USER GUIDE

C

User Guide

Beebug

This implementation of C was designed and written by David Allison.
This user guide was written by John Wallace and David Allison.

FIRST EDITION 1987

Published by Beebug Limited, Dolphin Place, Holywell Hill,
St. Albans, Herts. AL1 1EX, England. Telephone (0727) 40303

Contents

Appendices

1. Introduction

Introduction To C

C is a general purpose high level language which gives the programmer control over the low level workings of the computer. It is a fast and relatively compact language and has therefore become generally known as a systems programming language. In actual fact, its modern data structures, comprehensive set of operators and its general absence of restrictions make it suitable for almost any task.

C was written for the UNIX operating system by D. Ritchie, and the book *The C Programming Language* by Kernighan and Ritchie (Prentice-Hall 1978) describes the language in detail. Although Kernighan and Ritchie is not a formal national or international standard, it is generally accepted as 'standard' C. An ISO standard for C is currently under preparation, and features many improvements and extensions to the Kernighan and Ritchie standard. Some of these extensions are included in this implementation, and are detailed in Appendix B.

Beebug C features

Beebug C is an implementation of the language C based on the version described in the book *The C Programming Language* by Kernighan and Ritchie. The major features of this implementation are:

- Full Kernighan and Ritchie standard plus ISO extensions including:
    - new data types
    - function prototypes
    - initialisation of unions
    - new preprocessor control lines
    - local structure and union members
- Runs on a standard 32K BBC computer as well as the B+ and Master series.
- Works with both the DFS and ADFS, and only requires one disc drive.
- Full floating point mathematics, accurate to 7 significant figures.
- Extensive run-time library on disc, with some functions in ROM for fast, efficient code.
- Full support for the Acorn operating system (Mode, Osbyte, Vdu etc.).

- Libraries are searched only for those functions that are required in the program, thus producing compact code.
- Debugging facilities and helpful error messages.
- Compiler warning messages for non-portable features.
- Powerful command line interpreter.
- Full macro handling facilities.

A few minor differences exist between Beebug C and the Kernighan and Ritchie 'standard', and these are documented in Appendices B and C.

Beebug C technical summary

The following is a summary of the full specification.
Expressions: *, &, -, !, ~, ++, --, sizeof, ->, *, /, %, +, -, >>, <<, <, >, <=, >=, &, ^, |, &&, | |, ?:
Assignments: =, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=
Declarations: char (8bits), int (16 bits), short (8 bits), long (32 bits), float (32 bits), double (32 bits), unsigned, void, auto, static, extern, typedef, struct, union (inc. bit fields)
Statements: if, while, do, for, switch, case, default, break, continue, return, goto
Preprocessors: #define, #undef, #redef, #include, #if, #ifdef, #ifndef, #else, #endif, #line, #pragma
Library: over 90 library functions, plus 12 header files: h.stdio, h.stdlib, h.string, h.ctype, h.stdarg, h.stddef, h.setjmp, h.assert, h.math, h.call, h.limits, h.float

How to use this manual

This user guide is not intended to teach you the language C, but to explain the Beebug C system and how to compile and run C programs using it. You should have a basic knowledge of both the language C and the workings of the Acorn operating system to write C programs. The chapters in this guide deal with the following subjects:

Getting Started explains how to get the system up and running. It introduces each Beebug C command and describes the steps required to compile and run a simple C program.

Compiling C Source Text describes the compiler in detail, and explains the use of the optional qualifiers. C conditional compilation, error handling and debugging methods are also introduced.

Linking Object Code describes the linker and its optional qualifiers. This section also introduces the run-time library and user libraries.

Running C Programs tells you how to run the executable code, and explains how to use the optional qualifiers. The method for passing optional arguments to a C program is described.

Header Files & Library Functions lists the header files and all the library functions and macros, with a brief description and the syntax of each.

User Libraries describes how to use the library facility to produce user-defined run-time libraries.

Command Summary lists all Beebug C commands with their qualifiers, minimum abbreviations, and qualifier defaults. A summary of the library facility commands is also given.

Beebug C Extensions details the ISO extensions to the Kernighan and Ritchie standard.

Beebug C Limitations details the minor limitations of this implementation.

Summary of Library Functions lists all the functions and macros in alphabetical order. The function type is given, together with the name of the header file required to declare the function, and the page number on which a full description may be found.

Error Messages lists and explains all the errors that may be generated by this implementation.

Technical Information gives an insight into the workings of Beebug C, which may prove useful to advanced programmers.

Example Programs describes some example programs that may be useful to help you get started.

Conventions used in this manual

In this manual specific key presses such as the RETURN key are indicated thus:

`RETURN`

C commands are also shown in the same typeface:

`CLOSE`

If a word in the same typeface is shown in italics, it means the word is a parameter and should not be typed literally. In the example below:

```
MODE mode_number
```

you should type the command `MODE` followed by the actual mode number you require, and not the words *`mode_number`*.

Optional parameters are shown in square brackets thus:

```
COMPILE [qualifiers] filename
```

C program listings and screen output are shown as follows:

```
printf("C program");
```

Beebug C compatibility

Beebug C is compatible with the BBC Micro model B, B+, Master 128 and Master Compact. It may be used with the standard DFS (Disc Filing System) or the Acorn ADFS (Advanced Disc Filing System). Beebug C is not fully compatible with early versions of the DFS on the Master 128 (i.e. versions before 2.29). This is not a fault of Beebug C, but is due to the fact that the DFS does not close files correctly. The solution to this problem is to upgrade to version 2.29 of the DFS, or use the ADFS instead.You may find a sideways RAM version of the 2.29 DFS on your Master welcome disc. Alternatively, contact your local Acorn dealer for details of the upgrade.

To use Beebug C with the 6502 second processor a special version of the language ROM must be loaded into RAM. The new version is called `CSP` and is supplied on the library disc. To load the new version, simply type:

```
    *CSP
```

Once this file has been loaded Beebug C will operate as normal. Please note that both Beebug C ROMs must be fitted in your computer for this to work correctly.

2. Getting Started

This section describes how to install the C ROMs into your computer, and how to enter the C language. Each C command is introduced, and the steps required to compile and run a C program are described.

Installing the ROMs

Beebug C is supplied on two 16K ROMs and a dual-format disc containing the library of standard functions. The ROMs are labelled *lang* and *comp*, and both must be plugged into sideways ROM sockets for the software to operate. If you are unfamiliar with installing sideways ROMs, please contact us, or your local Acorn dealer for advice.

***Warning**: it is absolutely essential to insert the ROMs in the correct orientation. Failure to do so will destroy them.*

Once fitted, you can check that the ROMs are correctly installed by switching on the computer and typing:

```
    *HELP
```

A list of all the sideways ROMs in your computer will be displayed, and should include:

```
    Beebug C 1.0
```

The number following the name is the version number, and may be different to that shown above. Please make a note of the version number and always refer to it in any correspondence about this product.

Copying the library disc

The library disc supplied with C contains the standard library of functions, header files, and some example programs. Whenever you use the compiler or linker the system looks for your C programs on the default drive, and the header files and library functions in the default library. This means that normally all these files should be on the same disc. By changing the library drive (using `*LIB`), you can have a disc containing the header files and library functions in one drive, whilst keeping the other drive free for programs.

It is important that you make a backup of the library disc supplied with C before using the system. Use the normal `BACKUP` command to do this, and then store the original disc in a safe place and use the copy when compiling programs. To make a working copy of the library disc (i.e. a copy containing only the essential files for using C), type the following:

```
*COPY 0 1 $.*
*COPY 0 1 H.*
```

The above commands assume a dual disc drive is being used. To copy the disc using a single disc drive, copy the files to drive 0, inserting the correct source and destination discs when prompted.

You may use Beebug C with the ADFS by copying the contents of the directories $, E, and H to an ADFS disc. It is worth noting that each C program you write will require 3 files (*source, object* and *executable*), so you are limited to about 10 C programs per DFS disc (although in practice this number may be less since the standard library and header files may be stored on the disc). On the ADFS you can have up to 47 files per directory, and an almost unlimited number per disc!

Entering command mode

To enter C command mode, simply type:

```
*C
```

You should see the following message appear on the screen:

```
Beebug C
$
```

The symbol $ is the Beebug C prompt, and indicates that you may enter a command. In C, only a small set of commands is required:

```
CLOSE          COMMAND        COMPILE        LINK
MODE           REPORT         RUN
```

A description of each command is given below.

If you would like your computer to power-up automatically in C, install the ROM labelled *lang* in the highest priority ROM socket, or on a Master computer simply use the command:

```
*CONFIGURE LANG n
```

where n is the socket number containing the ROM labelled *lang*.

Beebug C commands

The following commands, and their associated qualifiers are recognised by the system. In the following list and throughout the manual, a string enclosed in square brackets denotes an optional part.

`CLOSE`
This command closes all open files, and may be useful if **BREAK** is pressed during compilation or linking. In these situations a file may be left open, and will be inaccessible until properly closed.

`COMMAND` ***`mode_number`***
This tells the system what to do if an unrecognised command is entered. The following modes are available:

mode 0 - print normal error message (default setting)
mode 1 - attempt to `*RUN` the file (passes it to OSCLI)
mode 2 - attempt to `RUN` the C program

You can also specify the mode on entry to C by issuing a number as a parameter after the `*C` command. For example:

```
*C 2
```

will enter C and set the command mode to 2. This means that the system will treat any unrecognised command as a C program, and will attempt to execute it.

`COMPILE [`*`qualifiers`*`]` ***`filename`***
Compile the C source text ***`filename`***. This command has a number of optional qualifiers which are described in detail in section 3 of this guide.

`LINK [`*`qualifiers`*`]` ***`filename`*** **`[,...]`**
Link the object ***`filename`*** to any other object files and to the standard library. This command has a number of optional qualifiers which are described in detail in section 4 of this guide.

`MODE` ***`mode_number`***
Set screen mode.

`REPORT`
Displays the last error message that was reported (see page 32).

`RUN [`*`qualifiers`*`]` ***`filename`*** **`[`*`arg1 arg2`*` ...]`**
Execute the C program ***`filename`***. This command has a number of optional qualifiers which are described in detail in section 5 of this guide.

In addition to the above commands, all the usual operating system star (*) commands may be entered whenever the dollar prompt ($) is displayed.

Entering C commands

Each command may be typed in either upper or lower case, and may be abbreviated in two ways: simply by typing enough characters to identify uniquely the command, or with a dot (.) after the abbreviation. For example the **MODE** command may be abbreviated to any of the following:

```
M
MO
MOD
M.
MO.
MOD.
```

If you type a command that cannot be uniquely identified, the system will print an error message. A list of the minimum abbreviations allowed is given in the command summary in Appendix A.

Command parameters

Most of the commands listed above require a certain number of parameters which must be supplied (the exceptions are **CLOSE** and **REPORT** which have no parameters). If the parameter is not supplied, the system will prompt for the missing parameter. For example, if you enter the **MODE** command without specifying a screen mode, the system would respond with a prompt asking for the required screen mode to change to:

```
MODE
Mode:
```

Parameters are typed on the command line after the command name, and each one must be separated by a space. Some commands allow a list of values to be supplied for a single parameter, and in this case each item should be separated by a comma (,). For example, the **LINK** command allows the parameter to be a list of filenames that are to be linked together:

```
LINK file1,file2,file3
```

A parameter may be either a number or a string of characters. A string need not be enclosed in quotes (") if it does not contain a separating character (i.e. a space, comma or slash). Strings not enclosed in quotes are automatically converted to upper case. A numeric parameter may be entered in either decimal, or in hexadecimal if prefixed by an ampersand (&).

Command qualifiers

The commands **COMPILE**, **LINK** and **RUN** accept a number of optional qualifiers. These are special keywords which modify the operation of the command in some way. The qualifiers may be placed at any position on the command line after the command itself, and are identified by their first character which must be a slash (/). A qualifier may be typed in upper or lower case, and may be abbreviated by supplying enough characters to uniquely identify it. If the abbreviation does not specify a unique qualifier, the first one the system matches will be used.

A qualifier may also require a value to be supplied with it. If it does, the value should be preceded by an equals character (=). For example, the **COMPILE** command accepts a qualifier called **LSPACE** which must take a numeric value:

```
COMPILE/LSPACE=&500 file1
```

This example compiles the file **file1** and sets **LSPACE** (Literal Space) to &500. Certain qualifiers are boolean - that is they can be either true or false. To specify that the qualifier should take a false value, precede the qualifier by the word **NO**. For example, the **RUN** command allows a qualifier **TRACEBACK**. This is a boolean qualifier and can be used as follows:

```
RUN/TRACEBACK
```

or

```
RUN/NOTRACEBACK
```

As with parameters, a qualifier may allow a list of values to be supplied. For example, in the **COMPILE** command there is a qualifier **DEFINE** which takes a list of string values:

```
COMPILE/DEFINE=DEBUG,LIST,VERSION1 file1
```

This command compiles the file **file1** and defines three macros **DEBUG**, **LIST** and **VERSION1**.

Compiling and running a C program

The rest of this section describes how to get a C program entered and running. The example program is called **welcome** and is included on the library disc to save you having to type it in. The source text for the sample program **c.welcome** is:

```
/* Beebug C Welcome Program */

#include <h.stdio>

main()

{
printf("Welcome to Beebug C\n");
}
```

Although the example program is very small, the steps taken to compile and run it are basically the same for all C programs no matter how long or complicated they may be. There are four stages in compiling and running a C program:

1. Prepare the C source text using a word-processor or text editor.

2. Compile the source text to object code.

3. Link the object code to other objects and to the library.

4. Run the executable C program.

Preparing the C source text

The C source text should be entered using a word-processor or text editor, and then saved to disc, normally in directory **c**. Most word-processors and text editors may be used, but those allowing editing in 80 columns will be more useful. It is important that you do not insert any special formatting codes in your text (no justification, formatting etc.), and that when you save it to disc you save it as a pure ASCII file. If you use View or InterWord you may use **TAB** to indent loops to create a readable listing. InterWord operates slightly differently to the other word-processors, but may be used successfully by following these simple rules:

1. Switch off paging
2. Move the left margin to the extreme left-hand edge of the screen
3. Save the text using the spool option number 8

Compiling the C program `c.welcome`

To compile the example program enter and save it as **`c.welcome`** (this file is supplied on the library disc if you prefer not to type it in). Ensure that you are in C command mode, and type:

```
COMPILE welcome
```

Notice that we do not specify the directory name since the compiler always searches directory **c** for it. If all is well the compiler will print a listing of the program on the screen as it compiles it. A new file will be produced called **`o.welcome`** which is the object code suitable for linking in the next step of the operation. When compilation is complete a confirmation message will be displayed, and the usual prompt should re-appear.

Linking the object `o.welcome`

This links any required library functions to the object code **`o.welcome`**. In this example the function *printf* is required to be linked from the library. The linker may also be used to link a number of user programs together to create much larger programs. To link **`o.welcome`** type:

```
LINK welcome
```

Note again that we do not need to specify the directory **o**. During linking the system will access the library and header files, and produce an executable program called **`e.welcome`**. When linking is complete the usual $ prompt will re-appear.

Running the executable program `e.welcome`

To run the **`welcome`** program type:

```
RUN welcome
```

Note again that we do not need to specify the directory **e**. This command will load the executable program **`e.welcome`** and, if all the previous steps have been followed correctly, execute it. The program is very small, and simply prints a message on the screen. To run the program again you will have to enter the command **`RUN welcome`** again in full.

Automatic compilation

In the example above, it would have been much easier to use a single command to compile, link and run the program **`welcome`**. A special utility on the library disc called **RUNC** allows just this, by programming four function keys with the following definitions:

```
f0  COMPILE/NOLIST filename  LINK filename  RUN filename
f1  COMPILE/NOLIST filename
f2  LINK filename
f3  RUN filename
```

To use this utility type:

```
*RUNC filename
```

where ***filename*** is the name of the C program you wish to compile. ***RUNC** automatically enters the language C, so it may be called from Basic or most other languages. To compile, link and run the example program **welcome** type:

```
*RUNC welcome
```

and press function key **f0**. Note that the directory name is not specified. If ***RUNC** is called without a filename, key **f0** will not be defined, and the other keys will prompt for a filename when pressed.

Please note that this utility is supplied on disc, and will only work as described above if the file **RUNC** is in the currently selected library. The function key definitions will normally remain active until they are re-defined using ***RUNC**, or until **CTRL BREAK** is pressed.

3. Compiling C Source Text

The last section described how to use the **COMPILE** command. This section describes the compiler in more detail and explains how to use the optional qualifiers. The **COMPILE** command has the following syntax:

```
COMPILE [qualifiers] filename
```

The optional qualifiers which modify the operation of the compiler, are:

```
/[NO]OBJECT[=filename]
/[NO]LIST[=filename]
/[NO]WARNINGS
/[NO]PORTABLE
/[NO]DEBUG
/DEFINE=macro_name[,...]
/LSPACE=buffer_size
/[NO]OPTIMISE
```

Compiling source text

Once completed, a C program should be saved to disc, normally in directory **c**. This file is called the source file. The compiler takes this source file and compiles it to an intermediate code called object code. During compilation any errors in the source code are identified, and only when all errors have been corrected, will the object code be generated. The object code is then linked to produce the final executable code (see sections 4 and 5 for further details).

To compile a source file, type:

```
COMPILE filename
```

The compiler looks for the source file ***filename*** in sub-directory **c** of the current directory. For example, if the current directory is $ (which it normally is) and you type:

```
COMPILE general
```

the compiler looks for the source file **$.c.general**. You may of course, enter a specific pathname if required. For example, to compile the source file **f.general**, simple type:

```
COMPILE f.general
```

If the compiler cannot find the source file, it will display an error message. When compilation is complete, an appropriate message is displayed.

Destination of object code

The object code generated by the compiler is normally saved in sub-directory **o** of the current directory. So, the command:

```
COMPILE general or COMPILE f.general
```

will generate the object code `o.general`. If you are using the ADFS, the error message `Not found` will be generated if directory **o** does not exist. You may specify a different object filename by using the optional qualifier **OBJECT**. For example, the command:

```
COMPILE/OBJECT=d.oldgen general
```

will compile the source file `c.general` and produce the object file `d.oldgen`. For further details about the use of optional qualifiers, please refer to section 2.

Compilation listing

During compilation the C source text is listed to the screen. The compiler automatically assigns a number to each line of the listing, so if an error occurs, the line may be identified. Line numbers start with line 1 at the beginning of the source text, and continue to the end of the text. To compile a program without producing a listing, use:

```
COMPILE/NOLIST filename
```

Alternatively, the listing may be directed to a file instead of the VDU. For example:

```
COMPILE/LIST filename
```

will compile ***filename*** and produce a file called ***l.filename*** containing a complete listing of the program, including line numbers and any associated error messages. Alternatively, the listing may be directed to any file specified. For example:

```
COMPILE/LIST=debug filename
```

will compile ***filename***, and produce a file called **debug**.

Error messages

If the compiler cannot understand the source code, an error message is displayed followed by the offending line of source text. Object code is not generated. In such a circumstance the compiler displays the appropriate error message, followed by the offending line of source text. The compiler then pauses allowing the user to press either **ESCAPE** to exit, or any other key to continue compiling. If the user chooses to continue, the compiler continues parsing the source text so that any other errors are located. Once one error has been detected, other errors may be reported later as a direct consequence of it. For example, if a vital character is omitted, the compiler may continue parsing from an unsuitable position in the code thus reporting further errors.

The following program contains two errors:

```
/* Error example */

#include <h.stdio>

main()

{
int ch;
float ch;
while ((c = getchar()) != EOF);
printf("%d ", ch);
}
```

The following output will be displayed when it is compiled:

```
Beebug C Compiler V1.0

    1  /* Error example */
    2
    3  #include <h.stdio>
    1  # pragma 1   /* list off */

    4
    5  main()
    6
    7  {
    8  int ch;
    9  float ch;

** Error - Multiply defined local symbol at line 9 in
C.ERREX
float ch;
```

```
    10  while ((c = getchar()) != EOF);

** Error - Undeclared identifier at line 10 in C.ERREX
while ((c = getchar()) != EOF);

    11  printf("%d ", ch);
    12  }

Compilation completed with 2 errors and 0 warnings
```

A complete list of all error messages is given in Appendix E.

Warning messages

Warnings simply notify the user that the program contains non-portable or undesirable code. In such circumstances the compiler displays the appropriate warning message, followed by the offending line of source text. The compiler then pauses allowing the user to press either **ESCAPE** to exit, or any other key to continue compiling. After such a situation, object code is still generated. A complete list of all warning messages is given in Appendix E. The following program is an example of a valid C program that will produce a warning:

```
/* Warning Example */

#include <h.stdio>

main()

{
unsigned long this_is_a_very_long_variable_name;
}
```

The following output will be displayed when it is compiled (the display may vary slightly on different systems):

```
Beebug C Compiler V1.0

    1   /* Warning Example */
    2
    3   #include <h.stdio>
    1   # pragma l   /* list off */

    4
    5   main()
    6
    7   {
    8   unsigned long this_is_a_very_long_variable_name;
```

```
** Warning - Symbol name truncated at line 8 in
C.WARNEX
int this_is_a_very_long_variable_name;

    9   }

Compilation completed with 0 errors and 1 warning
```

Here the compiler is warning the user that it has truncated the long symbol name in line 8 (symbol names are limited to 31 characters - see Appendix C). In this instance the warning can be ignored and linking may continue as normal. Warning messages may be switched off with the optional qualifier **WARNINGS**. For example, try compiling the above program again using the command:

```
COMPILE/NOWARNINGS warnex
```

The compiler may generate a special type of warning to indicate that code is non-portable. Non-portable code contains extensions to C that are not part of the Kernighan and Ritchie standard. Normally the compiler is in 'non-portable' mode, and does not generate these warnings. If you want to write portable code that will run on other C systems, use the optional qualifier **PORTABLE**. For example, try the above program again using the command:

```
COMPILE/PORTABLE warnex
```

The following is displayed:

```
Beebug C Compiler V1.0

    1   /* Warning Example */
    2
    3   #include <h.stdio>
    1   # pragma l   /* list off */

    4
    5   main()
    6
    7   {
    8   unsigned long this_is_a_very_long_variable_name;

** Warning - Type specifier is non portable at line 8
in C.WARNEX
unsigned long this_is_a_very_long_variable_name;

** Warning - Symbol name truncated at line 8 in
C.WARNEX
unsigned long this_is_a_very_long_variable_name;

9  }

Compilation completed with 0 errors and 2 warnings
```

Notice that there is now another warning message at line 8. It warns the user that the non-portable type specifier **unsigned long** has been identified.

The DEBUG qualifier

If a run-time error occurs, the line number, function and source file in which the error occurred, are displayed. This is possible because during compilation the compiler stores this debugging information within the program. This information is used to great effect with the qualifier **TRACEBACK**, explained in section 5. The optional qualifier **DEBUG** controls the generation of the debugging information, allowing more compact code to be produced. The length of the executable code for the welcome program **e.welcome** is &66. Try compiling it again without debugging code:

```
COMPILE/NODEBUG welcome
```

After linking, the executable code is only &57 bytes long. This represents quite a significant reduction in program length. The drawback of using **NODEBUG** is that if a run-time error occurs, there is no line number, function name or source file name in the error message. As you will see in section 4, debugging code can also be removed during linking. This is a powerful option allowing you to compile programs with debugging information switched on, and when fully tested, link them with **NODEBUG** i.e. debugging switched off. This is particularly useful for programs that consist of a lot of parts to be linked, as to switch debugging on or off, it is only necessary to re-link with **LINK/DEBUG** or **LINK/NODEBUG**.

Defining macros

The optional qualifier **DEFINE** allows a number of macro names to be defined at compile time. The syntax of the qualifier is:

```
COMPILE/DEFINE=MACRO1,MACRO2,MACRO3 filename
```

where **MACRO1**, **MACRO2**, and **MACRO3** are the defined macros, and ***filename*** is the file to be compiled. The main purpose of defining macros, is to control conditional compilation i.e. which parts of the source text are to be compiled. It is beyond the scope of this user guide to explain conditional compilation fully, but the following example shows a common application:

```
/* Conditional compilation */

#include <h.stdio>

main()
{
#ifdef SPECIAL
printf("Special code\n");
#else
printf("Normal code\n");
#endif
}
```

When compiled the condition at line 7 fails (because SPECIAL is not defined), therefore the second expression after #else is compiled. When run, the program prints the message:

```
Normal code
```

If the macro name SPECIAL is defined at compile time:

```
COMPILE/DEFINE=SPECIAL filename
```

the condition at line 5 is true, and the program prints:

```
Special code
```

Conditional compilation has many uses, but is especially useful in program testing and debugging. All the standard Kernighan and Ritchie directives are implemented i.e. **#if**, **#ifdef**, **#ifndef**, **#else**, and **#endif**.

Setting literal space

Beebug C sets aside &300 (768) bytes for storage of literals (literals are strings enclosed in quotes). This should be quite sufficient for small to medium applications, but if your program contains a lot of literals it may be changed with the optional qualifier **LSPACE**. For example:

```
COMPILE/LSPACE=1000 filename
```

sets literal space to 1000 (decimal) bytes. Literal space may be specified in hexadecimal by prefixing the number with an ampersand (&). For example:

```
COMPILE/LSPACE=&500 filename
```

sets literal space to &500 bytes. An error message is displayed if there is not enough literal space.

Program optimisation

Beebug C carries out limited program optimisation at compile time. If a program contains expressions containing only constants, then they will be evaluated at compile time, rather than run time. This leads to faster and more compact code. In some specialist applications it may be necessary to switch off optimisation. This is done with the optional qualifier `OPTIMISE`, as follows:

```
    COMPILE/NOOPTIMISE filename
```

4. Linking Object Code

This section describes the `LINK` command, and its optional qualifiers. The syntax of `LINK` is:

```
    LINK [qualifiers] filename[,...]
```

The optional qualifiers which modify the operation of the linker, are:

```
    /[NO]EXECUTABLE[=filename]
    /[NO]LIBRARY[=filename[,...]]
    /[NO]DEBUG
    /ORIGIN=start_address
    /[NO]STANDALONE
```

Each qualifier will be described in detail in this section.

Linking object code

Once a C program has been compiled to object code, it must be linked to produce executable code. The command to run the executable code is described in the next section. The linker's main task is to scan the object code for any unresolved functions, and load them in from the run-time library. Unresolved functions are those functions not defined in the main source text file, such as **printf**, **mode**, and **malloc**. To ensure that the final code is compact, the linker loads only those functions required by the program, and not the complete library. The linker is also used to link together a number of object files, to produce a single executable file. Please note that all C programs must be linked - even trivial programs that do not access library functions!

To link an object file, type:

```
    LINK filename
```

The linker looks for the object file ***filename*** in sub-directory `o` of the current directory. For example, if the current directory is $ (which it normally is) and you type:

```
    LINK applics
```

the linker looks for the object file `$.o.applics`. You may, of course, enter a specific pathname if required. For example, to link the object file `f.applics`, simply type:

```
LINK f.applics
```

If any errors occur during linking, an appropriate error message is displayed, and executable code is not produced.

The following example shows how the linker is used to link together a number of object files:

```
LINK prog1,prog2,prog3
```

This command links together the object files **`prog1`**, **`prog2`** and **`prog3`**. The name of the executable file will be **`e.prog1`** i.e. in sub-directory **`e`** using the first name in the list. This facility is particularly useful in modular programming. For example, **`prog1`** could be the main program, with **`prog2`** and **`prog3`** files containing all of the detailed functions.

Destination of executable code

The executable code generated by the linker is normally saved in sub-directory **e** of the current directory. So, the command:

```
LINK applics
```

or

```
LINK f.applics
```

will generate the executable code **`e.applics`**. If you are using the ADFS, the error message `Not found` will be generated if directory **e** does not exist. You may specify a different executable code filename by using the optional qualifier **EXECUTABLE**. For example, the command:

```
LINK/EXECUTABLE=d.general applics
```

will link the object file **`o.applics`** and produce an executable file **`d.general`**. The same qualifier can be used to prevent executable code from being generated i.e. linking will take place as usual but the executable code is not produced. For example:

```
LINK/NOEXECUTABLE game
```

Library of functions

During linking, Beebug C loads from disc any unresolved functions in the C program. These functions are stored in the run-time library *rtlib*, and are documented in section 6. Beebug C always searches the current library directory for **`rtlib`**. The library directory is normally :0.$ (drive 0, directory $), but it may be changed with the command **`*LIB`**. For example, you may wish to store the library in drive 1, leaving drive 0 free for C programs. To do this, type:

```
*LIB :1.$
```

Please note that other Beebug C files such as **RUNC** (see section 2), and all the header files (see section 6) should also be in the library directory. If you change the library, be sure to transfer all of these files as well as *rtlib*.

The optional qualifier **LIBRARY** is provided to allow the user to specify other run-time libraries in addition to **`rtlib`**. These libraries are called User Libraries, and can be created using the library facility described in section 7. A user library is particularly useful for holding frequently used functions. For example, if you regularly write programs involving statistics, you could create a library of statistics functions (using the library facility described in section 7). If this library was called **`stats`**, you could link the program **`survey`** with:

```
LINK/LIBRARY=stats survey
```

This would search library **`stats`** for any unresolved functions in the program **`survey`**. Please note that **`stats`** is a special file generated by the library facility, and is not simply object code. A number of user libraries may be linked, by listing their names separated by commas. For example:

```
LINK/LIBRARY=rtlib,stats1,stats2,stats3 survey
```

The advantage of using libraries in this way, is that only those functions that are required are appended to the executable code. If you simply linked the program **`survey`** to an object file **`funcs`** containing all the functions, then the whole of this file would be appended. For example:

```
LINK survey,funcs
```

would produce a much longer executable file. The library facility allows the run-time library to be extended, so if you wish, you can simply add new functions to it, instead of creating a new library.

The **DEBUG** qualifier

The **DEBUG** qualifier removes debugging information from the object files before generating the executable code. This has the effect of producing more compact code, at the expense of detailed error messages. The qualifier is used as follows:

```
LINK/NODEBUG game
```

For further details on the use and the effects of **NODEBUG**, please refer to section 3.

Setting the origin

Beebug C executable code is not relocatable and is executed at a fixed address - normally PAGE (OSHWM). The optional qualifier **ORIGIN** allows the execute address to be specified. This is useful to produce code for different systems which may have different PAGE settings. For example, if you write a C program on a Master 128 (PAGE=&0E00) and would eventually like to run it on a BBC B (PAGE=&1900), **ORIGIN** must be set to the highest PAGE value:

```
LINK/ORIGIN=&1900 utils
```

This will produce C code which loads and executes at &1900. The address is specified in decimal, or in hexadecimal if prefixed with an ampersand (&).

Producing stand-alone code

Normally C programs must be executed from within C with the **RUN** command (detailed in the next section). Using the **STANDALONE** qualifier you may produce completely independent code. For example:

```
LINK/STANDALONE utils
```

will produce a stand-alone version of the program **utils**, that may be executed at any time by typing:

```
*utils
```

The advantage of this type of file is that C programs will run on machines not fitted with Beebug C.

Please note that for this option to work, a special file called *rtsys* must be present in the current library. This file is not supplied as standard with Beebug C and is available at an extra charge.

5. Running C Programs

This section describes how to use the **RUN** command to execute the linked code. The **RUN** command has the following syntax:

```
RUN [qualifiers] filename [arg1 arg2 ...]
```

The optional qualifiers are:

```
/[NO]TRACEBACK
/INPUT=filename
/[NO]OUTPUT[=filename]
/[NO]ERROR[=filename]
```

Please note that this **RUN** command is not the same as the Basic command **RUN**, nor the operating system command ***RUN**. C programs can only be executed on a computer fitted with Beebug C, and only from within C command mode (unless the stand-alone option has been used - see page 28).

Running a C program

The full syntax of the **RUN** command is given above, but normally you will only need to use the command in its simplest form:

```
RUN filename
```

This simply runs the program ***filename***. For example:

```
RUN general
```

will run the program **general**. Note that the directory is not specified since directory **e** is always used for executable code. So in the example above, the command actually looks for the executable file **$.e.general**. You may of course, specify a different directory if required. For example, to execute the program **q.general**, simply type:

```
RUN q.general
```

If the command cannot find the executable file, or if the file specified is not executable, an appropriate error message will be displayed. Please note that to re-run a program it is necessary to enter the command again in full - you cannot run a program again by simply typing **RUN**.

Passing arguments to C

A number of string arguments may be passed to a C program by listing them after the name of the executable code. For example:

```
RUN argtest arg1 arg2 arg3 .... argn
```

will pass the strings ***arg1***, ***arg2***, ***arg3*** etc. to the C program **argtest**. The following C program demonstrates how C interprets these arguments. The integer argc contains the number of arguments, and the pointer *argv[] points to an array containing the arguments. The first array element argv[0] contains the program name **argtest**, argv[1] contains ***arg1***, argv[2] contains ***arg2*** etc.

```
/* Passing arguments to C */

#include <h.stdio>

main(argc, argv)

int argc;
char *argv[];
{
int arg;

printf("There are %d arguments.\n", argc-1);

for (arg = 1; arg < argc; ++arg)
printf("Argument %d = %s\n", arg, argv[arg]);
}
```

Enter and save the above program, with the name **argtest**. Compile and link it in the normal way, then run it as follows:

```
RUN argtest UK France Germany USA Italy Belgium
```

The program should output the following:

```
There are 6 arguments.
Argument 1 = UK
Argument 2 = France
Argument 3 = Germany
Argument 4 = USA
Argument 5 = Italy
Argument 6 = Belgium
```

Debugging with TRACEBACK

TRACEBACK is an optional qualifier to the **RUN** command that causes more detailed debugging information to be displayed when **ESCAPE** is pressed or when errors occur during program execution. The following program may be used to demonstrate its use. Enter the following C program and save it with the name **trdemo**.

```
/* TRACEBACK Demonstration */

#include <h.stdio>

main()
{
printf("function = main\n");
func1();
}

func1()
{
printf("function = func1\n");
func2();
}

func2()
{
printf("function = func2\n");
printf("Now press ESCAPE");
for (;;);
}
```

Now compile and link it in the usual way. Do not use the qualifier **NODEBUG**, as this prevents debugging information from being displayed (see **DEBUG** in sections 3 and 4). The program itself simply calls two functions, and then goes into an infinite loop. Run the program as follows:

```
RUN trdemo
```

then press **ESCAPE** to exit from the infinite loop. The following information should be displayed:

```
Escape at line 21
  in function 'func2' in file 'C.TRDEMO'
```

This gives the line number that was being executed when **ESCAPE** was pressed, the function containing that line, and the file containing that function. Now try using the qualifier **TRACEBACK**:

```
RUN/TRACEBACK trdemo
```

Again, press **ESCAPE** to exit from the infinite loop. The following information should be displayed:

```
Escape at line 21
  in function 'func2' in file 'C.TRDEMO'

  A = 00000010  B = 03001F00  PC = 1F9A
ASP = 00        FSP = 09       VSP = 7BF8

File       Function          Line

C.TRDEMO   func2             21
C.TRDEMO   func1             14
C.TRDEMO   main              8
```

This displays the usual error message together with the values of the internal C registers, and the active functions. These register values are only of interest to advanced C users, and may have different values to those shown above (a detailed description of these is given in Appendix F).

The active functions are those functions that have been called to reach the current line. In this example the display shows that **ESCAPE** was pressed at line 21 in function **func2**, which was called from function **func1** at line 14. This in turn was called from function **main** at line 8. In addition to this information, the source file containing each function is displayed. This will prove useful if a number of source files have been linked together.

The **REPORT** command can be used at any time to display the last error message reported. The format will be the same as the original message.

Redirecting the standard streams

Three optional qualifiers are available to redirect the standard input, output and error streams. These three streams are automatically declared in the header *h.stdio* as **stdin**, **stdout** and **stderr**.

stdin is the standard input stream as set by the OS command ***FX2**. This is normally the keyboard.

stdout is the standard output stream as set by the OS command ***FX3**. This is normally the VDU.

stderr is the standard error stream, to which user generated errors may be sent. This is normally the VDU.

Redirecting stdin

The standard input stream **stdin** may be redirected with the optional qualifier **INPUT**. For example:

```
RUN/INPUT=data redir
```

will redirect the input stream to come from the file **data**. This means that input will come from the file **data** and not the keyboard. The following program may be used to demonstrate this. Enter the program and save it with the name **redir**, then compile and link it in the usual manner.

```
/*Redirect Stream Example */

#include <h.stdio>

main()
{
char input[];
scanf("%s", input);
printf("%s\n", input);
}
```

This program simply gets characters from the input stream and puts them to the output stream. The normal input stream is connected to the keyboard, and the output stream to the VDU. This program therefore, prints on the screen anything that is typed at the keyboard. The program terminates when **RETURN** is pressed. You can test this by entering:

```
RUN redir
```

and typing a few characters followed by **RETURN**. Now type the following:

```
RUN/INPUT=c.redir redir
```

This should print the following:

```
/*Redirect
```

The input stream has now been redirected to come from a file instead of the keyboard. In this case the file is **c.redir**, which is the C source file for this example. Notice that only the first word of the source file has been output. This is simply because the input from scanf terminates at the first 'white space' character it receives, which is at the end of the first word.

Redirecting stdout

The standard output stream **stdout** may be disconnected or redirected with the optional qualifier **`OUTPUT`**. Try the following examples with the file **`redir`** from the last example:

```
RUN/NOOUTPUT redir
```

This will disconnect the output stream. Anything sent to **stdout** will not be displayed. The following example is more useful:

```
RUN/OUTPUT=outfile redir
```

This will redirect all output to the file **`outfile`**. Now anything printed to the VDU will not appear on the screen, but will be written to the file **`outfile`** in the default directory.

Redirecting stderr

The standard error stream **stderr** may be disconnected or redirected with the optional qualifier **`ERROR`**. The standard error stream may be disconnected with:

```
RUN/NOERROR redir
```

or redirected to a file:

```
RUN/ERROR=errfile redir
```

The following example shows how to send a string to the standard error stream.

```
/* Standard Error Stream Example */

#include <h.stdio>

main()
{
fprintf(stderr, "This is an error!\n");
}
```

Please note that the standard error stream is for user-generated errors and text. System run-time errors are always sent to the VDU.

6. Header Files & Library Routines

Header files

Beebug C is supplied with 12 header files which declare the functions in the standard library *rtlib*, and a number of macros. The header files are:

h.stdio
h.assert
h.call
h.stdlib
h.setjmp
h.math
h.string
h.stdarg
h.limits
h.ctype
h.stddef
h.float

To use any functions or macros defined in these headers you should **`#include`** the appropriate header file in the C source text. This may be done in two ways:

```
#include <h.stdio>
```

or

```
#include "h.stdio"
```

In the first method the header name is enclosed in angle brackets (<>), and causes the current library directory to be searched for the header file. This method is generally used for the standard header files listed above.

In the second method the header name is enclosed in quotes ("), and causes the current directory to be searched. If the header file cannot be found in the current directory, the library directory is searched. This method is generally used for user-defined header files.

The rest of this section describes in detail each function/macro. The information given includes: the type of definition, a synopsis giving the function/parameter types, a description, and in many cases a simple example showing how the function or macro is used. They are grouped according to the header file that declares them, and are in alphabetical order. An alphabetic summary of all functions and macros is given in Appendix D.

h.stdio - Standard Input/Output routines

The header h.stdio declares all the input/output functions and macros to provide C programs with an interface to the VDU, keyboard, printer, disc drives etc.

bget

Type :function
Synopsis :`#include <h.stdio>`

```
int bget(FILE *stream);
```

Description :The **bget** function obtains the next character from the input stream. It is identical to the function **fgetc**, but does not convert carriage return characters (**fgetc** converts the ASCII character 13 to ASCII 10 for compatibility with other systems).

```
int c;
c = bget(str1);
```

bput

Type :function
Synopsis :`#include <h.stdio>`

```
int bput(FILE *stream, char c);
```

Description :The **bput** function writes the character specified by **c** to the output stream. It is similar to the function **fputc** but does not convert carriage return characters.

```
char c;
bput(str1, c);
```

clearerr

Type :macro
Synopsis :`#include <h.stdio>`

```
#define clearerr(stream)
```

Description :This macro is not defined in this implementation, and has no function. It has been included to ensure compatibility with other implementations.

EOF

Type :macro
Synopsis :`#include <h.stdio>`

```
#define EOF (-1)
```

Description :The **EOF** macro expands to a negative integral constant that is returned by several functions to indicate end-of-file.

fclose

Type :function
Synopsis :`#include <h.stdio>`

```
int fclose(FILE *stream);
```

Description :The **fclose** function causes the stream pointed to by **stream** to be flushed and the associated file closed. Any buffered data for the stream is written out. This function returns zero if the stream is successfully closed, or nonzero if there are any errors, or if the stream was already closed.

```
fclose(str1);
```

feof

Type :function
Synopsis :`#include <h.stdio>`

```
int feof(FILE *stream);
```

Description :The **feof** function returns nonzero if end-of-file has been detected for stream.

```
FILE *strm1;
int c;
while(foef(strm1) == 0)
c = getc(strm1);
```

ferror

Type :macro
Synopsis :`#include <h.stdio>`

```
#define ferror(stream) 0
```

Description :The **ferror** macro expands to zero. It is has no function in this implementation, and is included for compatibility with other systems.

fflush

Type :function
Synopsis :`#include <h.stdio>`

```
int fflush(FILE *stream);
```

Description :The **fflush** function causes any unwritten data for the specified stream to be written to the file. The stream remains open. This function returns zero if successful, otherwise nonzero if a write error occurs.

```
fflush(str1);
```

fgetc

Type :function

Synopsis :`#include <h.stdio>`

```
int fgetc(FILE *stream);
```

Description :The **`fgetc`** function obtains the next character (if present) from the input stream pointed to by **`stream`**, and advances the file pointer one character. This function returns the next character, or **`EOF`** if the stream is at end-of-file. In Beebug C **`fgetc`** is identical to **`getc`**.

```
int c;
c = fgetc(str1);
```

fgetpos

Type :function

Synopsis :`#include <h.stdio>`

```
int fgetpos(FILE *stream, fpos_t *pos);
```

Description :The **`fgetpos`** function stores the current value of the file pointer for the stream pointed to by **`stream`** in the object pointed to by **`pos`**. The value stored contains unspecified information usable only by the **`fsetpos`** function for repositioning the file pointer. If successful, **`fgetpos`** returns zero.

```
int store, err;
err = fgetpos(str1, store);
```

fgets

Type :function

Synopsis :`#include <h.stdio>`

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

Description :The **`fgets`** function reads up to **`n`** characters from the stream pointed to by **`stream`** into the array pointed to by **`s`**. No additional characters are read after a new-line character (which is retained) of after end-of-file. A null character is written to **`s`** immediately after the last character. This function returns **`s`** if successful. If end-of-file is encountered and no characters have been read into the array, or if a read error occurs, the contents of the array remain unchanged and a null pointer is returned.

```
char buffer[255];
if (fgets(buffer, 32, strm1) == NULL)
printf("Read error\n");
else
printf("%s\n", buffer);
```

FILE

Type :macro

Synopsis :`#include <h.stdio>`

```
#define FILE short
```

Description :**`FILE`** is an object type capable of recording all the information needed to control a stream.

fopen

Type :function

Synopsis :`#include <h.stdio>`

```
FILE *fopen(char *filename, char *mode);
```

Description :The **`fopen`** function opens the file whose name is the string pointed to by **`filename`**, and associates a stream with it. The argument **`mode`** points to a string that indicates the type of access for which the file is being opened.

"r"	open file for reading
"w"	create file for writing, or truncate to zero length
"a"	append; open file or create for writing at end-of-file.
"r+"	open file for update (reading and writing)
"w+"	create file for update, or truncate to zero length
"a+"	append; open file or create for update, writing at end-of-file

This function returns a pointer to the stream. If the open operation fails, **`fopen`** returns a null pointer.

```
FILE *str1;
str1 = fopen("c.welcome", "r");
```

fpos_t

Type :macro

Synopsis :`#include <h.stdio>`

```
#define fpos_t long
```

Description :**`fpos_t`** is an object type capable of recording all the information needed to specify uniquely every position within a file.

fprintf

Type :function

Synopsis :`#include <h.stdio>`

```
int fprintf(FILE *stream, char *format, ...);
```

Description :The **`fprintf`** function writes output to the stream pointed to by **`stream`**, under control of the string pointed to by **`format`**. The format string specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behaviour is undefined. If the format is exhausted while arguments remain, the extra arguments are evaluated but otherwise ignored.

The format is a character string consisting of ordinary characters which are sent unchanged to the output stream, and conversion specifications, each of which results in fetching zero or more arguments. Each conversion specification is introduced by the character %, followed by:

- a minus sign which specifies left justification of the converted argument in its field
- an optional digit string specifying the minimum field width. If the converted value has fewer characters than the field width, it will be padded on the left (or right if the left justification flag is given) to make up the field width. This string may be replaced by an asterisk * instead of a digit string. In this case an argument supplies the field width. The argument supplying the field width should appear before the argument to be converted. The width can take values -255 to 255.
- a period, which separates the field from the next digit string
- an optional precision which specifies:
    - the number of digits after the decimal point character for **e**, E and **f** conversions
    - the maximum number of significant digits for the **g** and G conversions
    - the maximum number of characters to be written from a string in **s** conversion

The precision takes the form of a decimal integer, and if omitted, it is treated as zero. This string may be replaced by an asterisk * instead of a digit string. In this case an argument supplies the precision width. The argument supplying the precision width should appear before the argument to be converted. Precision can take values 0 to 255.

- an optional l specifying that a following **d**, **o**, **u**, **x**, or X conversion specifier applies to a `long int` or **`unsigned`** `long int` argument. An l before any other conversion character is ignored
- a character that specifies the type of conversion to be applied:

d, b, o, u, x, X

The **`int`** argument is converted to signed decimal (**d**), binary (**b**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or X); the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** are used 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 will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f

The **`float`** or **`double`** argument is converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal point is equal to the precision specified. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character is displayed.

e, E

The **`float`** or **`double`** argument is converted in the style [-]d.ddde±dd, where there is one non-zero digit before the decimal point character and the number of digits after it is equal to the precision. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character is displayed. The E conversion specifier will produce a number with E instead of e introducing the exponent.

g, G

The **`float`** or **`double`** argument is converted in style **f** or **e** (or in style E in the case of the **G** conversion specifier), with the precision specifying the number of significant digits. The style depends on the value converted; style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result.

c

The **`int`** argument is converted to an unsigned char, and the resulting character is written.

s

The argument should be a pointer to a string. Characters from the string are written up to the terminating null character. If the precision is specified, no more than that many characters are written. If precision is not used, strings greater than 255 characters long can be printed.

%

A % is written. No argument is converted.

This function returns the number of characters transmitted, or a negative value if an output error occurred

```
fprintf(str1, "%s, %s, %d, %.2d:%.2d\n", weekday,
month, day, hour, min);
```

fputc

Type :function
Synopsis :`#include <h.stdio>`
`int fputc(int c, FILE *stream);`
Description :The **`fputc`** function writes the characters specified by **`c`** to the output stream pointed to by **`stream`**, at the position indicated by the associated file position indicator, and advances the indicator appropriately. If the file position indicator is not defined, the character is appended to the output stream. This function returns the character written. If a write error occurs, it returns **`EOF`**.

```
int c;
fputc(c, str1);
```

fputs

Type :function
Synopsis :`#include <h.stdio>`
`int fputs(char *s, FILE *stream);`
Description :The **`fputs`** function writes the string pointed to by **`s`** to the stream pointed to by **`stream`**. The terminating null character is not written. This function returns non-zero if an error occurs.

```
fputs(word, str2);
```

fread

Type :function
Synopsis :`#include <h.stdio>`
`int fread(char *ptr, size_t size, size_t num, FILE *stream);`
Description :The **`fread`** function reads, into the array pointed to by **`ptr`**, up to num members whose size is specified by **`size`**, from the stream pointed to by **`stream`**. The file position indicator (if defined) is advanced by the number of characters successfully read. This function returns the number of functions successfully read.

```
int nr;
nr = fread(array, 8, 100, str1);
```

freopen

Type :function
Synopsis :`#include <h.stdio>`
`FILE *freopen(char *filename,char *mode,FILE *stream);`
Description :The **`freopen`** function opens the file whose name is pointed to by **`filename`** and associates the stream pointed to by **`stream`** with it. The **`mode`** argument is used just as in the **`fopen`** function. The **`freopen`** function first attempts to close any file that may be associated with the specified stream. This function returns the value of the stream, or null if it fails.

fscanf

Type :function
Synopsis :`#include <h.stdio>`
`int fscanf(FILE *stream, char *format, ...)`
Description :The **`fscanf`** function reads input from the stream pointed to by **`stream`**, under control of the string pointed to by **`format`** that specifies the admissible input sequences and how they are converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behaviour is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored.

The format is composed of zero or more directives: one or more white-space characters; an ordinary character (not %); or a conversion specification. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

- an optional assignment-suppressing character *
- an optional decimal integer that specifies the maximum field length (1 to 255)
- an optional **l**. The conversion characters **d**, **o** and **x** may be preceded by **l** to indicate that a pointer to long rather than int appears in the argument list.
- a character that specifies the type of conversion to be applied. The valid conversion specifiers are:

d
Matches an optionally signed decimal integer; the corresponding argument shall be an integer pointer.

o
Matches an optionally signed octal integer; the corresponding argument shall be an integer pointer.

b
Matches a binary integer.

x
Matches an optionally signed hexadecimal integer; the corresponding argument shall be an integer pointer.

h

A short integer is expected in the input; the corresponding argument should be a pointer to a short integer. This overrides option l.

c

Matches a single character; the corresponding argument should be a character pointer. The normal skip over white-space characters is suppressed.

s

Matches a sequence of non-space characters. The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which is added automatically.

e, f, g

Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the **strtod** function. The corresponding argument shall be a pointer to a **float**.

Non-space literal characters in the format string must be matched by non-space characters in the input. White-space characters are ignored in the format string. Note also that %% will match %, and % followed by a character will match that character provided it is not a conversion character. The **fscanf** function returns **EOF** if an input failure occurs before any conversion. Otherwise, **fscanf** returns the number of input items assigned.

```
char input[50];
int n;
scanf("%d %s", &n, name);
```

fseek

Type :function

Synopsis :`#include <h.stdio>`
`int fseek(FILE *stream, long offset, int whence);`

Description :The **fseek** function sets the file position indicator for the stream pointed to by **stream**. The new position is at the signed number of characters specified by **offset** away from the point specified by **whence**. The specified point is the beginning of the file for **SEEK_SET**, the current position in the file for **SEEK_CUR**, or the end-of-file for **SEEK_END**. This function returns nonzero for an improper request.

fsetpos

Type :function

Synopsis :`#include <h.stdio>`
`int fsetpos(FILE *stream, fpos_t *pos);`

Description :The **fsetpos** function sets the file position indicator for the stream pointed to by **stream** according to the value of the object pointed to by **pos**, which should be a value returned by an earlier call to the **fgetpos** function on the same stream. If successful this function returns zero.

ftell

Type :function

Synopsis :`#include <h.stdio>`
`fpos_t ftell(FILE *stream);`

Description :The **ftell** function obtains the current value of the file position indicator for the stream pointed to by **stream**. The value is the number of characters from the beginning of the file. If successful this function returns the current value of the file position indicator.

fwrite

Type :function

Synopsis :`#include <h.stdio>`
`int fwrite (char *ptr, size_t size, size_t nmemb,`
`FILE stream);`

Description :The **fwrite** function writes, from the array pointed to by **ptr**, up to **nmemb** members whose size is specified by **size**, to the stream pointed to by **stream**. The file position indicator is advanced (if defined) by the number of characters successfully written. This function returns the number of members successfully written, which will be less than **nmemb** only if a write error is encountered.

getc

Type :macro

Synopsis :`#include <h.stdio>`
`#define getc(s) fgetc(s)`

Description :In Beebug C this macro is identical to the **fgetc** function described earlier.

getchar

Type :macro

Synopsis :`#include <h.stdio>`
`#define getchar() fgetc(stdin)`

Description :The **getchar** macro is equivalent to **getc** with the argument **stdin**. It returns the next character from the input stream pointed to by **stdin**. If the stream is at end-of-file, getchar returns **EOF**.

gets

Type :function

Synopsis :`#include <h.stdio>`
`char *gets(char *s);`

Description :The **`gets`** function reads characters from the input stream pointed to by **`stdin`**, into the array pointed to by **`s`**, until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array. This function returns **`s`** if successful. If an error occurs, a null pointer is returned.

printf

Type :function

Synopsis :`#include <h.stdio>`
`int printf(char *format, ...);`

Description :The **`printf`** function is equivalent to **`fprintf`** with the argument **`stdout`** interposed before the arguments to **`printf`**.

```
int value;
printf("The value is %d\n", value);
```

putc

Type :macro

Synopsis :`#include <h.stdio>`
`#define putc(c, s) fputc(c, s)`

Description :In Beebug C this macro is identical to the **`fputc`** function described earlier.

putchar

Type :macro

Synopsis :`#include <h.stdio>`
`#define putchar(c) fputc((c), stdout)`

Description :The **`putchar`** macro is equivalent to **`putc`** with the second argument **`stdout`**.

puts

Type :function

Synopsis :`#include <h.stdio>`
`int puts(char *s);`

Description :The **`puts`** function writes the string pointed to by **`s`** to the stream pointed to by **`stdout`**, and appends a new-line character to the output. The terminating null character is not written. This function returns nonzero if an error occurs.

remove

Type :function

Synopsis :`#include <h.stdio>`
`int remove(char *filename);`

Description :The **`remove`** function deletes the file whose name is pointed to by **`filename`**. It returns zero if the operation is successful.

```
remove(oldfile);
```

rename

Type :function

Synopsis :`#include <h.stdio>`
`int rename(char *old, char *new);`

Description :The **`rename`** function renames the file pointed to by **`old`** to the name pointed to by **`new`**. It returns zero if the operation is successful.

rewind

Type :function

Synopsis :`#include <h.stdio>`
`void rewind (FILE *stream);`

Description :The **`rewind`** function sets the file position indicator for the stream pointed to by **`stream`** to the beginning of the file. This function returns no value.

```
rewind(strl);
```

scanf

Type :function

Synopsis :`#include <h.stdio>`
`int scanf(char *format, ...);`

Description :The **`scanf`** function is equivalent to **`fscanf`** with the argument **`stdin`** interposed before the arguments to **`scanf`**.

SEEK_CUR

Type :macro

Synopsis :`#include <h.stdio>`
`#define SEEK_CUR 1`

Description :This macro expands to a constant suitable for use as the third argument to the **`fseek`** function.

SEEK_END

Type :macro
Synopsis :`#include <h.stdio>`
`#define SEEK_END 2`
Description :This macro expands to a constant suitable for use as the third argument to the **`fseek`** function.

SEEK_SET

Type :macro
Synopsis :`#include <h.stdio>`
`#define SEEK_SET 0`
Description :This macro expands to a constant suitable for use as the third argument to the **`fseek`** function.

sprintf

Type :function
Synopsis :`#include <h.stdio>`
`int sprintf(char *s, char *format, ...);`
Description :This function is equivalent to **`fprintf`**, except that the argument **`s`** specifies an array into which the generated output is to be written, rather than to a stream. A null character is written at the end, but is not counted as part of the sum returned. This function returns the number of characters written into the array.

sscanf

Type :function
Synopsis :`#include <h.stdio>`
`int sscanf(char *s, char *format, ...);`
Description :The **`sscanf`** function is equivalent to **`fscanf`**, except that the argument **`s`** specifies a string from which the input is to be obtained, rather from a stream. This function returns the number of input items assigned, or **`EOF`** if an input failure occurs.

ungetc

Type :function
Synopsis :`#include <h.stdio>`
`int ungetc(int c, FILE *stream);`
Description :The **`ungetc`** function pushes the character specified by **`c`** back onto the input stream pointed to by **`stream`**. The character will be returned by the next read on that stream. It returns the character pushed back after conversion, or **`EOF`** if the operation fails.

```
int c, f;
f = ungetc(c, str1);
```

h.stdlib - Standard operating system routines

This header defines the standard library functions and macros. These include a number of Acorn OS and Basic functions.

abort

Type :function
Synopsis :`#include <h.stdlib>`
`void abort(void);`
Description :The **`abort`** function causes abnormal program termination. This function does not return to its caller.

abs

Type :macro
Synopsis :`#include <h.stdlib>`
`#define abs(j) j < 0 ? -j : j`
Description :The **`abs`** macro computes the absolute value of the integer **`j`**.

adval

Type :macro
Synopsis :`#include <h.stdlib>`
`#define adval(c) (osbyte (128, c, 0) &0xffffU)`
Description :The **`adval`** macro reads the ADC channel or gets buffer status. It is identical to the Acorn OSBYTE call 128.

atexit

Type :function
Synopsis :`#include <h.stdlib>`
`int atexit(int (*func) (void));`
Description :The **`atexit`** function registers the function pointed to by **`func`**, to be called without arguments at normal program termination. This function returns zero if registration succeeds.

atof

Type :function
Synopsis :`#include <h.stdlib>`
`double atof(char *nptr);`
Description :The **`atof`** function converts the initial portion of the string pointed to by **`nptr`** to **`double`** representation. It returns the converted value.

atoi

Type :function
Synopsis :`#include <h.stdlib>`
`int atoi(char *nptr);`
Description :The **`atoi`** function converts the initial portion of the string pointed to by **`nptr`** to **`int`** representation. It returns the converted value.

atol

Type :function
Synopsis :`#include <h.stdlib>`
`long atol(char *str);`
Description :The **`atol`** function converts the initial portion of the string pointed to by **`nptr`** to **`long int`** representation. It returns the converted value.

calloc

Type :function
Synopsis :`#include <h.stdlib>`
`char *calloc(size_t nmemb, size_t size);`
Description :The **`calloc`** function allocates space for an array of **`nmem`** objects, each of whose size is **`size`**.The space is initialised to all bits zero. This function returns a pointer to the start of the allocated space. If the space cannot be allocated, this function returns a null pointer.

clg

Type :macro
Synopsis :`#include <h.stdlib>`
`#define clg() vdu(16)`
Description :The **`clg`** macro clears the current graphics window.

```
clg();
```

cls

Type :macro
Synopsis :`#include <h.stdlib>`
`#define cls() vdu(12)`
Description :The **`cls`** macro clears the current text window.

```
cls();
```

colour

Type :macro
Synopsis :`#include <h.stdlib>`
`#define colour(x) vdu(17, x)`
Description :The **`colour`** macro defines the text colour.

draw

Type :macro
Synopsis :`#include <h.stdlib>`
`#define draw(x, y) plot(5, x, y)`
Description :The **`draw`** macro draws a line to the x and y coordinates specified by the arguments **`x`** and **`y`**.

```
draw(200, 300);
```

envelope

Type :function
Synopsis :`#include <h.stdlib>`

```
void envelope(short, short, short, short, short,
short, short, short, short, short, short, short,
short, short);
```

Description :The **`envelope`** function requires the same arguments as the BBC Basic ENVELOPE function.

exit

Type :function
Synopsis :`#include <h.stdlib>`
`void exit(int status);`
Description :The **`exit`** function causes normal program termination to occur. It cannot return to its caller.

free

Type :function
Synopsis :`#include <h.stdlib>`
`void free(char *ptr);`
Description :The **`free`** function causes the space pointed to by **`ptr`** to be deallocated, i.e. made available for further allocation. This function returns no value.

gcol

Type :macro
Synopsis :`#include <h.stdlib>`
`#define gcol(a, b) vdu(18, a, b)`
Description :The **`gcol`** macro defines the graphics colour. Argument **`a`** is the plotting mode, and argument **`b`** is the logical colour.

```
gcol(0, 5);
```

himem

Type :macro

Synopsis :`#include <h.stdlib>`
`#define himem() (osbyte (132, 0, 0) &0xffffU)`

Description :The **himem** macro returns the bottom of display RAM address i.e. HIMEM.

```
long addr;
addr = himem();
```

inkey

Type :macro

Synopsis :`#include <h.stdlib>`
`#define inkey(k) (osbyte (129,k,(k)>>8) &0xffffU)`

Description :The **inkey** macro reads if a key is pressed within the time limit specified by the argument **k**.

labs

Type :macro

Synopsis :`#include <h.stdlib>`
`#define labs(j) abs(j)`

Description :In Beebug C this macro is identical to the function **abs**.

malloc

Type :function

Synopsis :`#include <h.stdlib>`
`char *malloc(size_t size);`

Description :The **malloc** function allocates space for an object whose size is specified by **size**. It returns a pointer to the start of the allocated space (lowest byte address). If space cannot be allocated, it returns a null pointer.

mode

Type :function

Synopsis :`#include <h.stdlib>`
`void mode(short num);`

Description :The **mode** function selects the screen mode specified by the argument **num**.

```
mode(3);
```

move

Type :macro

Synopsis :`#include <h.stdlib>`
`#define move(x, y) plot(4, x, y)`

Description :The **move** macro moves the graphics cursor, to coordinates **x**, **y**.

osbyte

Type :function

Synopsis :`#include <h.stdlib>`
`long osbyte(short A, short X, short Y);`

Description :The **osbyte** function calls the OSBYTE routine at 0xfff4 passing **A**, **X** and **Y** to the 6502 registers.

oscli

Type :function

Synopsis :`#include <h.stdlib>`
`void oscli(char *str);`

Description :The **oscli** function passes the string pointed to by **str** to the Acorn Command Line Interpreter. This function does not return a value.

```
oscli(*ADFS);
```

osword

Type :function

Synopsis :`#include <h.stdlib>`
`long osword(short A, char *para_block);`

Description :The **osword** function calls the Acorn OSWORD routine. The argument **A** is passed to the 6502 accumulator, and the argument **para_block** points to the parameter block containing the parameters.

page

Type :macro

Synopsis :`#include <h.stdlib>`
`#define page() (osbyte (131, 0, 0) &0xffffU)`

Description :The **page** macro returns the top of operating system RAM (OSHWM or PAGE).

plot

Type :function

Synopsis :`#include <h.stdlib>`
`void plot(short k, int x, int y);`

Description :The **plot** function calls the general graphics plotting routine. The plotting mode is determined by the value of the argument **k**. The arguments **x** and **y** specify the screen coordinates.

```
plot(69, 400, 400);
```

point

Type :function

Synopsis :`#include <h.stdlib>`
`int point(int x, int y);`

Description :The **`point`** function returns the colour of the screen pixel at the coordinates specified by the arguments **`x`** and **`y`**.

```
int p;
p = point(250,350);
```

rand

Type :function

Synopsis :`#include <h.stdlib>`
`int rand(void);`

Description :The **`rand`** function returns a pseudo-random integer in the range 0 to **`RAND_MAX`**.

```
int r;
r = rand();
```

RAND_MAX

Type :macro

Synopsis :`#include <h.stdlib>`
`#define RAND_MAX 32767`

Description :This macro expands to an integer constant which is the maximum value returned by the **`rand`** function.

realloc

Type :function

Synopsis :`#include <h.stdlib>`
`char *realloc(char *ptr, size_t size);`

Description :The **`realloc`** function changes the size of the object pointed to by **`ptr`** to the size specified by **`size`**. It returns a pointer to the start (lowest byte address) of the possibly moved object. If space cannot be allocated, this function returns a null pointer.

settime

Type :function

Synopsis :`#include <h.stdlib>`
`void settime(long t);`

Description :The **`settime`** function sets the internal timer to the value specified by the argument **`t`**.

sound

Type :function

Synopsis :`#include <h.stdlib>`
`void sound(int, int, int, int);`

Description :The **`sound`** function takes four parameters which are identical in operation to the those in the Basic command SOUND.

srand

Type :function

Synopsis :`#include <h.stdlib>`
`void srand(unsigned seed);`

Description :The **`srand`** function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **`rand`**.

strtod

Type :function

Synopsis :`#include <h.stdlib>`
`double strtod(char *uptr, char **endptr);`

Description :The **`strtod`** function converts the initial portion of the string pointed to by **`nptr`** to **`double`** representation. After successful conversion, a pointer to the final string is stored in the object pointed to by **`endptr`**. This function returns the converted value.

strtol

Type :function

Synopsis :`#include <h.stdlib>`
`long strtol(char *nptr, char **endptr, int base);`

Description :The **`strtol`** function converts the initial portion of the string pointed to by **`nptr`** to **`long int`** representation. If the value of **`base`** is zero, the expected form of the subject sequence is that of an integer constant. If the value of **`base`** is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer with the radix specified by **`base`**, optionally preceded by a plus or minus sign. The letters a (or A) through z (or Z) are ascribed the values 10 to 35. If the value of **`base`** is 16, the letters 0x or 0X may optionally precede the sequence of letters and digits.

After successful conversion, a pointer to the final string is stored in the object pointed to by **`endptr`**. This function returns the converted value.

system

Type :macro

Synopsis :`#include <h.stdlib>`
`#define system(s) oscli(s)`

Description :This function is identical to the **`oscli`** function.

tab

Type :macro
Synopsis :`#include <h.stdlib>`
`#define tab(x, y) vdu(31, x, y)`
Description :This macro moves the text cursor to the screen coordinates specified by the arguments **x** and **y**.

time

Type :function
Synopsis :`#include <h.stdlib>`
`long time(void);`
Description :The **time** function returns the value of the internal timer.

```
long t;
t = time();
```

vdu

Type :function
Synopsis :`#include <h.stdlib>`
`void vdu(`*`variable argument list`*`);`
Description :The **vdu** function sends the list of arguments specified to the VDU driver.

```
vdu(31, 10, 6);
```

h.string - Standard string and memory routines

The header h.string declares several functions useful for manipulating memory and character arrays. If an array is written beyond the end of an object, the behaviour is undefined. Please note that strings in these functions are limited to 255 characters in length.

memchr

Type :function
Synopsis :`#include <h.string>`
`char *memchr(char *s, int c, size_t n);`
Description :The **memchr** function locates the first occurrence of **c** in the initial **n** characters of the object pointed to by **s**. It returns a pointer to the located character, or a null pointer if the character does not occur in the object.

memcmp

Type :function
Synopsis :`#include <h.string>`
`int memcmp(char *s1, char *s2, size_t n);`
Description :The **memcmp** function compares the first **n** characters of the object pointed to by **s2** to the object pointed to by **s1**. It returns an integer greater than, equal to, or less than zero, depending on whether the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

memcpy

Type :function
Synopsis :`#include <h.string>`
`char *memcpy(char *s1, char *s2, size_t n);`
Description :The **memcpy** function copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. If overlapping objects are copied, the behaviour is undefined. This function returns the value of **s1**.

memmove

Type :function
Synopsis :`#include <h.string>`
`char memmove(char *s1, char *s2, size_t n);`
Description :The **memmove** function copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. This function operates correctly for objects that overlap. It returns the value of **s1**.

memset

Type :function
Synopsis :`#include <h.string>`
`char *memset(char *2, int c, size_t n);`
Description :This function writes the value of **c** into the first **n** characters of the object pointed to by **s**. It returns the value of **s**.

strcat

Type :function
Synopsis :`#include <h.string>`
`char *strcat(char *s1, char *s2);`
Description :The **strcat** function appends a copy of the string pointed to by **s2**, to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. This function returns the value of **s1**.

```
char s1[] = "abc";
char s2[] = "def";
printf("%s\n", strcat(s1, s2));
```

strchr

Type :function
Synopsis :`#include <h.string>`
`char *strchr(char *s, int c);`
Description :This function finds the first occurrence of **c** in the string pointed to by **s**. The terminating null character is considered part of the string. It returns a pointer to the located character, or a null pointer if the character does not occur in the string.

```
char s[] = "abcdefg";
int pos;
if ((pos = strchr(s, 'd')) != NULL)
printf("Found at: %d\n", pos - s);
```

strcmp

Type :function
Synopsis :`#include <h.string>`
`int strcmp(char *s1, char *s2);`
Description :The function **strcmp** compares the string pointed to by **s1** to the string pointed to by **s2**. It returns an integer greater than, equal to, or less than zero, depending on whether the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.

```
char s1[] = "abcdefg";
char s2[] = "abcdefgh";
```

```
printf("%s : %s = %d\n", s1, s1, strcmp(s1, s1));
printf("%s : %s = %d\n", s2, s1, strcmp(s2, s1));
printf("%s : %s = %d\n", s1, s2, strcmp(s1, s2));
```

strcpy

Type :function
Synopsis :`#include <h.string>`
`char *strcpy(char *s1, char *s2);`
Description :The **strcpy** function copies the string pointed to by **s2** into the array pointed to by **s1**. If copying takes place between objects that overlap, the behaviour is undefined. This function returns the value of **s1**.

```
char s1[] = "abcdefg";
char s2[] = "1234567";
strcpy(s1, s2);
printf("%s\n", s1);
```

strcspn

Type :function
Synopsis :`#include <h.string>`
`int strcspn(char *s1, char *s2);`
Description :The **strcspn** function calculates the length of the initial segment of the string pointed to by **s1** which consists entirely of characters not from the string pointed to by **s2**. The terminating null character is not considered part of **s2**. This function returns the length of the segment.

```
char s1[] = "abcdefg";
char s2[] = "pqvejnl";
printf("%d\n", strcspn(s1, s2));
```

strlen

Type :function
Synopsis :`#include <h.string>`
`int strlen(char *s);`
Description :The **strlen** function calculates the length of the string pointed to by **s**. It returns the number of characters that precede the terminating null character.

```
char s[] = "abcdefg";
printf("%d\n", strlen(s));
```

strncat

Type :function

Synopsis :#include <h.string>
char *strncat(char *s1, char *s2, size_t n);

Description :The **strncat** function appends not more than **n** characters of the string pointed to by **s2** (not including the terminating null character) to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**, and a terminating null character is always appended to the result. This function returns the value of **s1**.

```
char s1[] = "abcdefg";
char s2[] = "1234567";
strncat(s1, s2, 4);
printf("%s\n", s1);
```

strncmp

Type :function

Synopsis :#include <h.string>
int strncmp(char *s1, char *s2, size_t n);

Description :The **strncmp** function compares not more than **n** characters from the string pointed to by **s1** to the string pointed to by **s2**. It returns an integer greater than, equal to, or less than zero, depending on whether the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.

```
char s1[] = "abcdefg";
char s2[] = "abcdegh";
printf("%d\n", strncmp(s1, s2, 5));
printf("%d\n", strncmp(s1, s2, 6));
```

strncpy

Type :function

Synopsis :#include <h.string>
char *strncpy(char *s1, char *s2, size_t n);

Description :The **strncpy** function copies not more than **n** characters from the string pointed to by **s2** to the array pointed to by **s1**. If copying takes place between objects that overlap, the behaviour is undefined. If the string pointed to by **s2** is shorter than **n** characters, null characters are appended to the copy in the array pointed to by **s1**, until **n** characters in all have been written. This function returns the value of **s1**.

```
char s1[] = "abcdefg";
char s2[] = "1234567890";
strncpy(s1, s2, 8);
printf("%s\n", s1);
```

strpbrk

Type :function

Synopsis :#include <h.string>
char *strpbrk(char *s1, char *s2);

Description :The **strpbrk** function locates the first occurrence in the string pointed to by **s1** of any character from the string pointed to by **s2**. It returns a pointer to the character, or a null pointer if no character from **s2** occurs in **s1**.

```
char s1[] = "abcdefg";
char s2[] = "pqrfxyz";
printf("%d\n", strpbrk(s1, s2) - s1);
```

strrchr

Type :function

Synopsis :#include <h.string>
char *strrchr(char *s, int c);

Description :The **strrchr** function locates the last occurrence of **c** in the string pointed to by **s**. The terminating null character is considered to be part of the string. It returns a pointer to the character, or a null pointer if **c** does not occur in the string.

```
char s[] = "abcdabcd";
printf("%d\n", strrchr(s, 'c') - s);
```

strspn

Type :function

Synopsis :#include <h.string>
int strspn(char *s1, char *s2);

Description :The **strspn** function calculates the length of the initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**. It returns the length of the segment.

```
char s1[] = "abcdefg";
char s2[] = "xyaxybx";
printf("%d\n", strspn(s1, s2));
```

strstr

Type :function

Synopsis :#include <h.string>
char *strstr(char *s1, char *s2);

Description :The **strstr** function locates the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2**. It returns a pointer to the located string, or a null pointer if the string is not found.

```
char s1[] = "abcdefg";
char s2[] = "cde";
printf("%d\n", strstr(s1, s2) - s1);
```

strtok

Type :function
Synopsis :`#include <h.string>`
`char *strtok(char *s1, char s2);`

Description :The **`strtok`** function breaks the string pointed to by **`s1`** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **`s2`**. The first call in the sequence has **`s1`** as its first argument, and is followed by calls with a null pointer as their first argument. The separator string **`s2`** may be different from call to call.

The first call in the sequence searches **`s1`** for the first character that is not contained in the current separator string **`s2`**. If no such character is found, then there are no tokens in **`s1`** and **`strtok`** returns a null pointer. If such a character is found, it is the start of the first token.

The **`strtok`** function then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **`s1`**, and subsequent searches for a token will fail. If such a character is found, it is overwritten by a null character, which terminates the current token. The **`strtok`** function saves a pointer to the following character, from which the next search for a token will start. Each subsequent call with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above. This function returns a pointer to the first character of a token, or a null pointer if there is no token.

```
char s1[] = "The quick, brown fox.";
char s2[] = " .,";
char *arg, *word;
for (arg = s1; word = strtok(arg, s2)) != NULL;
arg = NULL)
printf("%s\n", word);
```

h.ctype - Character handling routines

This header declares a number of functions for testing characters. In each case the argument is an int, the value of which should be representable as an unsigned char or shall equal the value of the macro **`EOF`**. If the argument has any other value the behaviour is undefined.

isalnum

Type :function
Synopsis :`#include <h.ctype>`
`int isalnum(int c);`

Description :The **`isalnum`** function tests for any character for which **`isalpha`** or isdigit is true. It returns nonzero (**`TRUE`**) if **`c`** is a letter or digit.

isalpha

Type :function
Synopsis :`#include <h.ctype>`
`int isalpha(int c);`

Description :The **`isalpha`** function tests for any character for which **`isupper`** or **`islower`** is true. It returns nonzero (**`TRUE`**) if **`c`** is a letter.

isascii

Type :macro
Synopsis :`#include <h.ctype>`
`int isascii(int c);`

Description :The **`isascii`** macro tests for any character in the normal 7-bit ASCII range. It returns nonzero (**`TRUE`**) if **`c`** <= 127. This is not a standard ISO function.

iscntrl

Type :function
Synopsis :`#include <h.ctype>`
`int iscntrl(int c);`

Description :The **`iscntrl`** function returns nonzero (**`TRUE`**) if **`c`** is an ASCII control character.

isdigit

Type :function
Synopsis :`#include <h.ctype>`
`int isdigit(int c);`

Description :The **`isdigit`** function tests for any decimal digit character. It returns nonzero (**`TRUE`**) if **`c`** is a digit between 0 and 9 inclusive.

isgraph

Type :function

Synopsis :`#include <h.ctype>`
`int isgraph(int c);`

Description :The **`isgraph`** function tests for any printing character except space. It returns nonzero (**`TRUE`**) if **`c`** is not a space, control or delete character.

islower

Type :function

Synopsis :`#include <h.ctype>`
`int islower(int c);`

Description :The **`islower`** function tests for any lower-case letter. It returns nonzero (**`TRUE`**) if **`c`** is a lower-case letter.

isprint

Type :function

Synopsis :`#include <h.ctype>`
`int isprint(int c);`

Description :The **`isprint`** function tests for any printing character including space. It returns nonzero (**`TRUE`**) if **`c`** has an ASCII code between 32 and 126 inclusive.

ispunct

Type :function

Synopsis :`#include <h.ctype>`
`int ispunct(int c);`

Description :The **`ispunct`** function tests for any printing character except space or a character for which **`isalnum`** is true. It returns nonzero (**`TRUE`**) if **`c`** is a punctuation character (neither a space nor alphanumeric).

isspace

Type :function

Synopsis :`#include <h.ctype>`
`int isspace(int c);`

Description :The **`isspace`** function tests for the following white-space characters: space, formfeed (**`\f`**), new line (**`\n`**), carriage return (**`\r`**), horizontal tab (**`\t`**), or vertical tab (**`\v`**).

isupper

Type :function

Synopsis :`#include <h.ctype>`
`int isupper(int c);`

Description :The **`isupper`** function tests for any upper-case letter. It returns nonzero (**`TRUE`**) if **`c`** is an upper-case character.

isxdigit

Type :function

Synopsis :`#include <h.ctype>`
`int isxdigit(int c);`

Description :The **`isxdigit`** function tests for any hexadecimal digit. It returns nonzero (**`TRUE`**) if **`c`** is in the range 0 to 9, A to F (or a to f).

tolower

Type :function

Synopsis :`#include <h.ctype>`
`int tolower(int c);`

Description :If the argument is an upper-case letter, **`tolower`** returns the corresponding lower-case letter; otherwise the argument is returned unchanged.

toupper

Type :function

Synopsis :`#include <h.ctype>`
`int toupper(int c);`

Description :If the argument is a lower-case letter, **`toupper`** returns the corresponding upper-case letter; otherwise the argument is returned unchanged.

h.assert - Diagnostics

This header defines routines to put diagnostics into programs.

assert

Type :macro

Synopsis :`#include <h.assert>`

```
define assert(expression) _assert(expression)
```

Description :If the expression is **FALSE**, the following message is output:

```
Assertion failed at line LL
in function XXXX, in file YYYY
```

YYYY is the name of the source file, XXXX is the function, and LL is the source line number of the assert statement. If the function is **TRUE**, the **assert** macro returns no value.

This feature may be disabled by declaring the macro **NDEBUG** before **#include <h.assert>.**

_assert

Type :function

Synopsis :`#include <h.assert>`

```
void _assert(int);
```

Description :The function called by the macro **assert**.

NDEBUG

Type :macro

Synopsis :`#include h.assert`

Description :The macro **NDEBUG** allows the assert feature to be removed. It is not defined by <h.assert>. If **NDEBUG** is defined as a macro name at the point in the source file where <h.assert> is included, the **assert** macro is not defined. Alternatively, **NDEBUG** may be defined at compile time:

```
COMPILE/DEFINE=NDEBUG filename
```

h.setjmp - Non-local jumps

This header declares two functions and one type, for bypassing the normal function call and return discipline.

jump_buf

Type :type

Synopsis :`#include <h.setjmp>`

```
typedef char jmp_buf[6];
```

Description :The **jmp_buf** function is an array type suitable for holding the information needed to restore a calling environment.

longjmp

Type :function

Synopsis :`#include <h.setjmp>`

```
void longjmp(jmp_buf env, int val);
```

Description :The **longjmp** function restores the environment saved by the most recent call to **setjmp** in the same invocation of the program, with the corresponding **jmp_buf** argument. After **longjmp** is completed, program execution continues as if the corresponding call to **setjmp** had just returned the value specified by **val**.

setjmp

Type :function

Synopsis :`#include <h.setjmp>`

```
int setjmp(jmpbuf env);
```

Description :The **setjmp** function saves its calling environment in the **jmp_buf** argument for later use by the **longjmp** function. If the return is a from a direct invocation, **setjmp** returns zero (**FALSE**). If the return is from a call to the **longjmp** function, **setjmp** returns nonzero (**TRUE**).

h.stdarg - Variable arguments

This header declares a type and a function and defines two macros, for advancing through a list of arguments whose number and types are not known to the called function.

va_arg

Type :macro
Synopsis :`#include <h.stdarg>`
`#define va_arg(va_list ap, type)`
`(mode)*((mode*)((list+=4)-4))`

Description :The **va_arg** macro expands to an expression that has the type and value of the next argument in the call. The parameter **ap** is the same as the **va_list ap** initialised by **va_start**. The parameter type is a type name such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to type. The first invocation returns the value of the argument after that specified by **pn**. Successive invocations return the values of the remaining arguments.

va_count

Type :function
Synopsis :`#include <h.stdarg>`
`unsigned short va_count(void);`

Description :The **va_count** function returns the number of arguments passed to the function.

va_end

Type :macro
Synopsis :`#include <h.stdarg>`
`#define va_end(list);`

Description :The **va_end** macro facilitates a normal return from the function whose variable argument list was referenced by the expansion of **va_start** that initialised the **va_list ap**.

va_list

Type :type
Synopsis :`#include <h.stdarg>`
`typedef char *va_list;`

Description :**va_list** is an array type suitable for holding information needed by the macro **va_arg** and the function **va_end**.

va_start

Type :macro
Synopsis :`#include <h.stdarg>`
`#define va_start(list, pn) list = &pn`

Description :The **va_start** macro should be executed before any access to the unnamed arguments. The parameter **ap** points to an object that has type **list**. The parameter **pn** is the identifier of the rightmost parameter in the variable list in the function definition.

h.stddef - Standard definitions

This declares some types and macros that are defined in several headers.

bool

Type :type

Synopsis :
```
#include <h.stddef>
typedef int bool;
```

Description :`bool` is an integer boolean type.

byte

Type :type

Synopsis :
```
#include <h.stddef>
typedef unsigned short int byte;
```

Description :`byte` is an `unsigned short int` type representing a byte of memory.

FALSE

Type :macro

Synopsis :
```
#include <h.stddef> also <h.stdio>
#redef FALSE 0
```

Description :`FALSE` is a macro that expands to the constant 0.

NULL

Type :macro

Synopsis :
```
#include <h.stddef> also <h.stdio>
#redef NULL 0
```

Description :`NULL` is a macro that expands to the constant 0.

offsetof

Type :macro

Synopsis :
```
#include <h.stddef>
offsetof(type,identifier);
```

Description :`offsetof` expands to an integral constant expression that has the type `size_t`, the value of which is the offset in bytes, from the beginning of a structure designated by type, of the member designated by `identifier`.

ptrdiff_t

Type :type

Synopsis :
```
#include <h.stddef>
typedef int ptrdiff_t;
```

Description :`ptrdiff_t` is the signed integral type of the result of subtracting two pointers.

size_t

Type :type

Synopsis :
```
#include <h.stddef> also <h.stdio>, <h.stdlib>,
<h.string>
typedef int size_t;
```

Description :`size_t` is the unsigned integral type of the result of the `sizeof` operator.

TRUE

Type :macro

Synopsis :
```
#include <h.stddef> also <h.stdio>
#redef TRUE 1
```

Description :`TRUE` is a macro that expands to the constant 1.

h.call - Machine code call routines

This header defines a function and a type used for calling machine code routines. It also defines a number of macros that expand to the addresses of many standard operating system routines.

call

Type :function
Synopsis :`#include <h.call>`
`long call(char *addr, call_buf *regs,...);`
Description :The **`call`** function calls a machine code sub-routine. It takes as a first parameter **`addr`**, the address of the routine to call. The optional second parameter is the address of the **`call_buf`** structure containing the 6502 register values to pass. The rest of the parameters are arguments to be passed. This function returns a long integer (4 bytes) containing the contents of the registers at the execution of the RTS. The order of the registers is: APYX (hi to lo).

The header h.call defines the type **`call_buf`** which is used to set up the 6502 registers for a call. The type is a structure containing 4 members. The members are all 8 bits, and are the A, P, Y and X register values (where P is the Processor Status byte).

To set up the registers, first declare a variable of type 'call_buf'.

```
call_buf registers;
```

Then, assign to each of the members in the structure, the value to be supplied to the appropriate register:

```
registers.A = 236;
registers.X = 2;          /*   00000010 in binary */
registers.Y = 0xfd;       /*   11111101 in binary */
```

If a value is not assigned to any of the members, the appropriate register will not be set up on entry to the machine code routine (eg P in the above example). Then call the machine code using the **`call`** function:

```
call (OSBYTE, &registers);
```

supplying the address to call (in this case OSBYTE), and the address of the structure containing the registers. The above example switches off the VDU drivers and is equivalent to a call to the **`osbyte`** function with A=236, X=2 and Y=&FD.

The called machine code routine can also be passed arguments from C. The arguments to be passed are included after the address of the register structure in the function call:

```
call (address,&regs,arg1,arg2,arg3);
```

The address of the first argument is held in the zero page locations 86 and 87 (hex). These are the upper 2 bytes of the A internal C accumulator. Therefore to access the first argument, which was, say, a character, the following code may be used:

```
LDY #0
LDA (&86),Y
```

and A will contain the character passed. Each argument occupies 4 contiguous bytes of storage. Multi-byte numbers are stored low byte first. Therefore, the second argument would be accessible at offset 4 from the start of the arguments, the third at offset 8, etc.

The call function returns the values of the registers as its value. The registers are stored in the order A,P,Y,X from high byte to low. Therefore, the registers are accessible from a call as follows:

```
value = call (addr,&regs);
A = value >> 24;
P = (value >> 16) & 0xff;
Y = (value >> 8) & 0xff;
X = (value & 0xff);
```

Sometimes it is useful to obtain a 2 byte value returned in the X and Y registers from a call. For example, OSBYTE 130 (read machine high order address) returns the address required in X and Y. Y contains the high byte, and X the low byte. This value may be obtained by:

```
hi_address = value & 0xffff ;
```

This is the main reason for the order of the registers in the value returned by the 'call' function. A few of the macros (eg **`page`**) are defined in this way in h.stdlib.

The Beebug C system guarantees that the zero page addresses 00 to 2F (hex) are free for user programs. The following addresses are those used by Beebug C for internal use. The names used correspond to those in Appendix F of this user guide.

register	address (hex)	size (bits)
A	84	64
B	80	32
ASP	75	8
FSP	79	8
VSP	76	16
PC	7A	16
heap pointer	6E	16

Stack	Address (hex)	size (bytes)
AS	400	256
FS	500	256

The memory from the 'heap pointer' to VSP is free. However, during the execution of a program, these addresses may change.

call_buf

Type :type

Synopsis :

```
#include <h.call>
typedef struct
{
unsigned short A;
unsigned short P;
unsigned short Y;
unsigned short X;
} call_buf
```

Description :`call_buf` is a structure that holds the registers to pass to the `call` function.

GSINIT

Type :macro

Synopsis :

```
#include <h.call>
#define GSINIT 0xffc2
```

Description :The `GSINIT` macro expands to the call address of the 'general string input initialise' routine.

GSREAD

Type :macro

Synopsis :

```
#include <h.call>
#define GSREAD 0xffc5
```

Description :The `GSREAD` macro expands to the call address of the 'read character from string input' routine.

NVRDCH

Type :macro

Synopsis :

```
#include <h.call>
#define NVRDCH 0xffcb
```

Description :The `NVRDCH` macro expands to the call address of the non-vectored `OSRDCH` routine.

NVWRCH

Type :macro

Synopsis :

```
#include <h.call>
#define NVWRCH 0xffc8
```

Description :The `NVWRCH` macro expands to the call address of the non-vectored `OSWRCH` routine.

OSARGS

Type :macro

Synopsis :

```
#include <h.call>
#define OSARGS 0xffda
```

Description :The `OSARGS` macro expands to the call address of the 'adjust file arguments' routine.

OSASCI

Type :macro

Synopsis :

```
#include <h.call>
#define OSASCI 0xffe3
```

Description :The `OSASCI` macro expands to the call address of the 'write character to output stream' routine.

OSBGET

Type :macro

Synopsis :

```
#include <h.call>
#define OSBGET 0xffd7
```

Description :The `OSBGET` macro expands to the call address of the 'get one byte from file' routine.

OSBPUT

Type :macro

Synopsis :

```
#include <h.call>
#define OSBPUT 0xffd4
```

Description :The `OSBPUT` macro expands to the call address of the 'write a single byte to file' routine.

OSBYTE

Type :macro

Synopsis :`#include <h.call>`
`#define OSBYTE 0xfff4`

Description :The **OSBYTE** macro expands to the call address of the 'operating system call' routine.

OSCLI

Type :macro

Synopsis :`#include <h.call>`
`#define OSCLI 0xfff7`

Description :The **OSCLI** macro expands to the call address of the 'pass line of text to CLI' routine.

OSEVEN

Type :macro

Synopsis :`#include <h.call>`
`#define OSEVEN 0xffbf`

Description :The **OSEVEN** macro expands to the call address of the 'generate an event' routine.

OSFILE

Type :macro

Synopsis :`#include <h.call>`
`#define OSFILE 0xffdd`

Description :The **ODFILE** macro expands to the call address of the 'read/write file' routine.

OSFIND

Type :macro

Synopsis :`#include <h.call>`
`#define OSFIND 0xffce`

Description :The **OSFIND** macro expands to the call address of the 'open/close file' routine.

OSGBPB

Type :macro

Synopsis :`#include <h.call>`
`#define OSGBPB 0xffd1`

Description :The **OSGBPB** macro expands to the call address of the 'read/write group of bytes' routine.

OSNEWL

Type :macro

Synopsis :`#include <h.call>`
`#define OSNEWL 0xffe7`

Description :The **OSNEWL** macro expands to the call address of the 'write newline to output stream' routine.

OSRDCH

Type :macro

Synopsis :`#include <h.call>`
`#define OSRDCH 0xffe0`

Description :The **OSRDCH** macro expands to the call address of the 'read character from input stream' routine.

OSRDRM

Type :macro

Synopsis :`#include <h.call>`
`#define OSRDRM 0xffb9`

Description :The **OSRDRM** macro expands to the call address of the 'read byte from paged ROM' routine.

OSRDSC

Type :macro

Synopsis :`#include <h.call>`
`#define OSRDSC 0xffb9`

Description :The **OSRDSC** macro expands to the call address of the 'read byte from screen memory' routine.

OSWORD

Type :macro

Synopsis :`#include <h.call>`
`#define OSWORD 0xfff1`

Description :The **OSWORD** macro expands to the call address of the 'operating system call' routine.

OSWRCH

Type :macro

Synopsis :`#include <h.call>`
`#define OSWRCH 0xffee`

Description :The **OSWRCH** macro expands to the call address of the 'write character to output stream' routine.

OSWRSC

Type :macro
Synopsis :`#include <h.call>`
`#define OSWRSC 0xffb3`
Description :The **OSWRSC** macro expands to the call address of the 'write character to screen memory' routine.

h.math - Mathematics

This header defines two mathematical constants.

HUGE_VAL

Type :macro
Synopsis :`#include <h.math>`
`#define HUGE_VAL 1.7014118e38`
Description :Expands to a positive double expression. This is the maximum possible number in Beebug C.

PI

Type :macro
Synopsis :`#include <h.math>`
`#define PI 3.141593`
Description :Expands to the value of PI.

h.limits - Integer arithmetic limits

The header h.limits defines a number of integer arithmetic limits.

CHAR_BIT

Type :macro
Synopsis :`#include <h.limits>`
`#define CHAR_BIT 8`
Description :The maximum number of bits for the smallest object that is not a bit-field (byte).

CHAR_MAX

Type :macro
Synopsis :`#include <h.limits>`
`#define CHAR_MAX 127`
Description :The maximum value for an object of type **`char`**.

CHAR_MIN

Type :macro
Synopsis :`#include <h.limits>`
`#define CHAR_MIN -127`
Description :The minimum value for an object of type **`char`**.

INT_MAX

Type :macro
Synopsis :`#include <h.limits>`
`#define INT_MAX 32767`
Description :The maximum value for an object of type **`int`**.

INT_MIN

Type :macro
Synopsis :`#include <h.limits>`
`#define INT_MIN -32767`
Description :The minimum value for an object of type **`int`**.

LONG_MAX

Type :macro
Synopsis :`#include <h.limits>`
`#define LONG_MAX 2147483647`
Description :The maximum value for an object of type **`long int`**.

LONG_MIN
Type :macro
Synopsis :`#include <h.limits>`
`#define LONG_MIN -2147483647`
Description :The minimum value for an object of type **`long int`**.

SCHAR_MAX
Type :macro
Synopsis :`#include <h.limits>`
`#define SCHAR_MAX 127`
Description :The maximum value for an object of type **`signed char`**.

SCHAR_MIN
Type :macro
Synopsis :`#include <h.limits>`
`#define SCHAR_MIN -127`
Description :The minimum value for an object of type **`signed char`**.

SHRT_MAX
Type :macro
Synopsis :`#include <h.limits>`
`#define SHRT_MAX 127`
Description :The maximum value for an object of type **`short int`**.

SHRT_MIN
Type :macro
Synopsis :`#include <h.limits>`
`#define SHRT_MIN -127`
Description :The minimum value for an object of type **`short int`**.

UCHAR_MAX
Type :macro
Synopsis :`#include <h.limits>`
`#define UCHAR_MAX 255U`
Description :The maximum value for an object of type **`unsigned char`**.

UINT_MAX
Type :macro
Synopsis :`#include <h.limits>`
`#define UINT_MAX 65535U`
Description :The maximum value for an object of type **`unsigned int`**.

ULONG_MAX
Type :macro
Synopsis :`#include <h.limits>`
`#define ULONG_MAX 4294967295U`
Description :The maximum value for an object of type **`unsigned long int`**.

USHRT_MAX
Type :macro
Synopsis :`#include <h.limits>`
`#define USHRT_MAX 255U`
Description :The maximum value for an object of type **`unsigned short int`**.

h.float - Floating point arithmetic limits

The header h.limits defines a number of floating-point arithmetic limits.

DBL_DIG
Type :macro
Synopsis :`#include <h.float>`
`#define DBL_DIG 7`
Description :The number of decimal digits of precision.

DBL_EPSILON
Type :macro
Synopsis :`#include <h.float>`
`#define DBL_EPSILON 1.192093e-7`
Description :The minimum positive floating-point number.

DBL_MANT_DIG
Type :macro
Synopsis :`#include <h.float>`
`#define DBL_MANT_DIG 24`
Description :The number of base **`FLT_RADIX`** digits in the floating-point mantissa.

DBL_MAX
Type :macro
Synopsis :`#include <h.float>`
`#define DBL_MAX 1.701412e38`
Description :The maximum representable finite floating-point number.

DBL_MAX_10_EXP

Type :macro
Synopsis :`#include <h.float>`
`#define DBL_MAX_10_EXP 38`
Description :The maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers.

DBL_MIN

Type :macro
Synopsis :`#include <h.float>`
`#define DBL_MIN 1.469368e-39`
Description :The minimum normalised positive floating-point number.

DBL_MIN_10_EXP

Type :macro
Synopsis :`#include <h.float>`
`#define DBL_MIN_10_EXP -39`
Description :The minimum negative integer such that 10 raised to that power is in the range of normalised floating-point numbers.

FLT_DIG

Type :macro
Synopsis :`#include <h.float>`
`#define FLT_DIG 7`
Description :The number of decimal digits of precision.

FLT_EPSILON

Type :macro
Synopsis :`#include <h.float>`
`#define FLT_EPSILON 1.192093e-7`
Description :The minimum positive floating-point number.

FLT_MANT_DIG

Type :macro
Synopsis :`#include <h.float>`
`#define FLT_MANT_DIG 24`
Description :The number of base **`FLT_RADIX`** digits in the floating-point mantissa.

FLT_MAX

Type :macro
Synopsis :`#include <h.float>`
`#define FLT_MAX 1.701412e38`
Description :The maximum representable finite floating-point number.

FLT_MAX_10_EXP

Type :macro
Synopsis :`#include <h.float>`
`#define FLT_MAX_10_EXP 38`
Description :The maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers.

FLT_MIN

Type :macro
Synopsis :`#include <h.float>`
`#define FLT_MIN 1.469368e-39`
Description :The minimum normalised positive floating-point number.

FLT_MIN_10_EXP

Type :macro
Synopsis :`#include <h.float>`
`#define FLT_MIN_10_EXP -39`
Description :The minimum negative integer such that 10 raised to that power is in the range of normalised floating-point numbers.

FLT_RADIX

Type :macro
Synopsis :`#include <h.float>`
`#define FLT_RADIX 2`
Description :The radix of exponent representation

FLT_ROUNDS

Type :macro
Synopsis :`#include <h.float>`
`#define FLT_ROUNDS`
Description :The addition rounds.

7. Beebug C Library Facility

Introduction

The Beebug C linker searches libraries for any unresolved function calls in a program. The standard library *rtlib* is supplied with Beebug C, but other libraries may be created using the library facility described in this section. The library facility is a program written in Beebug C and is responsible for the upkeep of libraries. It is supplied on the library disc as both source files (see later) and an executable file. It may be run in the usual way by typing:

```
RUN library
```

The following prompt is then displayed:

```
LIBRARY>
```

When this prompt is visible, any of the following commands may be entered:

USE	establish a working library
LIST	list the contents of a library
INSERT	insert a Beebug C object module into the library
DELETE	delete a module from a library
EXTRACT	extract a module into an object file
COMMIT	commit all changes to a library
HELP	list all commands available
CREATE	create a new library
NEW	clear memory for a new library
QUIT	quit the facility, committing changes if required
EXIT	exit the facility, committing changes

If the command typed starts with an asterisk, the command is passed to the Operating System. This enables the usual 'star' commands to be used. When a module is deleted or inserted, its entry in the library is only marked as having been deleted or inserted as appropriate. No actual file operations are carried out until the **COMMIT** command is used, or the session is terminated.

Please note that in the descriptions that follow, a module refers to the object code of a function or number of functions.

Library facility commands

USE

This command establishes a working library. The facility will prompt for the name of the library to use:

```
LIBRARY> USE
Name of library to use:
```

Type in the name of the library to be worked upon. The system will check that the library is valid and if it isn't, an appropriate message will be displayed.

When the library has been validated, the facility will display a copyright message and the identification text for the library (if they have been defined). They serve to identify the library to the user, and may contain any text, such as version numbers, etc. The modules in the library are then read; the number of each one being displayed as it is identified.

LIST

This command allows the contents of a library to be displayed. The listing consists of two pieces of information:

1 - The module name
2 - The length of the module in bytes

If a module is currently deleted, or has just been inserted, the length information will not be displayed, but instead the word `Deleted` or `New` will appear in its place. The module name is the field by which the module is identified.

INSERT

This command allows a module to be inserted into the library. The facility will prompt for the name of the module to insert. If a module of the name supplied already exists in the library, an appropriate message is displayed.

Then the name of the object file of the module must be entered. This must be a valid object file.

The module is marked as `New` until a **COMMIT** is performed.

DELETE
This command allows a module to be deleted from the library. The actual delete operation is not carried out until the **COMMIT** command is used, or the session terminated.

EXTRACT
This allows a module to be extracted from the library into an object file. The object file created will be suitable for linking with a users program by the Linker.

COMMIT
This command commits all changes made to the library to disc. Once this has been done, the library is permanently changed, and any module which has been deleted cannot be recovered.

The command will ask if the identification text is to be changed. An answer of 'yes' will cause the facility to ask for the new identification text. If identification text is not changed, the old text will be used.

HELP
This will list all the library facility commands available.

CREATE
This allows a new library to be created on disc. It will then be used as the working library.

NEW
This will inform the facility that all changes have been made to the working library, and all memory should be freed. If any changes have been made to the library, a chance to commit these to the disc is offered. A new library may be **USE**d after a **NEW** command.

QUIT
This informs the facility that the current session is finished. If any changes have been made to the working library, a chance to commit these will be offered. The facility will then terminate.

EXIT
This informs the facility that the current session is finished. Any changes to the working library will be committed and the session terminated.

Creating Library modules

There are certain rules which must be obeyed when using this facility to create a new library, or add functions to an existing library. If these are not followed, the program will not work, even if it links correctly.

1. The library only accepts Beebug C object modules, not executable images.
2. A module may contain many functions, but must have one main function.
3. The main function must have the same name as the module being inserted.
4. The main function must be first in an object module. This means it must be defined before any subsidiary functions.
5. The main function must be declared as 'static'. This avoids multiple definition of a symbol in the linker.
6. It is preferable that the module is compiled with no debugging information.

The following example shows a valid source for an object module called 'hello_world' to be inserted into a library.

```
static void hello_world(void)
{
printf ("Hello world\n") ;
}
```

As will be seen later, the main function has the same name as the module to be inserted. It is first in the file, and is declared as 'static'. The file in which the module is saved does not have to have the same name as the module.

To insert the above module into a library called TESTLIB, the following commands must be executed:

```
COMPILE/NODEBUG HELLO
RUN LIBRARY

Beebug C Library Facility
Version 1.0
(C) 1987 Beebug Ltd

LIBRARY> USE
Name of library to use: TESTLIB

Reading...10
LIBRARY> INSERT
```

```
Name of module to insert: hello_world
Name of object file: O.HELLO
LIBRARY> EXIT
Change identification text ? no
Committing...11
```

The object module is now inserted into the library. Please note that the above example assumes that a library called TESTLIB has been created, and that it contains 10 modules.

Compiling the library facility

The library facility is supplied on the library disc in both source and executable form. Advanced users may like to alter the program source files to suit their own needs. The source files are library, dirops, and commit, and they may be re-compiled and linked as follows:

```
compile library
compile dirops
compile commit

link/nodebug/origin=&2500 library,dirops,commit
```

Please note that these programs are fairly complex, and modifications should only be made by experienced users.

Appendix A

Beebug C Command Summary

Listed below are all the commands available in Beebug C. Each command is followed by its minimum abbreviation, and a list of any optional qualifiers.

CLOSE **CL**
Close all open files

COMMAND ***mode_number*** **COMM**
Set error handling for un-recognised command

mode 0 - print normal error message (default setting)
mode 1 - attempt to ***RUN** a file (passes it to OSCLI)
mode 2 - attempt to **RUN** a C program (defaults to directory **E**)

COMPILE [*qualifiers*] *filename* **COMP**
Compile the C source text ***filename***

optional qualifiers	abbrev.	default setting
/[NO]OBJECT[=*filename*]	O	object code put in dir O
/[NO]LIST[=*filename*]	L	list to screen
/[NO]WARNINGS	W	warnings on
/[NO]PORTABLE	P	no portable
/[NO]DEBUG	DEB	debugging on
/DEFINE=*macro_name*[,...]	D	no macros
/LSPACE=*buffer_size*	LS	&300 (768 bytes)
/[NO]OPTIMISE	OP	optimise on

LINK [*qualifiers*] *filename*[,...] **L**
Link ***filename*** to other object files and to standard library

optional qualifiers	abbrev.	default setting
/[NO]EXECUTABLE[=*filename*]	E	executable code in dir E
/[NO]LIBRARY[=*filename*[,...]]	L	standard library *rtlib*
/[NO]DEBUG	D	debugging on
/ORIGIN=*start_address*	O	top of OS RAM (OSHWM)
/[NO]STANDALONE	S	no standalone

MODE ***mode_number*** **M**
Set screen mode

REPORT **RE**
Report last error encountered

RUN [*qualifiers*] *filename* [*arg1 arg2* ...] **RU**
Execute the C program ***filename***.

optional qualifiers	abbrev.	default setting
/[NO]TRACEBACK	**T**	no traceback
/INPUT=filename	**I**	input from keyboard
/[NO]OUTPUT[=*filename*]	**O**	output to VDU
/[NO]ERROR[=*filename*]	**E**	errors to VDU

Library Facility Command Summary

COMMIT	commit all changes to a library
CREATE	create a new library
DELETE	delete a module from a library
EXIT	exit the facility, committing changes
EXTRACT	extract a module into an object file
HELP	list all commands available]
INSERT	insert an object module into the library
LIST	list the contents of a library
NEW	clear memory for a new library
QUIT	quit the facility, committing changes if required

Appendix B

Beebug C Extensions

Beebug C is a full implementation of the Kernighan and Ritchie standard with the following extensions:

- function prototypes
- **#pragma** preprocessor line
- **#redef** preprocessor line
- data type **void**
- new data types
- initialisation of unions
- local structure and union members

If the compiler is switched to the portable mode (using the optional qualifier **PORTABLE**), the use of these extensions will cause a warning to be produced at compile time. This does not affect compilation.

1. Function prototypes

In standard C a function definition takes the following form:

```
int function(p1,p2,p3)
   char p1;
   short int *p2;
   int p3[];
```

This format is also allowed by Beebug C. However, the proposed ISO standard for C defines a new format for function definitions. Beebug C allows this new format to be used, and indeed it is recommended that it is used. The format is:

```
int function(char p1, short int *p2, int p3[]);
```

The use of this type of syntax allows the compiler to set up a function prototype for the function. The compiler can then check the number and type of the parameters against the arguments supplied in the prototype. In an external declaration of a function, the argument types may be specified, and a prototype built by supplying the argument types in the declaration:

```
extern int function(char, short int*, int []);
```

In this case the identifiers may also be included, but only for documentation purposes.

A function taking a variable number of arguments may be defined as:

```
extern int printf(char *format,...);
```

The three dots '...' are known as an *ellipsis*.

2. The `#pragma` preprocessor line

This allows control over the compiler options from within the source code. It is followed by a single letter which specifies whether the option is to be switched on or off. If the letter is in upper case the option is switched on, otherwise it is switched off.

The following options are implemented:

D or **d**	- debugging
O or **o**	- optimisation
W or **w**	- warnings
P or **p**	- portability
L or **l**	- listing

3. The `#redef` preprocessor line

The **`#define`** preprocessor line may not be used to redefine a macro. This extension allows any macro to be redefined. Its syntax is the same as that for **`#define`**.

4. Data type `void`

Beebug C allows the **`void`** data type. A function whose type is **`void`** may not return a value. An error will be generated if this is attempted.

It may also be used to specify that a function may not take any parameters:

```
extern long time(void);
```

5. New data types

The following new data types may be used:

unsigned char
unsigned short
unsigned long

6. Initialisation of unions

Unions may be initialised as structures, but only the first member will be initialised. Any attempt to initialise further members will result in the error `Too many initialisers.`

7. Local structure and union members

Structure and union members are local and can therefore be duplicated in different structures.

Appendix C

Beebug C Limitations

Beebug C is a full implementation of the Kernighan & Ritchie standard, but it does have a few limitations. Most of these are due to the method of implementation, and in normal use they will not be exceeded.

Data type limits

type	bits
char	8
short	8
int	16
long	32
float	32
double	32

Floating point arithmetic is to 7 significant figures.

Nested `#include` files

A maximum of 3 **`#include`** files may be nested (DFS limitation)

Array declarations in type declarations

Only one array declaration may appear in a type declaration. This would mean that the following is illegal:

```
int (*(*(*a)[])())[]
```

The above declares **a** as a pointer to an array of pointers to functions, returning a pointer to an array of ints! This is almost unintelligible and is bad programming practice.

This limitation does not mean that multi-dimensional arrays are illegal, it only means that the dimensions of an array may not be separated.

The above declaration will work if the type is split up into separate typedefs, as it should be.

Array dimensions

There is a maximum of 126 dimensions in any one array

Pointer levels

There is a maximum of 255 pointer to pointer levels. This means that there may be a maximum of 255 asterisks in the following:

```
int *****************ptr;
```

Block nesting

The 6502 stack limits the maximum level of nesting for blocks. This level has not been calculated.

Line lengths

Lines can be a maximum of 128 characters

Identifier lengths

Identifiers can be a maximum of 31 characters

Number of errors

There is a maximum of 255 errors in one compilation

Forward references to structures

A structure tag cannot be forward referenced without being defined. It must be declared as a character and casted when needed.

Appendix D

Summary of Library Functions

Listed below is an alphabetical list of all functions and macros available in Beebug C. Each function or macro is listed together with its file type, the header file which declares it, and a brief description. The page number on which a full description can be found is also given. The function/macro types are: *f*=function, *m*=macro, *t*=type definition and *e*=expression.

Function	Type	Header	Description	Page
abort	*f*	*h.stdlib*	cause abnormal program termination	49
abs	*m*	*h.stdlib*	return absolute value of an integer	49
adval	*m*	*h.stdlib*	read ADC channel or buffer status	49
assert	*m*	*h.assert*	terminate if assertion fails	66
_assert	*f*	*h.assert*	assert function (for internal use)	66
atexit	*f*	*h.stdlib*	call function at program termination	49
atof	*f*	*h.stdlib*	convert ASCII to float	49
atoi	*f*	*h.stdlib*	convert ASCII to int	50
atol	*f*	*h.stdlib*	convert ASCII to long	50
bget	*f*	*h.stdio*	get a byte from stream	36
bool	*t*	*h.stddef*	boolean type int	70
bput	*f*	*h.stdio*	put a byte to a stream	36
byte	*t*	*h.stddef*	byte type unsigned short int	70
call	*f*	*h.call*	call user machine code	72
call_buf	*t*	*h.call*	structure for holding 6502 registers	74
calloc	*f*	*h.stdlib*	allocate space for an array	50
CHAR_BIT	*m*	*h.limits*	constant 8	79
CHAR_MAX	*m*	*h.limits*	constant 127	79
CHAR_MIN	*m*	*h.limits*	constant -127	79
clearerr	*m*	*h.stdio*	clear error indicators	36
clg	*m*	*h.stdlib*	clear graphics screen	50
cls	*m*	*h.stdlib*	clear text screen	50
colour	*m*	*h.stdlib*	set text colour	50
DBL_DIG	*m*	*h.float*	decimal digits of precision	81
DBL_EPSILON	*m*	*h.float*	minimum floating-point number	81
DBL-MANT-DIG	*m*	*h.float*	number of digits in the mantissa	81
DBL_MAX	*m*	*h.float*	maximum finite floating point number	81
DBL_MAX_10_EXP	*m*	*h.float*	see full description	82
DBL_MIN	*m*	*h.float*	the minimum floating point number	82
DBL_MIN_10_EXP	*m*	*h.float*	see full description	82
draw	*m*	*h.stdlib*	draw a line	51
envelope	*f*	*h.stdlib*	set pitch and amplitude envelope	52
EOF	*m*	*h.stdio*	an integer indicating *end-of-file* (-1)	36
exit	*f*	*h.stdlib*	cause normal program termination	51
FALSE	*m*	*h.stddef*	constant 0 (also in *h.stdio*)	70
fclose	*f*	*h.stdio*	close a stream	37
feof	*f*	*h.stdio*	test for *end-of-file* for a stream	37
ferror	*m*	*h.stdlib*	test stream error condition (returns 0)	37
fflush	*f*	*h.stdio*	flush a stream's buffer	37
fgetc	*f*	*h.stdio*	get character from a stream	38
fgetpos	*f*	*h.stdio*	get file position in a stream	38
fgets	*f*	*h.stdio*	get string from a stream	38
FILE	*t*	*h.stdio*	type for specifying stream information	39
FLT_DIG	*m*	*h.float*	the number of decimal digits of precision	82
FLT_EPSILON	*m*	*h.float*	the minimum +ve floating point number	82
FLT_MANT_DIG	*m*	*h.float*	see full description	82
FLT_MAX	*m*	*h.float*	the maximum floating point number	82
FLT_MAX_10_EXP	*m*	*h.float*	see full description	83
FLT_MIN	*m*	*h.float*	the minimum +ve floating point number	83
FLT_MIN_10_EXP	*m*	*h.float*	see full description	83

Function	Type	Header	Description	Page
FLT_RADIX	*m*	*h.float*	the radix of exponent representation	83
FLT_ROUNDS	*m*	*h.float*	the addition rounds	83
fopen	*f*	*h.stdio*	open a stream	39
fpos_t	*m*	*h.stdio*	type for specifying all positions in file	39
fprintf	*f*	*h.stdio*	formatted print to a stream	39
fputc	*f*	*h.stdio*	put character to a stream	42
fputs	*f*	*h.stdio*	put string to a stream	42
fread	*f*	*h.stdio*	read item from a stream	42
free	*f*	*stdlib*	free previously allocated memory	51
freopen	*f*	*h.stdio*	close and re-open a stream	42
fscanf	*f*	*h.stdio*	formatted input from a stream	43
fseek	*f*	*h.stdio*	set file position on a stream	44
fsetpos	*f*	*h.stdio*	set file position on a stream	45
ftell	*f*	*h.stdio*	read file position pointer of stream	45
fwrite	*f*	*h.stdio*	write items to a stream	45
gcol	*m*	*h.stdlib*	set graphics colour	51
getc	*m*	*h.stdio*	get character from stream	45
getchar	*m*	*h.stdio*	get character from **stdin**	45
gets	*f*	*h.stdio*	get string from stream	46
GSINIT	*m*	*h.call*	address of GSINIT routine	74
GSREAD	*m*	*h.call*	address of GSREAD routine	74
himem	*m*	*h.stdlib*	read bottom of display RAM (HIMEM)	52
HUGE_VAL	*m*	*h.math*	max positive double (1.7014118E38)	78
inkey	*m*	*h.stdlib*	read key with time limit	52
INT_MAX	*m*	*h.limits*	constant 32767	79
INT_MIN	*m*	*h.limits*	constant -32767	79
isalnum	*f*	*h.ctype*	test if character is alphanumeric	63
isalpha	*f*	*h.ctype*	test if character is alphabetic	63
isascii	*m*	*h.ctype*	test if character is 7 bit Ascii	63

Function	Type	Header	Description	Page
iscntrl	*f*	*h.ctype*	test for control character	63
isdigit	*f*	*h.ctype*	test if character is a decimal digit	63
isgraph	*f*	*h.ctype*	test for printing character except space	64
islower	*f*	*h.ctype*	test for any lower case character	64
isprint	*f*	*h.ctype*	test for any printing character	64
ispunct	*f*	*h.ctype*	test for character not alpha or space	64
isspace	*f*	*h.ctype*	test for white space characters	64
isupper	*f*	*h.ctype*	test for any upper case character	64
isxdigit	*f*	*h.ctype*	test for hexadecimal digit	65
jmp_buf	*t*	*h.setjmp*	array type for non-local jump info	67
labs	*m*	*h.stdlib*	return absolute value as a long int	52
longjmp	*f*	*h.setjmp*	restore function environment	67
LONG_MAX	*m*	*h.limits*	constant 2147483647	79
LONG_MIN	*m*	*h.limits*	constant -2147483647	80
malloc	*f*	*h.stdlib*	allocate space for an object	52
memchr	*f*	*h.string*	search memory for character	57
memcmp	*f*	*h.string*	compare memory	57
memcpy	*f*	*h.string*	copy memory	57
memmove	*f*	*h.string*	copy memory with overlap corrected	57
memset	*f*	*h.string*	set block of memory to a character	58
mode	*f*	*h.stdlib*	set screen mode	52
move	*m*	*h.stdlib*	move graphics cursor	52
NDEBUG	*m*	*h.assert*	switch off assertions (not defined)	66
NULL	*m*	*h.stddef*	null constant 0 (also in *h.stdio*)	70
NVRDCH	*m*	*h.call*	address of non-vectored OSRDCH	75
NVWRCH	*m*	*h.call*	address of non-vectored OSWRCH	75
offsetof	*m*	*h.stddef*	offset from start of structure	70
OSARGS	*m*	*h.call*	address of OSARGS routine	75
OSASCI	*m*	*h.call*	address of OSASCI routine	75

Function	Type	Header	Description	Page
OSBGET	*m*	*h.call*	address of OSBGET routine	75
OSBPUT	*m*	*h.call*	address of OSBPUT routine	75
OSBYTE	*m*	*h.call*	address of OSBYTE routine	76
osbyte	*f*	*h.stdlib*	OSBYTE call	53
OSCLI	*m*	*h.call*	address of OSCLI routine	76
oscli	*f*	*h.stdlib*	pass string to Command Line Interp.	53
OSEVEN	*m*	*h.call*	address of OSEVEN routine	76
OSFILE	*m*	*h.call*	address of OSFILE routine	76
OSFIND	*m*	*h.call*	address of OSFIND routine	76
OSGBPB	*m*	*h.call*	address of OSGBPB routine	76
OSNEWL	*m*	*h.call*	address of OSNEWL routine	77
OSRDCH	*m*	*h.call*	address of ODRDCH routine	77
OSRDRM	*m*	*h.call*	address of OSRDRM routine	77
OSRDSC	*m*	*h.call*	address of OSRDSC routine	77
OSWORD	*m*	*h.call*	address of OSWORD routine	77
osword	*f*	*h.stdlib*	OSWORD call	53
OSWRCH	*m*	*h.call*	address of OSWRCH routine	77
OSWRSC	*m*	*h.call*	address of OSWRSC routine	78
page	*m*	*h.stdlib*	read top of OS RAM (OSWRCH)	53
PI	*m*	*h.math*	constant PI (3.1415927)	78
plot	*f*	*h.stdlib*	multi-purpose graphics plotting	53
point	*f*	*h.stdlib*	return colour at screen coordinate	54
printf	*f*	*h.stdio*	formatted output to **stdout**	46
ptrdiff_t	*t*	*h.stddef*	type for subtracting two pointers	70
putc	*m*	*h.stdio*	put character to a stream	46
putchar	*m*	*h.stdio*	put character to **stdout**	46
puts	*m*	*h.stdio*	put string to a stream	46
rand	*f*	*h.stdlib*	return a pseudo-random integer	54
RAND_MAX	*m*	*h.stdlib*	max value returned by rand (32767)	54

Function	Type	Header	Description	Page
realloc	*m*	*h.stdlib*	changes the size of an object	54
remove	*f*	*h.stdio*	delete a file	47
rename	*f*	*h.stdio*	rename a file	47
rewind	*f*	*h.stdio*	set file pointer to start of file	47
scanf	*f*	*h.stdio*	formatted input from **stdin**	47
SCHAR_MAX	*m*	*h.limits*	constant 127	80
SCHAR_MIN	*m*	*h.limits*	constant -127	80
SEEK_CUR	*m*	*h.stdio*	for use as a third argument to fseek	47
SEEK_END	*m*	*h.stdio*	for use as a third argument to fseek	48
SEEK_SET	*m*	*h.stdio*	for use as a third argument to fseek	48
setjmp	*f*	*h.setjmp*	save function calling environment	67
settime	*f*	*h.stdlib*	set internal timer	54
SHRT_MAX	*m*	*h.limits*	constant 127	80
SHRT_MIN	*m*	*h.limits*	constant -127	80
size_t	*t*	*h.stddef*	type of the result of sizeof operator	71
sound	*f*	*h.stdlib*	generate sounds	55
sprintf	*f*	*h.stdio*	formatted output to a character array	48
srand	*f*	*h.stdlib*	set seed for pseudo-random sequence	55
sscanf	*f*	*h.stdio*	formatted input from a string	48
stderr	*e*	*h.stdio*	standard error stream **stderr**	32
stdin	*e*	*h.stdio*	standard input stream **stdin**	32
stdout	*e*	*h.stdio*	standard output stream **stdout**	32
strcat	*f*	*h.string*	append two strings	58
strchr	*f*	*h.string*	search string for character	58
strcmp	*f*	*h.string*	compare two strings	58
strcpy	*f*	*h.string*	copy a string	59
strcspn	*f*	*h.string*	length not containing specified chars.	59
strlen	*f*	*h.string*	compute length of a string	59
strncat	*f*	*h.string*	concatenate part of a string	60

Function	Type	Header	Description	Page
strncmp	*f*	*h.string*	compare two part strings	60
strncpy	*f*	*h.string*	copy part of a string	60
strpbrk	*f*	*h.string*	locate string within a string	61
strrchr	*f*	*h.string*	locate last occurrence of char. in string	61
strspn	*f*	*h.string*	length consisting of specified characters	61
strstr	*f*	*h.string*	locate string within a string	61
strtod	f	*h.stdlib*	string to float conversion	55
strtok	f	*h.string*	break string into a sequence of tokens	62
strtol	f	*h.stdlib*	string to long conversion	55
system	*m*	*h.stdlib*	pass string to Command Line Interp.	55
tab	*m*	*h.stdlib*	position cursor at specified position	56
time	*f*	*h.stdlib*	read internal timer	56
tolower	*f*	*h.ctype*	convert upper case to lower case	65
toupper	*f*	*h.ctype*	convert lower case to upper case	65
TRUE	*m*	*h.stdio*	constant 1 (also in *h.stddef*)	71
UCHAR_MAX	*m*	*h.limits*	constant 255U	80
UINT_MAX	*m*	*h.limits*	constant 65535U	80
ULONG_MAX	*m*	*h.limits*	constant 4294967295U	81
ungetc	*f*	*h.stdio*	push character back onto input stream	48
USHRT_MAX	*m*	*h.limits*	constant 255U	81
va_arg	*m*	*h.stdarg*	type and value of next argument	68
va_count	*f*	*h.stdarg*	count arguments	68
va_end	*m*	*h.stdarg*	normal return from function	68
va_list	*t*	*h.stdarg*	array type for *va_arg* & *va_end* info	68
va_start	*m*	*h.stdarg*	used before access to unnamed args	69
vdu	*f*	*h.stdlib*	send control code to VDU driver	56

Appendix E

Command mode errors

`Ambiguous command`
The command cannot be unambiguously identified.

`Bad command`
The command cannot be recognised. This message will only be issued when command mode 0 has been set.

`Bad list`
The list of values supplied is terminated incorrectly.

`Bad mode`
The screen mode or command mode specified is invalid.

`Bad negative`
The specified qualifier cannot be prefixed by **NO** to specify a negative value.

`Bad number`
The numeric value supplied is invalid.

`Bad qualifier.`
The qualifier supplied is not recognised as being valid for the command.

`Bad string`
The string supplied is incorrectly terminated; i.e. the closing quote is missing.

`Cant run`
The program is not a valid executable file.

`Escape`
The **ESCAPE** key has been pressed.

`List not allowed`
A list of values is not allowed for the parameter or qualifier.

`No command`
No command has been supplied on the command line.

`No qualifier name`
A slash / has been found, but no qualifier was found following it.

`No value allowed`
A value is not allowed for the specified qualifier.

`Too many values`
Too many values have been supplied for a parameter or qualifier. The maximum allowed is 20.

`Value required.`
A value is required for the specified qualifier.

Compiler errors

These errors fall into three categories: non-fatal, fatal and warnings. After non-fatal errors or warnings the compilier may be allowed to continue compiling. After a fatal error the compilier will stop immediately.

`Bad array size`
An invalid array size has been found in an array declaration.

`Bad constant expression`
The constant expression supplied cannot be evaluated by the compiler.

`Bad #include filename`
The syntax for the filename supplied to #include was not valid. It must be enclosed in either angle brackets or double quotes.

`Bad initialisation`
The initialisation attempted is illegal.

`Bad line no.`
The line number supplied to #line is invalid. It must be a positive integer constant .

`Bad member`
An attempt has been made to declare an illegal member to a struct or union.

`Bad operation`
The arithmetic operation attempted is illegal in this context. For example, adding a floating point value to a pointer.

`Bad string`
A closing quote for a string was not found before the end of the line.

`Bad subscript`
An attempt has been made to subscript an object other than an array or a pointer.

`Bad typename`
Inside a cast, or as the argument to the 'sizeof' function, the typename specified is invalid.

`Can't initialise auto aggregates`
Cannot initialise an array, structure or union with automatic storage class. Make the aggregate 'static' in order to initialise it.

`Can't return value`
An attempt has been made to return a value from a function which is defined as 'void'.

`'char' array required`
An attempt has been made to initialise a non-char array with a character string.

`case or default expected`
A statement other than a 'case' or 'default' label was found inside a switch statement.

`Division by 0`
The optimiser found an expression that caused an attempt to divide by zero. Use the **/NOOPTIMISE** qualifier if this is required.

`EOF in comment`
End of file was found while processing a comment.

`EOF in #if`
End of file was found before a #endif statement.

`Error limit exceeded`
The maximum number of errors allowed in a compilation is 255. This is a fatal error.

`Fields not allowed`
Fields are only permitted within structs or unions.

`Identifier or constant expected`
An identifier or a constant was expected as the next lexical token. This is the message that is printed if no sense at all can be made of the input.

`Identifier expected`
An identifier was expected as the next token in the line. An identifier is any string that is not a keyword.

`#include file not found`
The required #include file was not found on the disc. Check the ***LIB** library.

`Int required`
An integer constant expression is required in an array declaration.

`Label required`
The destination for a 'goto' must be a label.

`Lvalue required`
An attempt has been made to take the address (eg for assignment) of an object for which it is illegal to do so.

`Macro needs parameters`
The macro requires parameters and none have been supplied.

`Multiple 'default'`
More than one 'default' statement was found in a single switch statement.

`Multiply defined external symbol`
It has been attempted to define an external symbol which has already been defined.

`Multiply defined label`
An attempt has been made to define a label which has already been defined in this function.

`Multiply defined local symbol`
An attempt has been made to define a local symbol which has already been defined.

`Multiply defined macro`
An attempt has been made to define a macro that has already been defined. Use #redef if this is required.

`Must be int or unsigned`
Type of bit field must be either int or unsigned int.

`Must be ordinal type`
A bitfield must be an ordinal type, i.e. and not an array or pointer. Arrays of bitfields are not allowed.

`Must be <=8 bits`
The maximum size of bitfield allowed is 8 bits.

`No active loops`
A 'break' or 'continue' has been found outside a loop or switch statement.

`No literal space`
Insufficient space has been set aside for storage of literals during the compilation. The literal space can be changed by use of the **`/LSPACE`** qualifier. The initial buffer size is &300 bytes. This is a fatal error.

`No room`
There is insufficient memory available to complete the compilation. This can occur if a function is too large. This is a fatal error.

`No such member`
The specified identifier is not a member of the specified struct or union.

`Not struct or union`
An attempt has been made to use the '.' or '->' operator on an object other than a struct or union.

`Parameter already declared`
The specified parameter has already been declared in a function header. The following would give this error:

`Parameter name expected`
In a function definition, a reserved word was found where an identifier specifying a parameter name was expected. This can occur if a mixture of the 'function prototype' and normal types of definition is found in a function header.

`Preprocessed line too long`
Expansion of macros in a line caused the line to overflow the allocated buffer. Split the line up. This is a fatal error.

`Size required`
A size is required for the array declaration.

`Stack Full`
The 6502 stack is full. This function is too complex and must be simplified. This is a fatal error.

`Static identifier required`
In the initialisation expression for an external object, an attempt has been made to take the address of a non-static identifier, or the constant expression supplied is invalid.

`Tag already defined`
The struct or union tag specified has already been defined.

`Too big`
The floating point constant was too large to represent in the computer.

`Too few function arguments`
The number of arguments in the function prototype is more than the number of arguments actually supplied to the function.

`Too few macro arguments`
Insufficient parameters have been supplied for the macro.

`Too many function arguments`
The number of arguments in the function prototype is less than the number of arguments actually supplied to the function.

`Too many #ifs`
The maximum nesting allowed for #if is 16.

`Too many initialisers`
Too many initialisers were supplied for the object to be initialised. This may also apply to character string initialisations.

`Too many macro arguments`
Too many parameters have been supplied for the macro.

`Type of parameter already declared`
The type of the parameter has already been declared in a function definition. The following would give this error:

```
function (a,b)
  int a ;
  char b ;
  int a ;

function (a,b,a)
```

`Type spec required`
A type specified was required.

`Undeclared identifier`
A reference was made to an identifier that has not been declared.

`Undeclared label`
At the end of a function, a 'goto' was attempted to a label which is not defined.

`Undefined tag`
A reference has been made to a struct or union tag that has not been defined.

`Unexpected EOF`
End of file was found before all blocks had been closed. This may occur due to previous errors in the compilation preventing a closing brace } for a block being found.

`Unmatched #else`
A #else was found with no matching #if.

`Unmatched #endif`
A #endif was found with no matching #if.

`'while' expected`
The 'while' at the end of a 'do' statement was not found.

`';' expected`
A semicolon was expected following an expression.

`' expected`
A quote character for a character constant was expected as the next lexical token. This can occur if more than one character is found inside a character constant.

`':' expected`
A colon was expected as the next lexical token. This can occur in conditional expressions and after 'case' labels.

`'(' expected`
A '(' was expected as the next lexical token

`')' expected`
A ')' was expected as the next lexical token.

`'{' expected`
A '{' was expected as the next lexical token.

`'}' expected`
A '}' was expected as the next lexical token

`'[' expected`
A '[' was expected as the next lexical token.

`']' expected`
A ']' was expected as the next lexical token.

`')' for macro required`
A close bracket for a macro invocation requiring parameters is require.

Warning messages

`Function prototypes are non portable`
This is flagged if the **/PORTABLE** qualifier is used. A function definition containing a prototype has been found.

`Input line truncated`
Input lines longer than 128 characters are split over 2 lines. This may cause further errors.

`Struct parameter converted to pointer`
A function definition specified that a struct or union be passed as a parameter. This implementation converts these to pointers.

`Symbol name truncated`
Symbol names longer than 31 characters are truncated to 31.

`Type specifier is non portable`
The type specifier is allowed in this implementation, but is not standard.

`Union initialisation is non portable`
This implementation allows unions to be initialised. This is not portable nor standard.

Linker Errors

`Bad byte`
An illegal byte has been found in the specified object file. In theory this error should never occur. It is followed by the following information.

```
   #<value> @<address>
```

where the <value> is the actual value found which is illegal, and the <address> is the offset of the next byte in the file.

`Bad library`
The library file specified has an invalid format. Check the version numbers of the library and the librarian by dumping the library file and checking the header.

`Bad object file`
The specified object file is an invalid Beebug C object file.

`Error limit exceeded`
The maximum number of errors allowed in a linking operation is 255.

`File not found`
The specified file could not be found by the linker.

`Library not found`
The specified library file could not be found.

`Link aborted`
The linking operation has been aborted due to the previous error.

`Multiply defined symbol`
The named symbol has been previously defined. Rename the required symbol.

`No main function`
The program is not executable since no 'main' function has been found.

`No room`
There is insufficient memory available for the linking operation.

`RTSYS not found`
The run-time system file RTSYS has not been found on the disc. This will only occur if the **/STANDALONE** qualifier has been specified.

`Undefined symbol`
The name symbol has not been found. This may occur if the storage class of the symbol definition and a symbol reference do not match, that is, one is external and one is static.

Run-time system errors

`Assertion failed`
The assertion stated in the program at the specified line has failed and the program has been aborted.

`Bad mode`
The mode function has been called specifying an illegal screen mode.

`Bad opcode`
The run-time virtual machine has found an opcode which it did not recognise in the program. Check the Program Counter PC in the register dump for the address.

`Division by zero`
An attempt has been made to perform a division by a zero value.

`Escape`
The escape key has been pressed, thus stopping the program.

`Illegal file access mode`
The fopen function has been called specifying an illegal file access mode as the second parameter.

`No room`
There is insufficient memory available to continue the execution of the program.

`Too complex`
An expression has been calculated which caused the Arithmetic Stack to overflow. Simplify the expression.

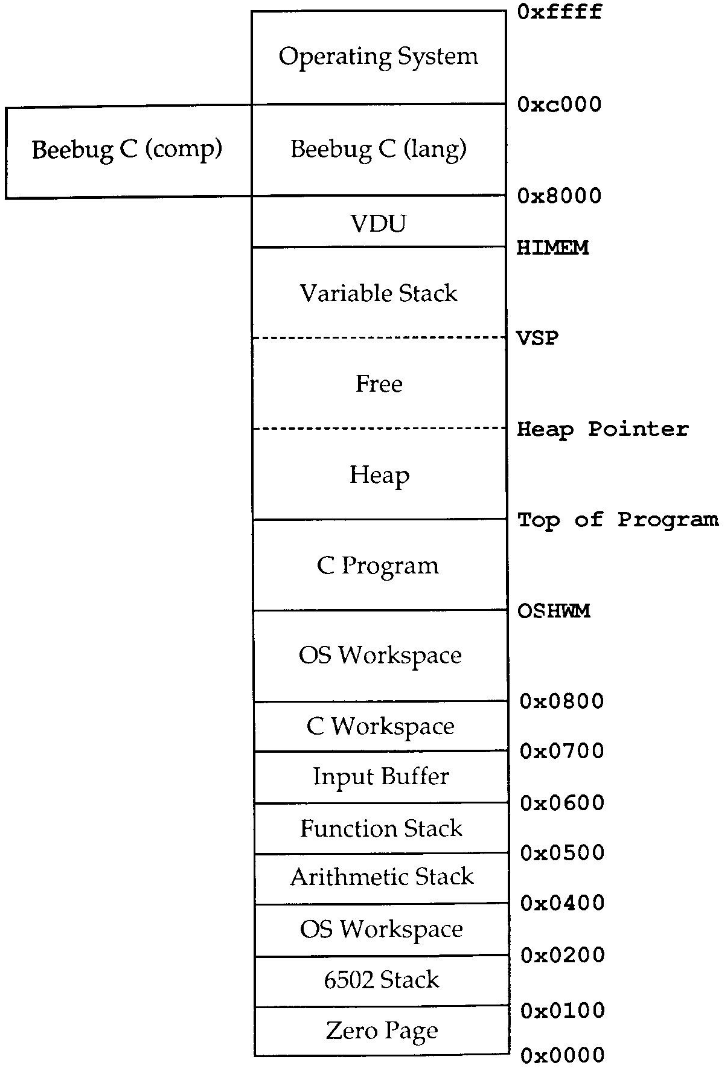
`Too many nested functions`
The Function Stack has overflowed. This is caused by nesting too many function calls.

Appendix F

Technical Information

Memory map

The following diagram shows the memory map for the Beebug C system.



Introduction

This section gives a description of the operation of the virtual machine used to perform the run-time operations of Beebug C. It is intended to be read by those who understand the complexity of run-time systems, or those who wish to further their knowledge of the Beebug C system and get the best out of the system's debugging facilities.

A knowledge of the workings of the virtual machine can enable the programmer to write programs which run faster, and are more efficient.

Machine architecture

The Beebug C virtual machine is a program which implements, in software, the workings of a real CPU. The machine's instruction set has been specifically designed to the requirements of the code generated by the compiler. The machine has a complex instruction set, one instruction performing many functions.

The machine has 6 internal registers:

- Two 32 bit accumulators, called A and B.
- A 16 bit Program Counter, PC.
- An 8 bit Arithmetic Stack Pointer, ASP.
- An 8 bit Function Stack Pointer, FSP.
- A 16 bit Variable Stack Pointer, VSP.

The Accumulators.

The two 32 bit accumulators are used for all the calculations performed in the C program. If the calculation can be performed using only one register, it will be performed in A, and B will not be affected. It is not true, however, that dyadic operations will always have one operand in A and the other in B. B is used only when it is inefficient to use any other method.

When a function returns a value, it places this value in A. If the function does not return a value, the contents of A when it returns are taken as the value if read. This corresponds to the C definition that 'garbage is returned' from a function that returns no value.

The A accumulator is actually 64 bits long, but it is only the bottom 32 bits that are used by C, the upper 32 bits being used for calculation purposes, such as multiplication of 32 bit numbers where the answer is 64 bits long.

Where an indirection operation is to be performed, the contents of the pointer will usually be placed in B. Thus by examining B, it is possible to see whether a pointer contained the correct address. If B is zero, it usually indicates a NULL pointer has been de-referenced, a very common cause of errors.

The Program Counter

The PC register contains the address of the instruction currently being executed. It always points to the operator of the instruction, and not any of the operands. When the instruction has been completed, PC is incremented by the number of bytes required to make it point to the next instruction.

By examining PC, it is possible to find the reason for a 'Bad Opcode' run-time error. This error is usually obtained by a program overwriting itself with data. This is common since static variables are allocated storage in the program itself, and C does not check array bounds. Also, uninitialised pointers can cause data to be written to the wrong place.

The Arithmetic Stack Pointer

The virtual machine uses a Reverse Polish Notation (RPN) method for calculation of expressions. This method requires the use of a stack to hold the operands and result of a calculation. This stack is known as the Arithmetic Stack (AS).

When a calculation is to be performed, the operands required are pushed on to the AS and the operation performed on the stack. The result of the operation replaces the operands on the top of the stack.

For example, suppose the calculation:

```
a + b * c
```

is to be performed. The sequence of instructions required to perform such a calculation would be:

```
LOAD a
LOAD b
LOAD c
MULT
ADD
POP
```

The LOAD operation pushes the value of its operands on to the top of the stack. The MULT and ADD operators take the top two elements on the stack and replace them with their product and sum respectively. This leaves one element on the stack, the result. The POP operation pops the top element on the stack into the A accumulator.

The Arithmetic Stack Pointer gives the address of the top element on the Arithmetic Stack. Before and after a calculation this should be the address of the bottom of the stack. The AS is one page (256 bytes) long, so only the bottom 8 bits of the address are required. The ASP can be regarded as being the offset into the AS.

When the run-time error 'Too complex' occurs, the AS has overflowed. This will occur only when the ASP has the value zero, since the ASP is a 'increment and store' stack pointer and the bottom element of the stack is never used.

Each element on the AS is 4 bytes long, so a maximum of 63 push operations can occur before a pop operation.

Care must be taken while using recursive functions to ensure that the AS does not fill up unnecessarily. This can be done by ensuring that the calculation involved in the recursion is performed before the value is remembered.

The AS grows upwards in memory.

The Function Stack Pointer

When a function is called by the virtual machine, an activation record is kept on the Function Stack (FS). This activation record consists of 3 bytes, which are the number of parameters passed to the function, and the address of the next instruction to be executed on return from the function.

On return from a function, the required number of parameters are popped off the variable stack (see next section), and the PC is loaded with the return address. The value returned from the function is in A.

The FSP gives the address where the next activation record will be placed. Thus it is a 'store and increment' register, as opposed to the ASP which is 'increment and store'.

When the run-time error 'Too many nested functions' occurs, the FS has overflowed. Each record is 3 bytes, therefore a maximum of 85 push operations can be performed without the pop operation.

As with the ASP, the FSP is 8 bits long, giving the offset into the FS as opposed to the absolute address. The FS grows upwards in memory.

The Variable Stack Pointer

There are two types of variables supported by the virtual machine. These are static and automatic. Static variables are allocated static storage in the program space itself, and automatic variables are allocated space on the Variable Stack (VS).

The VS starts at HIMEM and grows downwards towards the heap. When a collision occurs between the VSP and the heap pointer, a 'No room' error is flagged.

The VSP gives the address of the current top of the VS. When an automatic variable is declared in a function, an instruction is generated to decrement the VSP by the number of bytes required by the variable.

For example, if the variable i is declared as an automatic int:

```
auto int i ;
```

the instruction 'decsp' is generated at this point in the code:

```
decsp 2
```

The operand to the instruction gives the number of bytes by which the VSP should be decremented. When the block exits, all automatic variables which have been declared in the block are de-allocated by incrementing the VSP by the required number of bytes.

Function arguments are also pushed onto the VS. Each argument occupies 4 bytes regardless of its type. These are pushed on to the VS before the function is called and the 'rts' instruction pops the required number of arguments back off the VS, thus maintaining the integrity of the VS. This is the purpose of the first byte in the activation record on the Function Stack, the number of arguments.

The Instruction Set

It is not within the scope of this manual to explain every instruction implemented in the virtual machine in detail. This section gives a brief overview of the instruction set.

The code produced by the Beebug C compiler is known as D-code. This has been specifically designed for the virtual machine to be as efficient as possible.

Each D-code instruction consists of a single byte opcode, perhaps followed by a number of operands. The operands provide the instruction with data, such as addresses, offsets etc.

The instruction set can be divided into 5 logical types.

1. Arithmetic operations
2. Load and store operations
3. PC control operations
4. Function call and return operations
5. Stack control operations

Arithmetic Operations

These consist of instructions such as ADD and MULT, as well as more complex instructions such as POSTINC (post increment operator). All of these types of instructions use the Arithmetic Stack for their operation, as discussed later.

Many of the instructions require operands.

Load And Store Operations

These operations control access to variables used within the program. There are many types of load operation, but only one store operation.

A load operation can either load a value from static memory, an immediate value (such as a constant), or load a value from the Variable Stack. They can also load the address of a variable as well as its value.

They load the required value (or address) on to the top of the Arithmetic Stack for use by arithmetic operations.

The store operation takes the value on the top of the AS and places it in the address pointed to by the next value on the AS.

Many of the load and store operators need to know the size of the variable they are dealing with. This is supplied as an operand to the instruction.

PC Control Operations

There are only three instructions of this type. They are JF, JT and JMP. Which are jump false, jump true and jump respectively. They all take the address to which they should jump as an operand.

Function Call and Return Operations

There are two function calling instructions available in D-code, and one return instruction.

The function call instructions consist of a call absolute and a call indirect instructions. Both take as their first operand, the number of arguments passed to the function. The call absolute instruction also takes the address to call, whereas the call indirect instruction calls the address currently in the A accumulator.

The return instruction 'rts' pops the required number of parameters off the VS and pops the last activation record off the FS. It then sets up PC to the return address.

Stack Control Operations

The stack control operations consist of instructions to control the VS and AS. VS control instructions are used to allocate and de-allocate automatic variables and function arguments.

AS instructions are used to push and pop values off the AS explicitly.

Linkable D-code

The Beebug C compiler does not produce executable D-code directly. Instead it produces Linkable D-code. The Beebug C linker takes this linkable code and produces the executable code. In order to enable it to do this, the compiler produces, instead of explicit addressing, symbols which are resolved by the linker to addresses.

To control this, Linkable D-code has built in symbols to control operations, which are removed by the linker. These operations enable symbols to be defined and de-referenced by the linker. In general, at any position in the executable code, where an address is required, a symbol will be generated in the linkable code.

Linkable D-code also contains static variable allocation instructions which enables space to be reserved in the code, and explicit initialisations of statics to be performed.

D-code Disassembler

On the utility disc supplied with Beebug C is a program called DASM. This is a D-code disassembler written in C on the Beebug C compiler. Note that it does not disassemble executable object code. This program is intended for advanced users only, and a technical knowledge of the Beebug C virtual machine is required to understand the disassembled code correctly.

To disassemble the object code for the welcome program given in section 2, type:

```
run dasm o.welcome
```

If for some reason you wish to re-compile dasm, then the following steps should be taken. First compile the two source files **dasm** and **special**:

```
compile dasm
compile special
```

Then link both object files together with:

```
link/nodebug/origin=&2500 dasm,special
```

The executable code can now be run.

Appendix G

Example Programs

This section contains three example programs which, for convenience, are supplied as source files on the library disc. The files are called `c.list`, `c.compare` and `c.compasc`. For clarity, the lengths of the programs have been kept to a minimum, and subsequently they include less error checking than that normally required. However, in normal use they work very well, and should prove to be quite useful.

Each program accepts a number of parameters, which must be typed after the command name. We suggest that you use command mode 2, so that the programs may be executed by simply typing their name followed by any parameters. Remember that the executable code for each example should be in sub-directory **e** of the current directory.

Example 1 - List utility

Although source files may be listed with line numbers using the command **`*LIST`**, there is no facility to list a range of lines or individual lines. This program provides these features. For example:

```
list c.dasm 20 50
```

will list lines 20 to 50 of the file **`c.dasm`** (which is the source file of the object code disassembler). To list an individual line, say line 68, type:

```
list c.dasm 68 68
```

Please note that the last line of the file to be listed must terminate with a carriage return, otherwise it will not be listed.

```
/* List Utility */

#include <h.stdio>
#include <h.string>
#define MINLINE 1
#define MAXLINE 65535
#define MAXLEN 255

main(argc, argv)

int argc;
char *argv[];
{
unsigned int line = 0, start, end;
char string[MAXLEN];
FILE *instr;

start = (argc > 2) ? strtol(argv[2], NULL, 0) :
MINLINE;
end   = (argc > 3) ? strtol(argv[3], NULL, 0) :
MAXLINE;

if ((instr = fopen(argv[1], "r")) != NULL) {
while (fgets(string, MAXLEN, instr) != NULL && line <
end)
if (++line >= start)
printf("%5d %s", line, string);
}

else
printf("File '%s' not found\n", argv[1]);
}
```

Example 2 - Compare utility

This program provides a simple way of comparing the contents of two files. If the files are not the same, the address (in hex) of the first byte at which they differ is displayed. For example type the following:

```
compare c.welcome c.dasm
```

This compares the file **`c.welcome`** with the file **`c.dasm`**. It displays the message Compare failed at 0 since the first byte in each file is different. If no message is displayed then the files are identical. If the files are of different length, but are otherwise identical, an appropriate message is displayed.

```
/* Compare Utility */

#include <h.stdio>
#include <h.string>

main(argc, argv)

char *argv[];

{
int c1, c2;
FILE *file1, *file2;
unsigned long byte = 0;

file1 = fopen(argv[1], "r");
file2 = fopen(argv[2], "r");

if (file1 == NULL || file2 == NULL)
printf("File not found\n");

else {
do {
c1 = fgetc(file1);
c2 = fgetc(file2);
++byte;
}
while(c1 == c2 && c1 != EOF);
if ((c1 == EOF && c2 != EOF) || (c1 != EOF && c2 ==
EOF))
printf("Files are different lengths\n");else
if (c1 != c2) printf("Compare failed at 0x%x\n", --
byte);

}
}
```

Example 3 - ASCII compare utility

The previous program is useful for checking if two files, of any type, are identical. This example is written specifically for comparing ASCII files. It scans through two files line by line, displaying any lines that are not identical. It is particularly useful for locating differences between C source files. For example:

```
compasc c.prog1a c.prog1b
```

compares the ASCII file c.prog1a with the file c.prog1b.

```
/* ASCII Compare Utility */

#define MAXLEN 255

#include <h.stdio>
#include <h.string>

main(argc, argv)

char *argv[];

{
int charnum;
FILE *file1, *file2;
char string1[MAXLEN], string2[MAXLEN];
unsigned long linenum;

file1 = fopen(argv[1], "r");
file2 = fopen(argv[2], "r");

if (file1 == NULL || file2 == NULL)
printf("File not found\n");

else {
for (linenum = 0; (fgets(string1, MAXLEN, file1) !=
NULL) && (fgets(string2, MAXLEN, file2) != NULL);
++linenum)

if (strcmp(string1, string2) != NULL) {
printf("%10s %5d %s", argv[1], linenum, string1);
printf("%10s %5d %s\n\n", argv[2], linenum, string2);
}
}
}
```