
<td>&quot; or ' expected</td>
<td>7</td>
</tr>
<tr>
<td>The pre-processor was expecting the end of a string/character constant which was absent.</td>
<td></td>
</tr>
<tr>
<td>#assert failed</td>
<td>62</td>
</tr>
<tr>
<td>The pre-processor assertion test failed.</td>
<td></td>
</tr>
<tr>
<td>#else without a #if</td>
<td>10</td>
</tr>
<tr>
<td>A <code>#else</code> was encountered without a corresponding <code>#if</code> directive.</td>
<td></td>
</tr>
<tr>
<td>#endif without a #if</td>
<td>11</td>
</tr>
<tr>
<td>A <code>#endif</code> was encountered without a corresponding <code>#if</code> directive.</td>
<td></td>
</tr>
<tr>
<td>#line has to be followed by an integer expression</td>
<td>12</td>
</tr>
<tr>
<td>A <code>#line</code> directive was encountered without a corresponding <code>#if</code> directive.</td>
<td></td>
</tr>
<tr>
<td>'}' expected at the start of function body</td>
<td>26</td>
</tr>
<tr>
<td>The compiler expected an open brace <code>{}</code> to start the body of a function. The brace is absent.</td>
<td></td>
</tr>
<tr>
<td>'}' expected to start array initialisation</td>
<td>18</td>
</tr>
<tr>
<td>An initialiser for an array does not begin with an open brace <code>{}</code>.</td>
<td></td>
</tr>
<tr>
<td>'}' expected to start structure initialisation</td>
<td>19</td>
</tr>
<tr>
<td>An initialiser for a structure does not begin with an open brace <code>{}</code>.</td>
<td></td>
</tr>
<tr>
<td>'}' expected after single value initialiser</td>
<td>21</td>
</tr>
<tr>
<td>A single value initialiser starting with an open brace <code>{}</code> does not have a corresponding close brace <code>} </code>.</td>
<td></td>
</tr>
<tr>
<td>'}' expected to end initialiser list</td>
<td>20</td>
</tr>
<tr>
<td>An initialiser list was not followed by a close brace <code>} </code>.</td>
<td></td>
</tr>
<tr>
<td>'}' expected after a for statement</td>
<td>98</td>
</tr>
<tr>
<td>A for statement is not followed by an open bracket <code>}</code> to delimit the loop expressions.</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>Number</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>'c' expected before condition</td>
<td>101</td>
</tr>
<tr>
<td>A condition is not enclosed in brackets, for example, after an if.</td>
<td></td>
</tr>
<tr>
<td>'}' expected</td>
<td>114</td>
</tr>
<tr>
<td>A close bracket (}) is missing from an expression.</td>
<td></td>
</tr>
<tr>
<td>'}' expected after a for statement</td>
<td>99</td>
</tr>
<tr>
<td>A for statement is not followed by a close bracket (}) to delimit the loop expressions.</td>
<td></td>
</tr>
<tr>
<td>'}' or ',' required after actual parameter list</td>
<td>73</td>
</tr>
<tr>
<td>An open bracket (({ or a comma (,), needed after an actual parameter, is missing.</td>
<td></td>
</tr>
<tr>
<td>'}' missing from condition</td>
<td>102</td>
</tr>
<tr>
<td>A condition is not enclosed in brackets, for example, after an if.</td>
<td></td>
</tr>
<tr>
<td>'\0' not allowed in strings</td>
<td>1</td>
</tr>
<tr>
<td>A \0 was found in a string. Since \0 is the string terminator in C, you cannot have the character \0 in a string.</td>
<td></td>
</tr>
<tr>
<td>Local-variable is illegally initialised</td>
<td>111</td>
</tr>
<tr>
<td>The compiler has encountered the initialisation of an illegal local array/structure.</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX E

FURTHER READING ON C

If you want to find out more about C, you may find the following books of interest:


C Programming Guide by J Purdum, published by Que Corporation, Indianapolis, Indiana, USA.

The C Programming Language by B W Kernighan and D M Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, USA.

INDEX

access modes (amode) 139
Acorn Cambridge Workstation 1
Acornsoft C 1
   editor 13
   restrictions 2
   system calls 79,134
anachronisms 3
ASCII codes 19,21

base
   positive 199
   zero 199
BBC Master
   128 13,66,80
   Turbo 13
bitfields 3
BRK 136

classification 79,178
character
   display 19,21
   type 86
clear markers 19
CLOSE 7,213
code generation 56
command line interpreter 7
command macros 19,43-45
COMPILE 7,213
   -d macroname 7,213
   -f 7,10,59,213
   -l 8,61,66,213
   -m number 7,213
   -q 8,9,62,67,213
   -r<restofline> 8,9
compiler 11,13,47,63
options 59
   -d 59,65-66
   -dDEBUG 59
#assert constant expression 54
#elif constant expression 53
#else 53
#endif 53
#if constant expression 52
#elsedef identifier 52
#elsendef identifier 53
conditional compiler directives 52
control characters 21
data type
enum 3
void 2,4,60
default path values 9
default status line 15
delete character 21
deleting 17,19,20,26-27
digit string 89
directive 4
#define 4,59
display mode 19
double argument (see also float) 90
EDIT 13-14,213
*EDIT 213
Editor 13,15
command summary 211
data type
enum 3
void 2,4,60
default path values 9
default status line 15
delete character 21
deleting 17,19,20,26-27
digit string 89
enable (ENA_ESC) 142
return zero (TST_ESC) 142
extensions 4

file inclusion 51
fildes 130
#include 51
status flags 128

filing systems
   capable of random access 133
   incapable of random access 133

finding and deleting text
   a string globally 35-36
   a string only 34-35

finding and replacing text
   a line 31
   a string globally 35
   a string only 31-32
   a string selectively 32-33
   text 18,19,30-31
   with patterns 36-43

float 4,5,90
floating point
   accumulator (float) 210
   array of file control blocks (-iob) 210
   base of user zero page (usrZP) 210
   C stack pointer (-sp) 210
   integer accumulator (-iacc) 210
   last error in system or library function
      (errno) 117,210
   last error message (sysmsg) 210
   lowest point used by head (-hb) 210
   set system error for machine code
      (-seterr) 210
   16 bites (long[]) of library zero page
      (libZP) 210
   temporary accumulator (-fwrk) 210
   temporary mantissa (-ftmp) 210

   temporary stores (-fwsa, -fwsb, -fWSC,
   -fwsd) 210
   full stop (.) 10,37
   function declarations
      hstdio 73
      hstring 73
   function keys 16

   global replacements 45
      !:E 46
      <from>/<to> 46
      KEYB 45
      IM 45,46

   hard carriage return 25-27
   hard carriage return symbol 18
   ‘hard’ characters 19,21
   header files/flags 73
      herrno 73,74
      EACCES 74
      EBADF 75
      EBADF 75
      EBCOM 75
      EBNAME 75
      EEEXIST 75
      EEEXIST 75
      EFBIG 74
      EINTR 74
      EINVAL 74
      ENFILE 74
      EPERM 74
      ENFILE 74
      ENODEV 75
      ENODEV 75
      ENFILE 74
      ENFILE 74
      ENODEV 75
      ENOTDIR 75
      ENOTDIR 75
      ENOTDIR 75
      ENOTDIR 75
      EPATH 74
      ENOTS 75
      ENOTS 75
EWILD 75
h.fcntl 73,75
F_OK 76,139
L_OK 76,139
O_APPEND 75,117,128
O_CREAT 75,117,128
O_EXCL 76,117,128
O_NCSO 76,117,128
O_NPATH 76,117,128
O_RDONLY 75,117,128
O_RDWR 76,117,129
O_TRUNC 75,117,128
O_WRONLY 75,117,128
PL_OK 76,139
PR_OK 76,139
PW_OK 76,139
PX_OK 76
R_OK 76,139
SEEK_CUR 76
SEEK_END 76
SEEK_SET 76
W_OK 76,139
X_OK 76
h.math 73,77
HUGE 77
MAXFLOAT 77
M_E 77
M_LN2 77
M_LN10 77
M_LOG2E 77
M_LOG10E 77
M_PI 77
M_PI_2 77
M_PI_4 77
M_SQRT2 77
M_1_PI 77
M_1_SQRT2 77
M_2_PI 77
M_1_SQRTPI 77
h.osdefs 73
OSARGS 73
OSASCI 74
OSBGET 73
OSBPUT 73
OSBYTE 74
OSCLI 74
OSFILE 74,126
OSFIND 73
OSGBP 73
OSNEWL 74
OSRDCH 74
OSWORD 74
OSWRCH 74
heap allocation 79,153
heap storage 153
HELP 8,214
input device
*FX2 80
insert 15,18,19,24-25
installation 7

keyboard 16-19
cursor movement keys 16
editing control keys 17-18
[ COPY ] 17,44
[ CTRL ] 17,211
[ DELETE ] 17,44
[ ESCAPE ] 17
[ RETURN ] 17
[ SHIFT ] 17,211
[ TAB ] 17,18,44
[ ] 45,211
[ ] 45,211
[ ] 45,211
[ ] 45,211
Second Processor 13-14, 18, 46, 66, 212
sideways RAM 13-14, 18, 46, 212

libraries, specified 64
  minlib 64
  smallib 64
  stdlib 8, 63, 64-65
  system 64, 151, 217, 223
library functions (see library routines)
library routines 79-81
  character classification 79, 178
    isalnum 184, 219
    isalpha 179, 219
    isascii 190, 219
    iscntrl 189, 219
    isdigit 182, 219
    isgraph 188, 220
    islower 181, 220
    isprint 187, 220
    ispunct 186, 220
    isspace 185, 220
    isupper 180, 220
    isxdigit 183, 220
conversion operations 79, 191
  atof 192, 217
  atoi 193, 217
  atol 194, 217

ecvt 195, 218
fenv 196, 218
genv 197, 219
strtol 198, 222
strtol 199, 223
toascii 205, 223
tolower 202, 223
-tolower 204, 223
toupper 201, 223
-toupper 203, 223
heap allocation 79, 153
heap storage 153
calloc 155, 217
free 156, 219
mallinfo 157, 220
-mallinfo 157, 220
malloc 154, 220
low level i/o routines 79, 125
  close 126, 217
  lseek 127, 220
  open 117, 128, 130, 221
  read 130, 221
  unlink 132, 223
  write 133, 223
memory operations 79, 158
  memcpy 159, 220
  memchr 160, 220
  memcmp 161, 220
  memcmp 162, 220
  memset 163, 221
miscellaneous 206
  assert 207, 217
  -assert 207, 217
standard i/o routines 79-81, 125
  clearerr 116, 119, 217
  creat 117, 130, 217
  fclose 79, 82, 218
  feof 118, 121, 122, 218
INDEX

strncpy 172,222
strpbrk 173,222
strstr 174,222
strspn 175,222
strtok 176,222
system calls 79,134
access 139,217
call 135,217
chdir 140,217
chmod 141,217
escape 142,218
-exe 137,218
execl 144,218
exit 145,218
-exit 138,218
longjmp 146,220
oscall 136,221
-osret 136,221
-seterror 147,221
setjmp 148,221
setpath 149,222
showpath 150,222
system 64,151,223
trace 152,223
line numbering 55
#include 51
#include 55
LINK 8,214
-d address 8
-l 8,61,66,213
-n 8,67
-q 8,9,62,67,213
-r 68
-r<restoffine> 8,9
load text 19,22-23
low level i/o routines 79,125

terror 119,121,122,218
fflush 83,218
fgets 84,218
fgetc 85,218
fopen 79,86,218
fprintf 88,218
fputc 92,218
fputs 93,218
fread 94,219
freopen 96,219
fscanf 97,112,114,219
fseek 102,111,115,219
ftell 103,219
fwrite 104,219
getc 115,121,219
getchar 122,219
gets 105,219
getw 106,219
ferror 107,221
printf 108,221
putc 91,123,221
putchar 124,221
puts 109,221
putw 110,221
rewind 111,221
scanf 112,221
sprintf 113,222
sscanf 114,222
ungetc 102,115,223
string operations 79,164
strcat 165,222
strchr 166,222
strcmp 167,222
strcpy 168,222
strcspn 169,222
strlen 170,222
strncat 171,222

254

255
macros 178
macro substitution 48
#define 48
PSIDESQ 50
SIDESQ 50
SQUARE 50
#undef 49,50
markers 15
memory
  map 209
  operations 79,158
minus sign (-) 88
MODE 9,214
move text 19,28-29

\n 3

object code 65
  exiting 71
  standard error stream (stderr) 69,80,81
  standard input stream (stdin) 69,80,105,122
  standard output stream (stdout) 69,80,109
output device
  *FX3 80
over 25
overtype 15,18,19,24-25

path values, default 9
patterns 37
  . 37
  a 37
  # 37
  [xyz] 37
  a-z 38
  $ 38
  @ 38
  \ 38
  _ 38
  _ 38
specified libraries 64
    minlib 64
    smallib 64
    stdlib 8,63,64-65
    system 64,151,223
standard C 1
standard error stream 69
standard input stream 69
standard i/o routines 79-81,125
standard output stream 69
startup 7
status line 14,15
storage classes 3
string operations 164
structure member names 4
syntax checking 56
system calls 79,134
system command summary 213

 [TAB] 17,18,44
tab 19,25
text cursor 28
TRACE 9,215
type specifiers
    char 2
double 2
    int 2
    long [int] 2
    [long] float 2
    short [int] 2
    struct 2
    union 2
    unsigned [int] 2
    void 2,4,60
typedef 3

*unplug 13
user-defined types 3