HiSoft C

Fast Interactive K&R C Compiler

System Requirements:
Amstrad CPC computer running cassette or disc
Any CP/M system with at least 38K TPA
Tatung Einstein computer
MSX disk-based computer

First Edition July 1985
Second Printing October 1987
Second Edition May 1988

ISBN 0 948517 12 3

If you own one of the computers below then please take some time to read the notes specific to your computer ...and remember ...always make a backup first!

Amstrad CPC/PCW Owners

If you have bought the disc version of HiSoft C to run on your Amstrad Z80 computer then this manual describes two compilers; the native compiler which is on side A of your disc and the CP/M compiler which is on side B of your disc.

This manual describes both the CP/M and native compilers and sections specific to either compiler are carefully marked with icons. These icons are described in Chapter 1.

Note that the CP/M compiler must run under CP/M (so boot CP/M first) and that it comes with a GSX graphics library (described at the end of the manual) while the native compiler is supplied with an extra firmware library (no descriptive notes are available for this library, the source code documents itself).

If you bought HiSoft C on cassette, you only have the native version of the compiler; the CP/M version is available as an upgrade.

MSX Owners

The HiSoft C compiler runs on disc under MSXDOS which is supplied on your C master disc. Make a backup copy and boot your computer with the backed-up disc. If you do not understand how to use MSXDOS then a separate booklet describing MSXDOS commands and low-level access is available from us at a cost of £3.50 inclusive.

Note that there are some extra libraries supplied for MSX owners; these either document themselves or have instructions included on the disc.

Tatung Einstein Owners

Your HiSoft C compiler runs under the DOS system. To make a backup you must first format a disc with the DOS system tracks on it (use BACKUP to do this). Then use COPY to copy all the files from your HiSoft C disc onto your newly-formatted disc. There is a bug in the DOS system which may cause the editor installation program to crash and also may cause you problems in using files with C. To cure this bug, you should run the FIXDOS program supplied by us to patch the DOS; follow any instructions given by FIXDOS.

An extra library of useful funtions is supplied for Einstein owners called EIN.LIB; see EINLIB.HLP for information on this library.

Note that there may be two chapters headed Chapter 6; one describes the CPM.LIB library which is specific to CP/M implementations of HiSoft C whilst the other details BASIC.LIB which is Amstrad CPC specific.

Contents

1. Introduction

2. The Editor

3. Language Summary

4. The Expert's Guide

5. HiSoft C Standard Function Library

HiSoft C
CP M SPECIFIC

6. The CPM.LIB Library

HiSoft C
CPC SPECIFIC

6. The BASIC.LIB Library

HiSoft C

Fast Interactive K&R C Compiler

Chapter 1

Introduction

HiSoft

High Quality Software

1. Introduction

This manual describes the HiSoft implementation of the programming language C on Z80 CP/M computers including the entire range of Amstrad CPC and PCW machines, MSXDOS computers, the Tatung Einstein, the Spectrum +3 (under CP/M) and many more. It also describes the implementation of **HiSoft C** running on Amstrad CPC machines without disks and on Amstrad CPC machines with disks but using the AMSDOS operating system, instead of CP/M.

1.1 Conventions

So that you can easily identify which parts of the manual are pertinent to your particular computer we have marked those sections with the following icons:

indicates that this section or sub-section of the manual applies *only* to the Amstrad CPC version of **HiSoft C** i.e. the version that does not run under CP/M.

indicates that this section or sub-section of the manual applies *only* to the CP/M version of **HiSoft C** and not to the Amstrad CPC version running under the native operating system.

1.2 What is HiSoft C?

C is a general purpose programming language that places the emphasis on concise programs and flexible expressions. The user is provided with little protection - it is possible to write elegant, powerful programs but it is also possible to write amazingly obscure bugs - this is what makes it so powerful!

The HiSoft implementation of C is a compiler designed with home computers very much in mind. The emphasis is on the speed of development: compilation is extremely fast (we don't know of a faster one), and the overall development cycle is made much faster and less wearing on the fingers by automatic entry to the editor on detection of errors. There is no intermediate pass through assembly language or linkers such as is normally found in C compilation systems. This means that program development can be very rapid indeed.

The manual is divided into chapters which each discuss one major aspect of the compiler. This first chapter provides a general introduction. Following that is a chapter on the editor and how to create programs. The third chapter is a detailed description of the C dialect accepted by the compiler with examples of every part of the language. The fourth is an expert's guide to the language. The next two describe the extensive range of functions supplied with the compiler in the libraries. Finally there is a chapter devoted to errors - the messages they produce and how to find them.

We try to keep as much of the manual as possible the same for different computers (eg CP/M, Amstrad and Spectrum). This is partly because the compiler is compatible on the different micros; and partly because it reduces the chances of errors in the manual. We hope you will accept the odd piece of information which isn't relevant to your computer, and that you find it useful if you have more than one. Chapters One, Two and Six have machine-specific information in them, while chapters Three, Four, Five and Seven are general.

The C Programming Language is the title of a book by Brian Kernighan and Dennis Ritchie which provides the best definition of the language. It is also a good tutorial introduction to C. We do suggest that you read this book; perhaps you can borrow it from the library if you feel it is too expensive. Full details of the book are given in the Bibliography. Some other less expensive books are now becoming available and details of some of these can also be found in the Bibliography.

The implementation is designed to be as close as possible to the definition given in the *C Referernce Manual*, which is Appendix A of the Kernighan & Ritchie book mentioned above. There is a chapter in this manual - **Expert's Guide** - which describes in detail the differences between this implementation and the definition. There is one main omission - floating point arithmetic and one main area of difference - linkage of modules. There are also some minor differences for technical reasons.

1.3 CP/M Users

1.3.1 Getting Started - READ THIS FIRST

Before running the compiler, we suggest that you make a backup as discussed below. After doing that you may need to install the full-screen editor so that it uses the correct screen control codes for your particular computer. This is explained in the separate **ED80** manual. You will then be able to follow the example given in **Examples** which shows exactly how to type in a program, compile and run it.

The compiler, editor and various other files are provided on the disc; although the exact contents of the disc may vary as we improve the product. In particular any late news will be on the disc in a file called `READ.ME` because it is easier to change the disc than the manual. So have a look at the disc directory and if there is a `READ.ME` file read it now by using either **ED80** or the CP/M TYPE command.

When calm prevails again please read the rest of this chapter and at least skim through those on the editor, the language and the library so that you have a general idea what is there. You can read the chapter on errors when the need arises (of course you may never need it ...).

1.3.2 Using the Compiler; the Command Line

The compiler can be driven in the usual CP/M style by typing the name of the compiler followed by the name of the source program:

```
A>hc my-prog.c [ENTER]
```

It will produce a ready-to-run file called MY-PROG.COM as its output. This can of course be run just by typing its name:

```
A>my-prog [ENTER]
```

Note that the compiler must be called hc.com in order to be restarted properly by the editor. The filename extension of the source file (e.g..c) must not be omitted, but it can be any extension that you wish. An extension of .C is conventional.

A whole series of source file names can be typed on the command line instead of just one source file. The output COM file will be named after the first source file. You can cause the output binary object file to have a different name by using the #translate directive, either at the top of the source file or on the command line.

Compiler directives can also be included on the command line, in order to influence the behaviour of the compiler. This is particularly useful for #error, #list, #data. The directives are all described in the **Language Reference Guide**. (In fact the source file names are handled by simulating a #include directive inside the compiler!).

Command line arguments are separated by spaces. They can be enclosed in quotes if necessary. Eg:

```
A>hc four arguments including "one argument with spaces"

A>hc "#translate object.com" ex.c #list+ #list+ ?library?
```

1.3.3 Invoking the Editor after a Compilation Error

The compiler and editor are designed to form an integrated development environment. What this means is that after finding a compilation error the compiler will invoke the editor on the faulty line. After you have corrected the error (which may involve editing as many files as necessary) the editor will restart the compiler automatically. Full details of this powerful time-saving technique are given in **Chapter 2**.

1.3.4 Command Line Handling

Command line handling is provided by **HiSoft C**, but there are certain differences from UNIX command line handling as described in Kernighan & Ritchie. We have however tried to retain as much compatibility as possible. The compiled program can access a series of arguments supplied on the command line, and the standard input (stdin) and standard output (stdout) are automatically redirected by command lines such as:

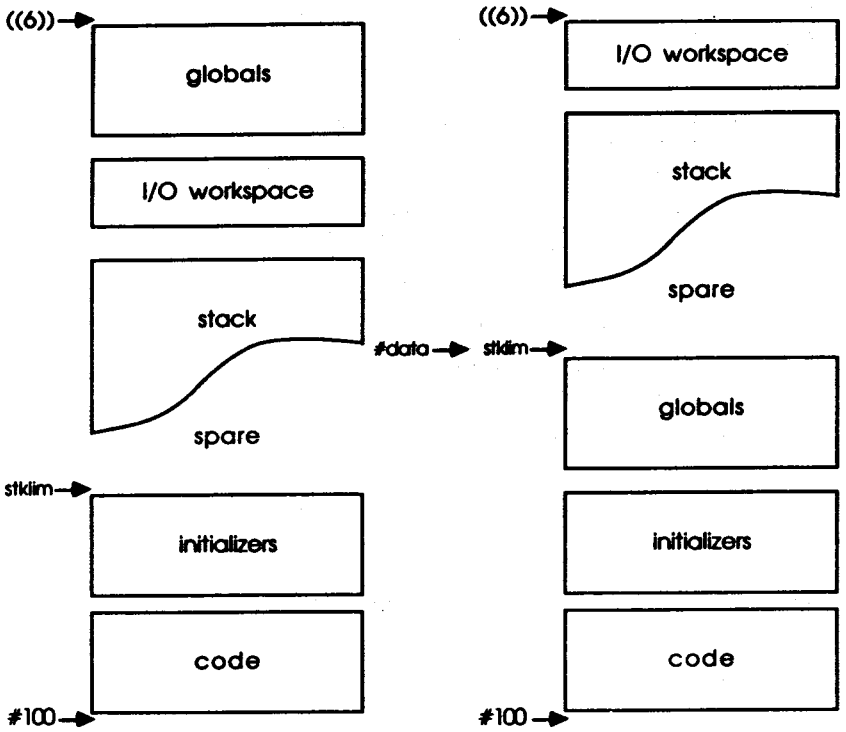
```
A>program arg1 <input arg2 arg3 >>append arg4
```

Full details of what is allowed on a command line are given in **Chapter 6**.

1.3.5 Memory Layout & the Memory Usage Report

The memory map of a compiled program can be important both for production programs and when debugging a program. There is a diagram of the overall layout on the next page, and a full explanation of the map in **Chapter 4**. The layout of a **HiSoft C** program can be controlled using the #data directive which is explained in **Chapter 3**.

The Memory Usage Report is displayed at the end of compilation. It shows the area of memory occupied by the various parts of the compiled program and also suggests a value to be used with the #data directive if portability is desired. An example of the report is shown in **Examples**.



1.3.6 How Input and Output Work with C

Input and output in C is done by a set of standard library functions. These functions do not form part of the C language itself but are modelled after those provided on UNIX systems. UNIX is the name of the operating system used where C originated and it is used as a reference for input-output in C.

Input and output in C is done serially via *files* These files cover not only what is normally thought of as files (on disc, tape etc) but also input from the keyboard and output to the display and other devices. This goes some way towards making input and output appear to be device independent as far as the C programmer is concerned.

There are three standard files in a C program. These are the standard input stdin, the standard output stdout, and the standard error output stderr These are assigned to the keyboard stream, text display stream and text display stream (again) respectively. They can be redirected to other devices or files by using the standard library function freopen. The names stdin, stdout, and stderr are defined in stdio.h for use in your programs.

All output is done by the function putc() although this will usually not be called directly by the programmer but rather will be called by a more sophisticated function and in particular by printf() which is the workhorse of C output. Similarly all input is done by getc() which has a similar relationship with scanf()

getc and putc need to know which file the input or output is directed at. They do this by taking a *file-pointer* as an argument (file-pointers are also often called *stream-numbers*). The three standard files stdin, stdout, and stderr are actually file-pointers. There are many special functions which use these files without needing to specify them explicitly. For example characters are input from the keyboard using the standard function getchar() and output to the display using putchar(). These correspond to getc(stdin) and putc(c, stdout) respectively.

Before using putc or getc or any other general file function you must have a file-pointer. You can use stdin and stdout as file-pointers for the keyboard and display as mentioned above. Before using another device such as a disc or serial line or printer it is necessary to call the standard library routine fopen(filename,mode) first to get the file-pointer. getc and putc can then be used to read or write characters and finally fclose tidies up and writes the last block of an output file.

The characters are collected into blocks before being put in a file. The input and output done by the compiler is done in the same way as by user programs so you can create or read C source files using your own programs. The memory for a block buffer is also arranged by fopen and fclose. The normal system messages are produced for disc and cassette handling.

1.3.7 CP/M Device Names

There are several special built-in filenames which refer to CP/M devices. These device names are used just like normal filenames in calls to fopen and freopen. In fact the runtimes set up the stdin file using the TRM:, and the stdout and stderr files using the CON: device name. For example:

```
list_file = fopen("LST:", "w");
fprintf(list_file, "Hello Hard-Copy World\n");
fclose(list_file);
```

The full list of devices names is as follows:

CON: Buffered Input & Output to console (uses BDOS functions 1 & 2).

RDR: Input from the reader (AUXIN on CP/M+) (uses BDOS function 3).

PUN: Output to the punch (AUXOUT on CP/M+) (uses BDOS function 4).

LST: Output to the list device (usually a printer) (uses BDOS function 5).

KBD: Direct I/O usually done by rawin() & rawout() (uses BDOS call 6).

TRM: Line buffered console input (uses BDOS function 10).

1.3.8 Files on CP/M

This section discusses the way in which **HiSoft C** has been fitted to CP/M computers and focusses on the input-output system which is the main part of the tailoring. CP/M file handling does not correspond very well with that expected in a C environment, and a great deal of support has to be provided by the runtime system of the compiler in order to provide a smooth interface.

The differences force us to divide file accesses into two principal types: *text* and *binary*. This need was foreseen by the original implementors of UNIX and we have adopted the solution which they proposed. There are two particular areas of difference which force this upon us, which are *end-of-line* and *end-of-file* handling. We will discuss these two problems first, and then move on to consider how we circumvent them by using two kinds of file access. **Chapters 5** and **6** contain full descriptions of the many library functions which we refer to.

1.3.9 End Of Line Handling

In C a line of text is separated from the next one by a single character called a NEWLINE. This is often abbreviated to NL and is represented within a C program as `\n`. It has a numeric value of 10. This convention is used on many large computers and some micros including the Sinclair QL. But many micros (including CP/M) use a different, older convention. On these, a line of text is separated from the next by two characters: a CARRIAGE-RETURN followed by a LINE-FEED. A CARRIAGE-RETURN has a numeric value of 13 and a LINE-FEED has a numeric value of 10 unfortunately!

So on input from a text file a CARRIAGE-RETURN + LINE-FEED is adjusted by the C runtime package so that it appears to produce a single NEWLINE character with a value of 10. And on output a value of 10 is made to produce the two values 13 and 10 in sequence.

These translations must not be made when reading or writing a binary file, because there is no special significance attached to the values 10 and 13 in a binary file. This then is the first reason that forces us to distinguish between text and binary access.

1.3.10 End Of File Handling

C programs expect their files to behave like those on a UNIX system because the design of the library functions comes from there originally. On UNIX a file can be thought of as a "sequence of characters", and the exact number of characters in the file is remembered by UNIX. So when reading a file character-by-character, a C program is passed each character in turn until they have all been seen; after that the program is passed the special value of -1 (ie `0xFFFF`) which is also known as EOF.

CP/M files are much more complicated and less precise about their end. CP/M only remembers how many "records" of 128 characters there are in a file, and cannot tell how many of the characters in the last record really are part of the file. For instance if a file actually consisted of 133 characters, there would be two records in the file, but only the first five characters in the second record would be meaningful.

In text files CP/M programs write a special character just after the last real character so that we can tell when we have reached the true end of the file. The special character is control Z (`[CTRL]-Z`, or `0x1A`, or decimal 26).

This in itself raises a problem because it means that we cannot have a `[CTRL]-Z` value in the file. Now `[CTRL]-Z` represents a perfectly ordinary Z80 machine code instruction (`LD A,(DE)`) so here is another reason why we cannot treat text and code files in the same way on a CP/M computer.

But it is even worse than that because some CP/M programs don't always write a [CTRL]-Z at the end of text files (they leave it off if the last character is exactly at the end of a record). This can also happen if the file wasn't written properly. So C programs must check for the end of the last record as well as for the last character! Another problem is brought by some supposedly CP/M compatible systems which behave differently and for example may expect every character in the last record after the last real character to be [CTRL]-Z.

There are two more pieces of bad news; one of which prevents us *appending* to a binary file properly.

The first piece of bad news is that CP/M-Plus can actually record the number of valid bytes in the last record and so calculate the true file size. In order to be compatible with CP/M 2 we don't feel able to use this. It also bears on another ambiguity in CP/M which you need to be aware of. If there are two or more [CTRL]-Z characters in a text file then we have to choose one to represent the end-of-file. *There is no unique way of doing this!* Any program accessing a file serially (ie most programs) will find the first [CTRL]-Z in the file and treat it as end of file. But the CP/M-Plus trick implies that the end of file is always in the last record and that is not necessarily true. The safe rule is that you must ensure that all text files have exactly one [CTRL]-Z in them and that must be in the last physically allocated record. *The C runtimes work on this assumption and actually choose the first [CTRL]-Z in the last record as the end of file (or choose physical end of file if there is no such [CTRL]-Z).*

The second piece of bad news is that since there is no equivalent end of file marker in a binary file it is impossible to find the exact end of file at all. So when you do a seek to the end of a binary file (which is implied when you open a file for appending) you are sent to the end of the last record in the file. This will almost never be where you wanted to be, and will only be guaranteed to coincide if your file contains data structures which are all multiples of 128 bytes. Most unlikely! All we can suggest is that you allocate the first four bytes of the file to hold the actual file size. Then after writing the file and just before closing it use tell32 to get the actual file size and write it into these bytes. The first thing to do after reopening the file at a later time is to read the first four bytes to get the actual file size. This can be passed to _seek in order to append to the file. *Do not try to use a mode2 seek with a binary file.*

1.3.11 Summary of CP/M File Modes

In summary there are two modes of access which can be applied to any file: *text* and *binary*. Binary mode provides straightforward access to each byte in the file, whilst text mode *filters* the ends-of-line and end-of-file to be what a C program expects. This fixes most of the difficulties but does occasionally produce surprising results. The reasons for this mixup are buried in the history of computing and telegraphy and it is small comfort to know that sometime in the future everything will be done the C way!

1.3.12 Special Keyboard and Display Handling

There are a couple of points which follow on from the preceding discussion and which concern the keyboard and display. These are usually handled as text files and this results in some keys appearing to have different values to normal.

[CTRL]-Z (ie press the Z key whilst holding down the [CTRL] key ([ALT] key on the Amstrad PCWs)) is used as the End-Of-File key because this is the key normally used on CP/M systems and many users will already be familiar with it. But note that when writing C programs this key appears to have a numeric value of -1 rather than the 26 you might expect. This is because C programs the world over expect -1 for EOF.

[ENTER] (also known as [RETURN], [CARRIAGE RETURN], or [CR]) is also different from what you might expect, because of the difference between the C end-of-line and the CP/M one. The [ENTER] key is adjusted by C so that it appears to produce a value of 10, and on output a value of 10 is made to produce the two values 13 and 10 in sequence.

There are also two special library functions (rawin and rawout) which sidestep these conversions and do direct I/O via CP/M BDOS function 6.

1.3.13 Making a Backup

We strongly suggest that you make a backup copy of the distribution disc before using the compiler. How to do this depends on the particular computer that you own; but it can usually be done by copying all the the files using the program PIP. Please read the manual that came with your computer.

Keep the original disc that we supply since you will need it for any upgrades.

Please note that while we encourage you to make a backup, this is not a licence to make extra copies for other computers. You are entitled to run the compiler on one computer only; if you are a company or institution which intends to use the compiler on more than one computer then you should contact us for details of multiple copy prices. Please read the next section also.

1.4 Amstrad non-CP/M

1.4.1 Getting Started - READ THIS FIRST

1.4.1.1 Loading from Cassette

To start running the **HiSoft C** compiler, load the cassette into your recorder with the label COMPILER facing upwards and type [CTRL]-[ENTER] (using the *small* [ENTER] key !). Press PLAY and any other key. The compiler will now load and autorun.

1.4.1.2 Loading from disk

From disc the compiler is run by typing:

```
run "HISOFT-C" [ENTER]
```

1.4.2 A Quick Example

The compiler is a single binary file called HISOFT-C on cassette and HISOFT-C.BIN on disc. There is also a fast loading 2000 baud version on the reverse of the compiler cassette. You may well be able to load this but its not guaranteed, because it is dependent on the alignment of the tape mechanism.

After loading, the compiler will display its sign-on message and enter the editor:

```
HISOFT-C Compiler V1.2
Copyright C 1984 HISOFT

>
```

After the sign-on message you will see a cursor, and you can now type a C program. First activate the compiler by typing `c` followed by `[ENTER]` and then a short program (using `[SHIFT]-[` for { and `[SHIFT]-]` for }):

```
>c

HISOFT-C Compiler V1.2
Copyright C 1984 HISOFT

main()
{
printf("Hello world");
}
```

Now type `[CTRL]-Z` (for *End Of File*) and the compiler will ask you:

```
Type y to run:
```

Type `y` and you will see the program run and prompt you again. You can rerun the program any number of times by typing `y` but this time just press `[ENTER]` and it will then go back to the editor:

```
Hello world
Type y to run: [ENTER]
>
```

You will probably now want to try a longer program (perhaps one of the examples given later in this chapter) and to do that we suggest that you turn to the next chapter on using the editor, rather than typing straight to the compiler.

After giving the `c` command to the editor the compiler is sitting just behind your keyboard and everything you type is sent straight to it. One thing that you will often want to type to the compiler is:

```
#include
```

to make it compile a program that you have created using the editor, or if you have put the program onto a tape as a file called `fred` then just type:

```
#include fred
```

When calm prevails again please read the chapters on the language and on the library - or at least skim through them so that you have a general idea what is there. You can read the chapter on errors when the need arises (of course you may never need it ...).

1.4.3 The Amstrad Keyboard

The keyboard is mainly laid out in a common way, and the compiler tries to use it in the same way as the Locomotive BASIC, so you should have very little trouble in using the C compiler. There are a few points worth making about particular keys.

Curly braces { } are actually provided on the keyboard but are not marked on the CPC464! They are above the square brackets [ ]. Left brace is `[SHIFT]-[` and right brace is `[SHIFT]-]`. These are rather important keys in C and you will quickly get used to them.

The vertical stroke | is shown on the keyboard at [SHIFT]-@ with a gap in the middle, although it is displayed on the screen without the gap. These are just alternative ways of showing it. The vertical stroke is used in C in logical expressions to mean 'or'.

The circumflex ^ is shown on the keyboard and screen as an arrow pointing upwards (next to the [CLR] key). This again is an alternative way of showing it. Don't confuse it with the cursor up key. The circumflex means 'exclusive-or' in C.

End-Of-File, or EOF, is an important concept in C and we have a key to represent it. We use [CTRL]-Z (ie press the Z key whilst holding down the [CTRL] key) because this is the key used on CP/M disc systems and many users will already be familiar with it. The C compiler recognises the end of a compilation when this key is pressed. Note that when writing C programs this key appears to have a numeric value of -1 rather than the 26 you might expect. This is because C programs the world over expect -1 for EOF. When you type [CTRL]-Z you will normally see a double ended horizontal arrow. This is the graphic character for a numeric value of 255, which is what -1 becomes when chopped down to a single byte.

[ENTER] is also different from what you might expect, because the C world uses a different convention from the small micro world. In C a line of text is separated from the next one by a single character called a NEWLINE. This is often abbreviated to NL and is represented within a C program as \n. It has a numeric value of 10. This convention is used on many large computers and some micros including the Sinclair QL. Many micros (including the Amstrads and CP/M) use a different convention. On them, a line of text is separated from the next by two characters: a CARRIAGE-RETURN followed by a LINE-FEED. A CARRIAGE-RETURN has a numeric value of 13 and a LINE-FEED has a numeric value of 10 unfortunately! The [ENTER] key is adjusted so that it appears to produce a value of 10, and on output a value of 10 is made to produce the two values 13 and 10 in sequence. This fixes most of the difficulties but does occasionally produce surprising results.

The reasons for this mixup are buried in the history of computing and telegraphy and it is small comfort to know that sometime in the future everything will be done the C way!

[ESC] is used to escape from various things. It is handled in rather a clever way by the computer and this can produce side-effects. Please read the special section below. When you use the [ESC] key you may see various graphics including a rocket ship! This is quite OK.

1.4.4 Function Keys

You will often have to type the same thing repeatedly when writing programs or operating the compiler, and you may find that it saves time to set up some function keys to help you. The Amstrad computers let you use up to 32 function keys but it is easiest just to use the ten on the numeric keypad. Some people like to set up keys so that they can type keywords like `while` or `continue` with one keystroke; and another useful sequence to have on a function key is:

```
c [ENTER] #include [ENTER]
```

which lets you compile a program from the editor with just one keystroke. You can use a C program to set up the function keys (see **BASIC.LIB**).

1.4.5 Files on the Amstrad CPCs

This section discusses the way in which **HiSoft C** has been fitted to the Amstrad CPC computers and focusses on the input-output system which is the main part of the tailoring. We start out with a basic knowledge of the computer and of the C input-output functions.

The functions provided in the standard library are modelled after those provided on Unix systems. Unix is the name of the operating system used where C originated and it is used as a reference for C compiler input-output. Input and output in C is done serially via *files*. These files cover not only what is normally thought of as files (on tape, disc etc) but also input from the keyboard and output to the display.

These files greatly resemble the streams provided on the computer and indeed on later versions of Unix *streams* is what they are sometimes called. This goes some way towards making input and output appear to be device independent as far as the C programmer is concerned.

There are three standard files in a C program. These are the standard input stdin, the standard output stdout, and the standard error output stderr. These are assigned to the keyboard stream, text display stream 0 and text display stream 0 (again) respectively. Some translations are done on the characters as discussed above for the keyboard. If you want to use these names in your programs then define them like this:

```
#define stdin 0
#define stdout 0
#define stderr 0
```

Input and output to other devices is done to files using *file-pointers*. The file-pointers used for input and output in C are used to represent stream numbers on the Amstrad. They have type pointer-to-FILE so that they are compatible with file-pointers on other systems, but actually their value (as a bit pattern) is just the stream number as in BASIC. So streams 0-7 are screen windows, 8 is the printer, and 9-10 are cassette.

Characters are input from the keyboard using the standard function getchar() and output to the display using putchar(c). These correspond to getc(0) and putc(c, 0) respectively.

Before using putc or getc you must make sure that you have a file-pointer. You can use 0 as a file-pointer for the display and keyboard as mentioned above. You can also use the screen streams 1-7 and the printer stream 8. All that is necessary to send output from a program to the printer is to use stream 8. For example:

```
putc('X', 8); fprintf(8, "\nThe answer is %d", 42);
```

Before reading from or writing to tape or disc it is necessary to call the standard library routine fopen(filename, mode) first to get the file-pointer. getc and putc can then be used to read or write characters and finally fclose tidies up and writes the last block of an output file. All input and output on cassette or disc is done using the functions getc(fp) and putc(c,fp). The characters are collected into blocks before being put in a file. The input and output done by the compiler is done in the same way as by user programs so you can create or read C source files using your own programs.

The memory for a block buffer is also arranged by fopen and fclose. The normal system messages are produced for disc and cassette handling. Note that you can only have one file open at once except you can have one input file and one output file simultaneously if you call setbufout first.

1.4.6 Breaking Out of Programs with the [ESC] key

You can pause or break out of a running C program by using the [ESC] key. It can also be used to break out of a compilation, and out of a listing to screen or printer.

Pressing [ESC] causes the computer to pause, and pressing any other key causes it to resume (eg press [ENTER]). Pressing the [ESC] key for a second time whilst paused breaks back to the editor. There is a short delay before keyboard scanning resumes so be sure not to type anything more until you can see the editor > prompt and the cursor again.

The computer may pause in the middle of drawing a character on the screen or even in the middle of scrolling the screen up (which may leave old text in the bottom right corner of the screen). This is normal. If you are interested, there are some details of why this happens in the next section.

1.4.7 The Firmware

Inside the Amstrad are some programs in ROM which are known as the *firmware*. These are called to use the features of the computer such as the screen, keyboard, cassette, disc etc. The compiler makes use of them and you can also make use of them in your own programs. These programs are fully described in the *Firmware Specification* (see **Bibliography**) and you should refer to this to learn what they do. We type their names using capital letters and underscores throughout this manual. A library of C functions is available which lets you access each of them.

The way that the firmware is used by the compiler may have an effect on your program's use of the firmware and so we give a comprehensive description here for those who find they need to know. You don't need to read or understand the rest of this section to make normal use of the compiler.

The first thing is that the compiler runs as a foreground program in order to gain access to the maximum possible memory. A return to BASIC therefore involves a complete restart with no possibility of saving any information. The normal way out is to press [CTRL]-[SHIFT]-[ESC] which restarts as though the computer had just been turned on. It is also possible to use the slightly less severe |BASIC command but we don't recommend this because it can leave BASIC confused and you may receive strange error messages when you start typing commands.

Starting the compiler as a foreground program involves calling KL_CHOKE_OFF which means that all external commands are lost. Commands from background ROMs such as the disc are reinstated by calling KL_ROM_WALK. There is no provision in the firmware to recover RSX commands except by execution of a program. The loaders supplied with commercial RSX programs are likely to assume a BASIC environment and would therefore need adaptation. There are also strong technical difficulties which make it more problematical.

SOUND_RESET , KM_INITIALISE and JUMP_RESTORE are also called along with KL_CHOKE_OFF. The call to KM_INITIALISE is necessary because under interrupts the keyboard manager buffer pointers can get out of step and require to be reset. JUMP_RESTORE is necessary to keep the disc commands such as |tape.in and |disc in good working order.

The initialisation sequence described above is also called every time the [ESC] key is used to abort a listing. This draconian step is taken to make sure the firmware is able to run, but does result in a number of things being set to their initial state and in particular prevents access to RSXs.

Special facilities are provided by the firmware to support the [ESC] key so that it can be used to break out of running programs. The compiler makes use of this feature to abort listings, compilations, and running user programs.

The feature is used not just because it is neat but also because it provides a way around the slow running that occurs if the keyboard is actively polled for a break key. In a compiled program this has to be done very frequently because of the speed of the compiled code and the necessity to check inside every loop and every function call to guarantee being able to break out of non-terminating programs. The firmware calls used to do this cause a significant time penalty which slows down the program in relation to identical programs on other computers.

The event-handling capabilities of the computer contribute to this problem as well. The implementation technique known as *synchronous events* is used to avoid certain real-time race conditions in user programs. Unfortunately regular polling is used as part of this technique and this has to be inserted into the frequently executed checking code in a compiled program. It would be easy to perform this polling in the main control loop of an interpreter, precisely because the interpreter runs at a much slower rate.

But there is hope! Another weapon in the firmware's armoury is the *asynchronous event* which drops unannounced much like the stone from a siege catapult. We can arrange for an asynchronous timer event to occur regularly at some slower and more acceptable rate. This event performs the polling of the synchronous events which is required (and this includes the [ESC] key as a special case which solves our original problem). But this is a nuclear age and there are two kinds of fallout from this final solution.

The first consequence follows from various unfortunate restrictions on what an asynchronous event may do - and that is what our [ESC] key is now disguised as. It turns out that in particular it cannot gently transfer control of the computer back to the compiler, but must do it in the very brutal fashion explained above.

The second consequence is that since all synchronous events are now dressed in the appearance of an asynchronous event, they no longer protect us against the race conditions they were designed to avoid. This is why the screen can be left in a wayward state when the [ESC] key is used to abort a listing.

1.4.8 Stand-alone Programs

If you write a C program which you use a lot, or that you want to sell or give a copy of to someone, then you will want to make a stand-alone version of it. This saves the time and fuss necessary to load the compiler before using your program, and lets you give or sell a working copy to someone who doesn't own the compiler.

The facility to make stand-alone programs is called #translate and details of how to use it are given in the **Language Reference** Chapter.

The program is executed immediately after it has been saved to disc or cassette.

A stand-alone program is loaded and run from BASIC in the usual way:

```
RUN"program_name
```

It does not permit users to break out by using the [ESC] key, but [CTRL]-[SHIFT]-[ESC] can be used to restart BASIC.

After the program finishes executing it will restart the machine (it is difficult to do anything else). If this will erase information that you have put on the screen, then put in a delay loop to allow it to be seen, or use a call to getchar() so that the user can press [ENTER] when he has finished looking at the screen. Eg:

```
printf("Press [ENTER] when finished");
getchar();
```

1.4.9 Making a Backup

Your compiler cassette is supplied with a lifetime guarantee, so if you have any problems loading the tape or if the cassette is damaged just get in touch with us and we will exchange it for an identical working copy. Compare this with the guarantee that you get with your copy of Kernighan & Ritchie or with your computer. If you have the cassette version and now need a disc version we provide an upgrade service. Please write to us or give us a call.

You may want to copy the function library so that you can change the functions or add new ones; and if you have discs then you may well want a working copy on disc.

The function library is supplied as normal C source text files, and you can use the compiler itself to make a copy. The function library comes in several parts and we are going to read each part of the library into the editor's buffer in turn and write it out again. At this stage put the library cassette into the datacorder, stdio side uppermost, and rewind the cassette. We start by making a copy of the header. Now type:

```
g,,stdio.h [ENTER]
```

Press PLAY on the tape recorder now and Press STOP when the cursor reappears. Now save the header by typing:

```
p1,9999,stdio.h [ENTER]
```

(to save to cassette - remember to put a blank one in).

Wait until the cursor reappears again.

Now we will repeat this process to save the library itself; but first we need to clear the editor's buffer: Type:

```
d1,9999 [ENTER]
```

Now read the cassette file:

```
g,,stdio.lib [ENTER]
```

Press PLAY and wait for the library to be loaded. Now save it by typing:

```
p1,9999,stdio.lib [ENTER]
```

and wait for the cursor again. After this turn the tape over and repeat the process for the remaining library files. That's all.

Keep the original tapes that we supply since you will need them for any upgrades. If you are making copies to your discs then, before following the instructions above, type:

```
|tape.in
|disc.out
```

1.5 Prices, Royalties, Publishing & Copying

We are often asked by people whether they would have to pay royalties if they were to sell programs compiled by our compilers. The answer is a resounding *no*! After all, that's what the compiler is for!

We are very happy for people to sell compiled programs and we wish you every success. We want to help you in every way we can, and are very happy to talk to you about products that you are writing. We may offer to publish the program for you, if we feel that it fits in with our range (and is of a high enough quality!). Or we may be able to suggest another publisher.

Any compiled programs will include our runtime routines, and we do require you to acknowledge our copyright of these on any programs that you publish. We'd also be very grateful if you mentioned that the program was written using our products.

What we don't like is *theft*. We sell our products at prices that are very low compared with similar products on other machines. The only way that we can do this and continue to produce more products in the future is by selling in volume. So please don't make copies for other people. The ugliest form of this is copying for money - piracy. We will do everything we can to stop this, and would appreciate being told of any pirate copies that you see.

Also please note that whilst we are happy to replace damaged cassettes, discs or manuals we cannot supply new copies of manuals to replace any that you lose so please take good care of them. (Again, compare this with what would happen with your computer or book).

Finally, I would like to apologise to the vast, honest majority of our customers for belabouring this point. *'Nuff said!*

I would now like to make our own acknowledgement to Leor Zolman at BD Software. This compiler was originally written entirely in BDS C (and a good C it is). It has been rewritten in assembler to achieve the size, but there are still a few C functions left in there. The runtime support for those functions is copyright of Leor Zolman. The functions themselves are copyright of **HiSoft**.

1.6 Example Programs

To demonstrate a few of the more basic techniques of C programming to the beginner, and to illustrate how to use the compiler, we have produced a few short example programs here which may be typed in, compiled and executed, or simply examined for reference. In addition, you may find some of the functions and techniques used may come in handy in more complex applications. For many more examples of C programming technique have a look in the library.

1.6.1 CP/M Hello world with Complications

This small program is just designed to show you how to type in and run a short program which uses the function library. First make sure that you have an installed copy of the editor **ED80**, then load it.

```
A>ed80 hello.c
```

Now type in the program finishing each line with [ENTER].

```
#include stdio.h

main()
{
char s[20];

s[0] = 'H';
s[1] = 0;
strcat(s, "ello, World!");
printf(s);
}

#include ?stdio.lib?
```

Save the program and exit from the editor with the command [CTRL]-K X. Now compile the program:

```
A>hc hello.c
HiSoft-C Compiler V1.3. Copyright (C) 1985
Line File
8 HELLO.C

MEMORY MAP Start End

Runtimes 0100 11FF
Code 1200 13A2
Initialisers 13A3 13A6
Fixed Data C2E7 C306

Smallest #data 0x13CF
```

Finally we get to run the program:

```
A>hello
Hello, World!
A>
```

1.6.2 CPC Hello world with Complications

This small program is just designed to show you how to type in and run a short program which uses the function library.

After loading the compiler type the INSERT command:

```
>i10,10 [ENTER]
```

Now type the program after the line numbers, finishing each line with [ENTER].

```
10 #include stdio.h
20
30 main()
40 {
50 char s[20];
60
70 s[0] = 'H';
80 s[1] = 0;
90 strcat(s, "ello, World!");
100 puts(s);
110 }
120
130 #include ?stdio.lib?
140
150 [ESC]
>
```

The [ESC] key brings you back to the editor prompt and so now we compile the program, by typing

```
>c [ENTER]

HiSoft C Compiler V1.2
Copyright c 1984 HiSoft

#include [ENTER]
```

The compiler starts to compile your program, listing it on the screen as it goes. The very first line is to include the library header, so put the library cassette in the recorder and press PLAY and another key when it asks. If you are using discs then the whole compilation will go through with no interruption. You should see

```
#include stdio.h
```

Press PLAY then any key:

```
Loading STDIO.H block 1

/*****************************/
/* HiSoft C                  */
/* Standard Function Library */
/* HEADER                    */
/*                           */
/* Copyright (C) 1984 HiSoft */
/* Last changed 15 Apr 1985  */
/*****************************/

#list-
/*****************************/
/* HiSoft C                  */
/* Standard Function Library */
/* End Header                */
/*****************************/

main()
{
char s[20];

s[0] = 'H';
s[1] = 0;
strcat(s, "ello, World!");
puts(s);
}

#include ?stdio.lib?
```

Press PLAY then any key:

```
Loading STDIO.LIB block 1

/*****************************/
/* HiSoft C                  */
/* Standard Function Library */
/* version 1.2               */
/* Copyright (C) 1984 HiSoft */
/* Last changed 20 Mar 1985  */
/*****************************/

Loading STDIO.LIB block 4

/*****************************/
/* HiSoft C                  */
/* Standard Function Library */
/* End                       */
/*****************************/
```

The cursor will reappears so that we can type in more of our program, but we have finished so we type `[CTRL]-Z` and then answer y to its question. The compiler asks us whether we want to run the program again, but we got back to the editor prompt by pressing `[ENTER]`. We should see: <> (double headed arrow).

```
Type y to run:y

Hello, World!

Type y to run:
>
```

1.6.3 The Sieve of Eratosthenes

This program uses the famous algorithm known as the *Sieve of Eratosthenes* to calculate all the prime numbers up to 16384. It is taken from BYTE magazine.

After you have run this program and timed it, add the word `static` just before `int` and time it again. The program is both smaller and faster now, as you can see . It is well worth using static variables wherever possible!

```
/* SIEVE BENCHMARK from June 84 BYTE */
/* compute primes using Sieve of Eratosthenes */

#define NTIMES 10 /* number of times to run sieve */
#define SIZE 8190 /* size of number array */
#define FALSE 0
#define TRUE 1

char flag[SIZE+1];

main()
{
  int i, j, k, count, prime;

  printf("%d iterations: ", NTIMES);
  for (i=1; i <= NTIMES; i++) {
    count = 0;
    for (j=0; j <= SIZE; j++)
      flag[j] = TRUE;
    for (j=0; j <= SIZE; j++) {
      if (flag[j] == TRUE) {
        prime = j + j + 3;
        for (k=j+prime; k <= SIZE; k += prime)
          flag[k] = FALSE; /* discard multiples */
        count++;
      }
    }
  }
  printf("%d primes.\n", count);
}
```

1.6.4 Numeric Conversion

This program is not particularly elegant but allows the user to enter a decimal number of up to five digits, which will subsequently be printed out in both hexadecimal (base 16) and binary (base 2). The readn function, which reads the decimal number in, is very unsophisticated and surprisingly easy to crash. If more than five digits are entered, or if one or more of the characters read in are not digits, then the results are quite unpredictable. Herein lies one of the golden rules of C - you can do literally anything, but any bugs are entirely *your* problem!

```
/* A program to convert a decimal number to hexadecimal and binary */

main()
{
  int n;
  char b[17];

  printf("\nGive me a number: ");
  n=readn();
  binary(n,b);
  printf("\nThis is %x in hex and %s in binary\n",n, b);
}

  int readn() /* reads in a decimal number of up to 5 characters */
{
  char s[5];
  int i,c,total;

  i=0;
  while ((c=getchar())!='\n')
    s[i++]=c;
  total=0;
  for (c=0;c<i;++c)
    total=total * 10 + s[c] - '0';
  return total;
}

binary(num,digits) /* converts a number to a binary string */
  int num;
  char digits[]; /* or char *digits; */
{
  int i,c;

  for (i=15;i>-1;--i)
  {
    c=num & (1 << i); /* progressively divide by 2 */
    digits[15-i] = c ? '1' : '0';
  }
  digits[16] = 0;
}
```

1.6.5 Memory Dump

This example is actually just a function rather than a complete program. It dumps out an area of memory in hexadecimal, which can be useful when debugging a program or trying to discover exactly what the operating system does. It dumps an area of 32 bytes after the starting address it is given.

```
dump(address)
char *address;
{
  static int i;

  for (i=0; i<32; ++i)
    printf(" %02x ", *address++);
  putchar('\n');
}
```


```
#direct+

dump(0xB000);
xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
xx xx xx xx xx xx xx xx xx xx xx xx
```

These examples are small pieces of C programming designed to impart the flavour of the language. Once you have studied them and know how they work the whole world is open to you. Remember that you can do just about anything in C!

1.7 Bibliography

1.7.1 Books about C

Most of the books in this section are of a tutorial nature, so we really recommend you to have a close look at several before buying because your preferences for style of writing are nearly as important as the actual contents. Also we have tried to concentrate on the less expensive books which are more in proportion with the cost of the compiler. There are some very good, but more expensive, books such as the first one in the list.

The C Programming Language
Brian Kernighan & Dennis Ritchie Prentice-Hall 1978
ISBN 0-13-110163-3

Contains the essential *C Reference Manual* and is a useful tutorial text. But it is expensive.

Learning to Program in C
Thomas Plum Prentice-Hall 1983
ISBN 0-13-527847-3

A good tutorial book which starts from first principles.

The Big Red Book of C
Kevin Sullivan Sigma Press 1983
ISBN 0-905104-68-4

A tutorial book which is very inexpensive.

The C Programming Tutor
Leon Wortman & Thomas Sidebottom Prentice-Hall 1984
ISBN 0-13-110024-6

At last Prentice-Hall produce a less expensive book about C!

C at a Glance
Adam Denning Chapman and Hall 1985
ISBN 0-412-27140-0

A tutorial book which uses **HiSoft C** in particular (along with DR C on the 8086) and also doesn't cost a lot. Very good value.

Practical C for the Home Micro
Mark Harrison Sigma Press 1985
ISBN 1850580359

Also uses **HiSoft C** in particular, is tutorial in nature and doesn't cost a lot! It is written in a different style to the previous book so there is probably at least one of them which will suit you. Very good value.

1.7.2 Books about CP/M

Mastering CP/M
Alan Miller Sybex 1983
ISBN 0-89588-068-7

Many people will already understand CP/M or will be able to read the manuals by Digital Research normally supplied with it. But some computers do not come with these manuals, and they can be rather daunting as a starting place. If you feel the need of more information then read this book.

1.7.3 Books about the Amstrad Computers

The Complete CPC464 Operating System
(Firmware Specification)
AMSOFT SOFT158

DDI-1 Firmware
(specification for Amstrad discs)
AMSOFT SOFT158A

HiSoft C

Fast Interactive K&R C Compiler

Chapter 2

The Editor

HiSoft

High Quality Software

2. The Editor

2.1 The CP/M Editor ED80

2.1.1 Introduction to ED80

The editor supplied with the CP/M version of **HiSoft C** is our full-screen editor **ED80** which is designed to be easy to use and to give the ability to edit programs quickly and efficiently. It is detailed fully in the separate manual contained within the binder.

As supplied, its commands match those of the very well known WordStar editor but you can tailor them to your particular preferences so it will be quickly learnt.

This version of **ED80** includes the ability to automatically restart the **HiSoft C** compiler when the editor was invoked as the result of a compilation error. For this reason you should not try to use any existing copy of **ED80** which you may already have. Instead use the installation program `ED80INST` supplied with this editor to make this version behave like your existing one (its quite painless if you use an E80 installation file).

If you do not have an existing copy of **ED80** then you will probably have to *install* it so that it knows about the control characters which are used by your particular display terminal. This is a simple procedure which is described in **ED80** manual.

2.1.2 How to Use the Compiler and ED80 Together

When the compiler finds an error in a compilation it prints an error message and waits for the user to press a key. If the user presses the `E` key then **ED80** is started with the cursor placed on the line where the error was detected. Pressing any key other than the `E` key will exit from the compiler and return to CP/M command level.

The user can make as many changes as are required after the editor has been automatically invoked by the compiler. All of **ED80**'s facilities can be used including the ability to finish editing one file and proceed immediately to another one - as shown in the example session below. This can be very useful where the error was caused by a fault in another file (eg a header file) or if it is simply required to look at another file before making the correction.

When all the corrections have been made the user just exits from **ED80** in the normal way. But in this case **ED80** will restart the original compilation instead of returning to CP/M.

The whole of this development cycle is shown in the diagram below, and illustrated by a worked example.

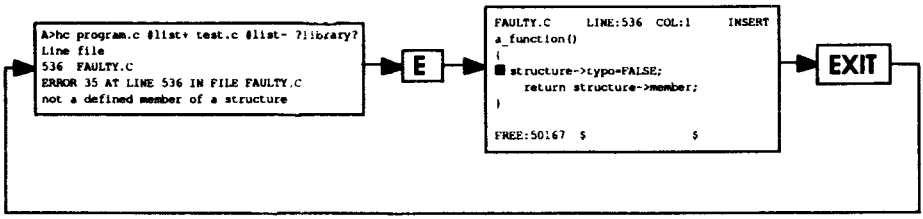


Diagram of Compiler/Editor Integration

2.1.3 A Worked Example of Compiler/Editor Usage

This worked example might better be called a contrived example, but it does show how the compiler and editor may be used together to correct errors very quickly. First we list the program file. The other two files will be seen later when we come to edit them.

```
/* A small program file called: PROG.C */

#include words.h

main()
{
  my_function();
}

#include ?my-lib.lib?

/* the end of the program file */
```

Having typed all three files in we type the command line to start the compiler. This can be quite complicated and as an example we have put a #data directive on the command line.

```
A>hc "#data 0x3000" prog.c
HiSoft-C Compiler V1.3. Copyright (C) 1985
Line File
my-lib.lib

ERROR 0 AT LINE 6 IN FILE MY-LIB.LIB
missing ')'
```

The compiler finds an error in the compilation and stops, waiting for us to press a key. We press the key in order to invoke the editor. The resulting screen looks something like the output on the next page:

```
MY-LIB.LIB    LINE:6   COL:1       INSERT

/* A library file called: MY-LIB.LIB */

my_function()
{
* printf("%s, %s!", HELLO, WORLD);
}

/* the end of the library file */

FREE:50351    $        $
```

The cursor is at the beginning of the line where the problem occurred but there doesn't seem to be anything that would cause the error. We suspect therefore that the problem must lie in the constants and which are defined in the header file. So we abandon editing of the library file and go to the header file by typing:

```
[CTRL]-K Q [SPACE] WORDS.H [ENTER]
```

You can refer to the **ED80** manual for full details of this command sequence. **ED80** now transfers us to the header file and the screen looks something like:

```
WORDS.H       LINE:1   COL:1       INSERT
*
/* A small header file called: WORDS.H */

#define HELLO "hello";
#define WORLD "world";

/* the end of the header file */

FREE:50351       $                    $
```

Now we can see what is actually a very common mistake: we have put semicolons at the end of the `#define` lines. These semicolons are substituted into the program wherever `HELLO` and `WORLD` are used and in this case spoiled the argument list of printf. All we need to do is delete the semicolons, which can be done by the following WordStar-like command sequence:

```
[CTRL]-X [CTRL]-X [CTRL]-X [CTRL]-Q D [DEL]
[CTRL]-X [CTRL]-G
```

Finally we just exit the editor, saving the altered header file.

```
[CTRL]-K X [ENTER]
```

This automatically restarts the compilation which proceeds swiftly to a satisfactory conclusion. Notice that the `#data 0x3000` command has taken effect because the original command line is used to control the compilation. We then immediately run the resulting `PROG.COM` file and are overjoyed by the startling message which it produces!

```
HiSoft-C Compiler V1.3. Copyright (C) 1985
Line File

MEMORY MAP Start End

Runtimes 0100 11FF
Code 1200 1244
Initialisers 1245 1248
Fixed Data 3000 3000

Smallest #data 1252
A>prog
hello, world!
A>
```

This ends the introduction to using the **HiSoft C** CP/M compiler with the interactive editor. Please consult the separate **ED80** manual for full details of the editor and how to use and configure it.

2.2 The Amstrad CPC Editor

2.2.1 Introduction to the CPC Editor

The editor supplied with the CPC **HiSoft C** is a line-based editor designed to be easy to use and to give the ability to edit programs quickly and efficiently. It has been tailored to the Amstrad computers and uses many of the same commands as the Locomotive BASIC editor so it will be quickly learnt.

The editor allows you to work on a program without using cassette (or disc) except when you want to save the program. This means that you can type in a program using the editor, then compile it and test it, returning to the editor to correct it and add more features until your whole program is finished. The editor keeps the text of your program in memory and allows you to add more, or change what is there, or delete some. The compiler can then read and compile the text (by #Include). The editor can also put the text onto cassette or disc (by p) and read it back (by g). The compiler is also able to read these files directly from tape or disc (by #include filename).

In order to go back to the editor when you are compiling or running a compiled program you should press the [ESC] key *twice*. There will be a short delay whilst the firmware resumes keyboard scanning and then you will see the editor prompt > and the cursor.

There is a special way of entering the editor after an ERROR message has been printed which makes it easier to correct mistakes. After the compiler has printed the error message it stops and waits for you to press a key. If you press the e key then the editor will perform an E command (edit line) on the last line compiled. You can make a correction immediately and carry on. You will go back to the editor in the normal way if you press any other key after an error message.

In response to the prompt you may enter a command line of the following format:

C N1,N2,S1,S2 followed by [ENTER].

C	is the command to be executed.
N1	is a number in the range 1 - 32767 inclusive.
N2	is a number in the range 1 - 32767 inclusive.
S1	is a string of characters with a maximum length of 20.
S2	is a string of characters with a maximum length of 20.

for example, to replace `fred` by `tom` in lines 1-50 of the file, you would type:.

```
F1,50,fred,tom[ENTER]
```

Very few of the editor commands need or expect all five parts of a command line and in some cases most would be inappropriate anyway.

To return to the C compiler you should type C followed by [ENTER]. You can also exit (permanently) to BASIC by typing [CTRL]-[SHIFT]-[ESC].

The comma is used to separate the various arguments (although this can be changed - see the S command). The editor remembers the previous numbers and strings that you entered and uses these former values, where applicable, if you do not specify a particular argument within the command line. The values of N1 and N2 are initially set to 10 and the strings are initially empty.

The editor uses line numbers in many of the commands. These line numbers are only used by the editor and are not stored in files on cassette or disc, nor are they used by the compiler. Lines are given numbers automatically by the editor when it reads them from tape or when you type them in with the I command. It is therefore best to regard these line numbers as notional, provided only for your benefit when using the editor.

If you enter an illegal command line such as `F-1,100,HELLO` then the line will be ignored and the polite message `Pardon?` displayed - you should then retype the line correctly e.g. `F1,100,HELLO`. This error message will also be displayed if the length of S2 exceeds 20; if the length of S1 is greater than 20 then any excess characters are ignored.

Command ; may be entered in upper or lower case.

The various commands available within the editor are described below - note that wherever an argument is enclosed by the symbols < > then that argument *must* be present for the command to proceed.

Text may be inserted into the textfile either by typing a line number, a space and then the required text or by use of the I command.

Text is typed in lines, and each line can have up to 80 characters in it. If you get to the end of a screen line then the screen will be scrolled up and you may continue typing on the next screen line.

Command lines, and new text lines, can all be edited as described below under the EDIT LINE command; for example the `[DEL]` key can be used to correct typing mistakes.

The text is kept in memory and so it is possible to fill up the memory. When this happens you will see the error message:

```
ERROR 60
LIMIT: no more memory
```

To save memory you can Put some of the text to cassette or disc, or you can use the compiler's #error feature to sacrifice the error messages, and then re-enter the editor:

```
>c
HISOFT-C Compiler V1.2
Copyright © 1984 HISOFT
#error
[ESC][ESC]
```

Once some text has been created there will inevitably be a need to edit some lines. Various commands are provided to enable lines to be amended, deleted, moved and renumbered. Most commands are executed immediately, but the E command selects a particular line for further special editing commands.

2.2.2 The Editor Commands

INSERT TEXT — Command: I m,n

The editor will display line numbers, and you can type in text (much like the AUTO command in BASIC). The line numbers start with **m** and go up in steps of **n**. Both the starting line number and the step size will default to 10 if you leave them out. You enter the required text after the displayed line number, and terminating the text line with [ENTER]. All the EDIT LINE commands described below are available when inserting text.

When you have typed as many lines as you want, press [ESC].

If you enter a line that already exists then the existing line will be shown on the screen and you will be able to edit it. If you wish to delete all the text from the existing line just hold down [CLR] until it has all disappeared. If the automatic incrementing of the line number produces a line number greater than 32767 then the Insert mode will exit automatically.

LIST TEXT ON THE DISPLAY — Command: L m,n

This lists the current text to the display from line number **m** to line number **n** inclusive. The default value for **m** is always 1 and the default value for **n** is always 32767 i.e. default values are not taken from previously entered arguments. To list the entire textfile simply use L without any arguments.

You can make the listing pause at any stage by pressing the [ESC] key once and resume it again by pressing any key other than [ESC] (eg press [ENTER]). Pressing the [ESC] key for a second time whilst the listing is paused will abort the listing immediately.

The listing may pause in the middle of drawing a character on the screen or even in the middle of scrolling the screen up (which may leave old text in the bottom right corner of the screen). This is normal.

WRITE TEXT TO PRINTER — Command: W m,n

The W command causes the section of text between lines **m** and **n** inclusive to be listed on the printer. If both **m** and **n** are omitted then the whole textfile will be printed. The [ESC] key can be used to pause or abort the listing just like the L command. If the W command is given and there is no printer then it will be necessary to use [ESC] [ESC] to abort the command.

VIEW DEFAULTS — Command: V

The V command displays the current delimiter and the current values of the two default line numbers and the default strings N1, N2, S1 and S2. This is useful before entering any command in which you are going to use default values, to check that these values are correct.

The command also displays the start and end address of the textfile in decimal.

SET DELIMITER — Command: S,,d

This command allows you to change the delimiter which is taken as separating the arguments in the command line. On entry to the editor the comma (,) is taken as the delimiter; this may be changed by the use of the S command to the first character of the specified string **d**. Remember that once you have defined a new delimiter it must be used (even within the S command) until another one is specified. Use V to discover the current delimiter. This command is mainly used when a find or substitute string contains a comma.

Note that the delimiter may not be a space.

RETURN TO THE C COMPILER — Command: C

When you have finished editing the text and want to compile it, just type C and press [ENTER].

RETURN TO BASIC — Command: (CTRL)-(SHIFT)-(ESC)

You can return to BASIC by pressing [CTRL]-[SHIFT]-[ESC]. This causes a full *Early Morning Start*, and all trace of the C compiler and your programs will be lost.

DELETE LINES — Command: D m,n

All lines from **m** to **n** inclusive are deleted from the textfile. If **m** > **n** then no action will be taken; this protects against careless mistakes. A single line m may be deleted by typing Dm [ENTER]. To prevent accidents, you cannot delete a line just by typing its number.

RENUMBER TEXT — Command: N m,n

Use of the N command causes the entire textfile to be renumbered with a first line number of **m** and in line number steps of **n**. Both **m** and **n** default to 10 and if the renumbering would cause any line number to exceed 32767 then the original numbering is retained.

EDIT LINE — Command: E n

Edit the line with line number **n**. If **n** does not exist then no action is taken; otherwise the line is displayed on the screen (with the line number). The cursor appears after the line number and the special editing commands can then be used to edit the line:

[CTRL]-[TAB]
The editor is automatically in insert mode when editing the line. This can be changed to overtype mode by pressing [CTRL]-[TAB] and back again by pressing it again (like Locomotive BASIC). There is no separate edit-command mode as with previous versions of the editor.

[ENTER]
terminate the line edit and keep all the changes made.

[ESC]
abort the line edit, discarding any changes.

→
step along to the next character on the line. You cannot step beyond the end of the line.

←
step back to the previous character on the line. You cannot step backwards beyond the beginning of the line.

[CTRL]-→
moves the cursor to the end of the line.

[CTRL]-←
moves the cursor to the start of the line.

[COPY]
Copy cursor editing is available and works in a similar way to the BASIC using [SHIFT] and cursor keys to move the read cursor around and [COPY] to copy the character under the read cursor down to the main write cursor.

[DEL]
deletes the character to the left of the cursor.

[CLR]
deletes the character under the cursor.

[CTRL]-F
find the next occurrence of the find string previously defined using the F command below. This sub-command will automatically end the edit on the current line (keeping the changes) if it does not find another occurrence of the find string in the current line. If an occurrence of the find string is detected in a subsequent line (within the previously specified line range) then the Edit mode will be entered for the line in which the string is found. Note that the text cursor is always positioned at the start of the found string.

[CTRL]-S
substitute the previously defined substitute string for the find string where the cursor is and then search for the next occurrence of the find string. This, together with the F sub-command above, is used to step through the textfile optionally replacing occurrences of the find string with the substitute string.

FIND STRING **Command: F m,n,f,s**

Text in the line range **m** to **n** is searched for an occurrence of the string **f** - the find string. If such an occurrence is found then the relevant text line is displayed and the Edit mode is entered - see above. You may then use commands within the Edit mode to search for subsequent occurrences of the string **f** within the defined line range or to substitute the string **s** (the substitute string) for the current occurrence of **f** and then search for the next occurrence of **f**; see the EDIT command above.

Note that the line range and the two strings may have been set up previously by any other command so that it may only be necessary to type F [ENTER].

Cassette and Disk Commands

These are fully compatible with the DDI-1 discs running AMSDOS and operation can be switched between tape and disc operation in the normal way using the external ROM commands from within the editor:

```
>|tape.in
>|disc.out          etc.
```

The [ESC] key can be used to abort a tape/disc operation.

Note that filenames must be CP/M compatible if they are to be used with a disc rather than the more general filenames possible on cassette. (ie up to eight character name optionally followed by a dot and up to three characters of extension, rather than arbitrary sixteen character filenames). Bad filenames cause output files to be sent to the screen and input files to be read from the keyboard.

PUT TEXT TO CASSETTE OR DISC **Command: P m,n,s**

Text in the line range **m** to **n** is put onto cassette or disc, using the filename specified by the string **s**. Remember that these arguments may have been set by a previous command. The line numbers are not put into the file. The normal system messages are given for cassette operation.

GET TEXT FROM CASSETTE OR DISC **Command: G,,s**

The cassette or disc is searched for a file with a filename of **s**; when found, it is loaded at the end of the current text.

Line numbers are attached to the lines of the file as they are read in. The line numbers go up in steps of 10. They start at 10 if there is no existing text and otherwise the text from tape is put at the end of the already resident text.

External Commands

External ROM commands can be used from the editor. This allows the disc commands to be used for example. The commands are used very easily in a similar way to BASIC, but more tidily. For technical reasons it is not possible to access RSX commands (see **Chapter 1**).

A vertical stroke | at the start of a line tells the editor that the name of an external command follows. The name is followed by any parameters, separated by spaces. Strings are just typed as a sequence of characters, although quote marks (") can be used if a string needs to include space characters or starts with a digit. Numeric parameters are typed as decimal numbers. Examples are:

```
>|basic
>|tape.in
>|dir *.c
>|user 3
>|ren newfile oldfile.ext
>|FUNny$cOmMand 1234 "a long parameter with spaces" third_parameter
```

2.2.3 An Example Session using the Editor

As a simple example of using the editor provided with the C compiler, we'll write a short program and attempt to compile it. We won't explain the program in too much detail here as it is not appropriate.

Load the C system. You will be presented with a > prompt. Type

```
i [ENTER]
```

to start inserting text at line 10 with a line increment of 10.

The line number is displayed on the screen along with a space and the cursor. Type in the lines below exactly as you see them, pressing the [ENTER] key at the end of each line. Blank lines are entered into the textfile by simply typing [ENTER] with no text.

```
10  /* An example editor session */
20
30  #define EOF -1;
40  #define FILE int
50
60  main()
70  {
80    int c,i,count,inword;
90
100   char s[20];
110
120   static FILE fp;
130
140   do
150   {
160     printf("Filename: ");
170
180     i = 0;
190     while ((c=getchar()) != '\n')
200     {
210       s[i++] = c;
220       if (i==19) break;
230     }
240
250     s[i] = '\0';
260
270     fp = fopen(s,"r");
280     if (fp==0)
290       printf("\nFile not found!\n");
300   }
310   while (fp==0)
320
330   count=inword=0;
340
350   while ((c=getc(fp)) != EOF)
360   {
370     if (i=isspace(c))
380       inword = 0;
390     else if (inword==0)
400     {
410       inword = 1;
420       ++count;
430     }
440   }
450   fclose(fp);
460
470   print("\nWords: %d\n",count);
480 }
```

Now, this program contains four errors, three of which could be described as *syntax errors*. See if you can spot them before going any further.

Once the program is entered, leave INSERT mode by pressing [ESC] and then go to the compiler by typing C and pressing [ENTER]. Now compile the program by typing #include and pressing [ENTER].

The first thing we see is that the compilation stops at the line containing the call to fopen with `error 32 - bad type combination`. Seasoned C programmers will immediately notice that the declaration of `fp` in line 120 is wrong - it should be a pointer, so we have to put an asterisk immediately in front of `fp`. To do this, re-enter the editor by hitting `[ENTER]` and entering the editor in the usual way. Now edit line 120 by typing:

```
E120 [ENTER]
```

The line appears on the screen along with its line number and the cursor. We want to step along the line and insert a `*` before `fp`, so repeatedly press → until the cursor is over the `f` of `fp`. Now type the asterisk, which appears in the line. Press `[ENTER]` and the altered line is entered as it is displayed on the screen. The cursor returns to the main editor and the machine awaits another command.

Go back to the compiler (`C` and `[ENTER]`) and try compiling the program again. It progresses quite smoothly but suddenly stops with `error 0 - missing ';'` right after displaying `count=`. It does this because the `while` immediately above this line is the terminating condition of a do ... while loop, and unlike simple while loops, there must be a semi-colon after the while condition. So we must edit this line.

Use the `E` command to edit the line as before but rather than stepping along the line, press `[CTRL]`-→. The cursor jumps straight to the end of the line. We can now insert characters at the end of the line, so type in a semi-colon and press `[ENTER]` to get back to the main editor.

Compile the program again. This time it stops at line 350 with `error 0 - missing ')'`. Why? Well, this is a very obscure bug that frequently catches programmers out, especially if they are used to other languages like Pascal. It stems from the `#define` statement on line 30.

#define allows *macros* to be used in C programs. In this context a macro is any amount of text from the first separator after the identifier up to the next end of line marker. Whenever the identifer is used in the program, it is substituted for the entire macro text. This means that whenever we use `EOF` it is replaced by the rest of the line - `-1;`. This is where the problem lies - the semi-colon after the `-1` is confusing the compiler. It shouldn't be there. So we'll have to edit it out.

Use `E30 [ENTER]` to edit line 30, and then press `[CTRL]`-→ as before to go to the end of the line. Now press `[DEL]` to delete the semicolon and press `[ENTER]` to leave the edit mode. If we look at the line we see that it no longer has the offending semi-colon, so we can attempt compilation yet again!

It works! Or at least it seems to. Press `[CTRL]-Z` to indicate end-of-source-file to the compiler, but *don't* execute the program yet. Up springs another error message - `error 27 - undefined variable(s)`. The offending identifier is `print` on line 470. It should of course be `printf`. Edit the line using `E470 [ENTER]` and insert the requisite `f` in the proper place using the techniques already discussed.

The final compilation now produces an error-free program - at least as far as the compiler is concerned - and it can be run by pressing `[CTRL]-Z` followed by `Y`. It counts the words in a given file.

The other features of the editor are all in their turn useful and it is a good idea to experiment with it as much as possible. As the editor is such a powerful program-building tool, it is vital that you can use it with familiarity and ease.

HiSoft C

Fast Interactive K&R C Compiler

Chapter 3

Language Summary

HiSoft

High Quality Software

3. LANGUAGE SUMMARY.

'Wouldst thou' - so the helmsman answered. -
'Learn the secret of the C?
Only those who brave its dangers
Comprehend its mystery!'

H W Longfellow

This chapter aims to provide a concise but complete reference to the C programming language as implemented by Hisoft C. It is intended that programmers familiar with other high-level languages such as Pascal will quickly grasp the concepts involved in C programming. If you are learning programming for the first time, or have any doubts, we suggest you look at some of the books recommended in the Bibliography in Chapter 1.

C provides the programmer with a structured and very concise way of representing data and algorithms. A C program is essentially a series of data and function definitions; the data structures describe the facts that we have and the results that we want; and the functions describe how to calculate the results from the facts.

This Language Summary is based around the syntax of the language (ie the rules for putting together a program), and it summarises each part of the language in turn. The approach is from the outside-in (or top down), but before starting it is useful to set out the basic building blocks or tokens from which the language is constructed.

3.1 Numbers and Characters.

There are several ways of writing numbers. The most straightforward is just as a decimal number. The largest number which can be handled is 65535.

For example 0 1 10 32767.

Numbers can also be written in two other bases. Octal numbers use 8 as their base and are written by putting a leading zero onto the number. Note that this means that decimal numbers must NOT start with a leading zero. Octal numbers are mainly of historical interest nowadays but they are used in strings in C. The first twenty octal numbers are written as:

0 01 02 03 04 05 06 07 010 011 012 013 014 015 016 017 020 021 022 023

The third base is hexadecimal, which uses sixteen as its base. These numbers are written by putting 0x or 0X in front of the number, and using letters A to F (or a to f) for the extra digits that are needed. Hexadecimal numbers are used a lot today because they are closely related to both binary and to a byte. A hexadecimal digit represents the same as four binary digits or bits, so two hex digits are a neat way of writing down the state of all the bits in a byte. The first twenty hex numbers are written as:

0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF 0x10 0x11 0x12 0x13

Single characters are written between single quote marks ('). They are really just a convenient way of writing down the numerical ASCII value which corresponds to them. For characters which can't be shown easily there are two special ways of writing known as escape sequences. One way is a set of special abbreviations for commonly used control characters.

name	abbreviation	value
newline	\n	10
horizontal tab	\t	9
backspace	\b	8
carriage return	\r	13
form feed	\f	12
backslash	\\	92
single quote	\'	39
double quote	\"	34

The other way is to write the OCTAL code value of the required character after a backslash.

control-Z	\32	26
null character	\0	0

One extra feature is that a backslash right at the end of a line is ignored together with the end of line character. These escape sequences are most frequently used in strings.

Some example characters are shown below. The last one is the character with code value 255 (or 0xFF) and illustrates a useful technique for getting at graphics characters on many micros.

```
'A' 'b' 'C' 'n' '\n' '\'' '\32' '\0' '\377'
```

3.2 Strings.

Strings are very important and quite complicated in terms of the place that they occupy in the C language, but writing them down is easy. A string is just a sequence of characters enclosed in double quotes. The compiler automatically adds a null character (ie with value 0) at the end of each string so that programs can check for the end of the string. This means that each string occupies one more character space than might be expected; and also means that the null character cannot be incorporated into a string and dealt with successfully by normal C functions. The string is stored as a sequence of bytes in ascending order, so a typical string, its storage, and appearance when printed might be:

```
"Hello\n\"John\""

72 101 108 108 111 10 34 74 111 106 110 34 0

Hello
"John"
```

3.3 Names or Identifiers.

Many things in C are given names, also called identifiers. Names follow a set of rules which are similar to those in many programming languages. A name must start with a letter or an underscore (A-Z, a-z, _) and then the rest of the name must consist of letters, digits or underscores (A-Z, a-z, _, 0-9). There can be an arbitrary number of characters in a name but only the first eight are taken account of in a C program. The rest are just there to make a more meaningful name. Upper and lower case letters are different so abc is a different name to aBc. C programmers normally use lower-case wherever possible and reserve upper case names for preprocessor names (qv). There is a convention that names beginning with an underscore are reserved for the compiler's own use. Examples of names allowed for normal use in programs are:

```
A       a       fred    Sinclair_ZX_Spectrum    Sinclair_QL
main    ____    _exit   EX_DE_HL
```

There are some points to make about some of these names:

Sinclair_ZX_Spectrum and Sinclair_QL are THE SAME NAME.

main is always the name of the main function in your program where execution of the program begins.

____ and _exit may well be system names. It would be unwise to use them in your programs.

EX_DE_HL would conventionally be the name of a constant (#define EX_DE_HL 0xEB).

3.4 Keywords.

Some names are kept for special uses in the language and they cannot be used for anything else. These names are called keywords or reserved words. The list of keywords is:

```
auto        break       case        cast
char        continue    default     do
double      else        entry       extern
float       for         goto        if
inline      int         long        register
return      short       sizeof      static
struct      switch      typedef     union
unsigned    while
```

3.5 Formal Syntax Summary.

After looking at the tokens which are used to write C programs we will now look at the overall plan of a program and begin our examination of the parts which make it up.

We start with a syntax summary based on that in Kernighan & Ritchie so that it is easy to compare, but it reflects the syntax accepted by the Hisoft-C compiler and has been re-ordered so that the overall structure of a program is clearer.

The aim of the summary is to describe what is permitted in Hisoft-C programs, and this is done by splitting a program up into its components and then describing what makes up each part. A special notation is used to help with this task, but the notation is very simple and easy to understand. The name of each component is given, followed by a list of possible alternative ways to write one of these parts. Each alternative is on a separate line and if there are several parts making up an alternative then they are just listed after one another on the line.

If an item has "_OPT" attached to it then it is an optional item. If an item has "_LIST" attached to it then it may be repeated an arbitrary number of times, but must be there at least once. If an item has both _LIST and _OPT attached, then it may be missing altogether or repeated an arbitrary number of times.

For example, if we look at the very first definition for a "program", we can see that a "program" is just a sequence of "external_definition"s. The next definition tells us that an "external_definition" is either a "function_definition" or a "data_definition" and so on.

3.5.1 Programs and External Definitions.

```
program:
     external_definition_LIST

external_definition:
     function_definition
     data_definition



function_definition:
     type_specifier_OPT   function_declarator   function_body

function_declarator:
     declarator  (  )
     declarator  (  parameter_list  )

parameter_list:
     identifier
     identifier  ,  parameter_list

function_body:
     type_declaration_LIST          {          declaration_LIST_OPT          statement_LIST          }



data_defintion:
     type_specifier_OPT          init_declarator_list          ;
     extern          type_specifier_OPT          init_declarator_list          ;
     static          type_specifier_OPT          init_declarator_list          ;
```

3.5.2 Declarations.

```
init_declarator_list:
     init_declarator
     init_declarator  ,  init_declarator_list

init_declarator:
     declarator
     declarator  initializer

declaration:
     decl_specifiers  declarator_list  ;

type_declaration:
     type_specifier  declarator_list  ;

declarator_list:
     declarator
     declarator  ,  declarator_list

declarator:
     identifier
     (  declarator  )
     *  declarator
     declarator  (  )
     declarator  [  ]
     declarator  [  constant_expression  ]

decl_specifiers:
     sc_specifier
     type_specifier
     sc_specifier  type_specifier

sc_specifier:
     auto
     static
     extern
     register
     typedef

type_specifier:
     char
     short
     int
     long
     unsigned
     short  int
     long  int
     unsigned  int
     struct_or_union_specifier
     typedef_name
```

```
typedef_name:
     identifier

struct_or_union_specifier:
     struct { struct_decl_list }
     struct tag_name { struct_decl_list }
     struct tag_name
     union { struct_decl_list }
     union tag_name { struct_decl_list }
     union tag_name

tag_name:
     identifier

struct_decl_list:
     type_declaration_LIST

initializer:
     = co stant_expression
     = { initializer_list }
     = { initializer_list , }

nitializer_list:
     constant_expression
     initializer_list , initializer_list
     { initializer_list }
```

3.5.3 Statements.

```
compound_statement:
     { statement_LIST_OPT }

tatement:
     compound_statement
     expression ;
     if ( expression ) statement
     if ( expression ) statement else statement
     while ( expression ) statement
     do statement while ( expression ) ;
     for ( expression_OPT ; expression_OPT ; expression_OPT ) statement
     switch ( expression ) statement
     case constant_expression : statement
     default : statement
     break ;
     continue ;
     return ;
     return expression ;
     goto label_name ;
     label_name : statement
```

```
    inline ( constant_expression_list ) ;
    ;

label_name:
    identifier
```

3.5.4 Expressions.

```
expression_list:
    expression
    expression , expression_list

expression:
    primary
    & lvalue
    - expression
    ! expression
    ~ expression
    ++ lvalue
    -- lvalue
    lvalue ++
    lvalue --
    sizeof ( type_specifier )
    cast ( type_specifier ) expression
    expression binary_operator expression
    expression ? expression : expression
    lvalue assign_operator expression

primary:
    primary_lvalue
    constant
    string
    ( expression )
    primary ( )
    primary ( expression_list )

primary_ivalue:
    identifier
    primary [ expression ]
    lvalue . identifier
    primary -> identifier
    ( lvalue )

lvalue:
    primary_lvalue
    * expression

binary_operator:
    *       /       %
```

```
      +       -
      >>      <<
      <       >       <=      >=
      ==      !=
      &
      ^
      |
      &&
      ||

assign_operator:
      =       +=      -=      *=      /=      %=      >>=     <<=     &=      ^=      |=

constant_expression_list:
      constant_expression
      constant_expression   ,   constant_expression_list

constant_expression:
      expression
```

3.5.5 Preprocessor Lines.

```
      #define   identifier   token_string
      #include
      #include   filename
      #include  "filename"
      #include  <filename>
      #include  ?filename?
      #direct+
      #direct-
      #list+
      #list-
      #error
```

3.5.6 Notes.

We have broken our rules in the definition of "binary_operator" and "assign_operator" in order to achieve a neater appearance. Each operator is just one of those shown, not a whole line of them!

Constant expressions are only brought in as a syntactic term to emphasise those places where they are needed. They are identical to ordinary expressions except that their value can be (and is) calculated when the compilation is being done.

3.6 Programs.

A C program is just a collection of functions and interspersed data. Functions are similar to the procedures, subroutines etc of other languages. A function has a name and possibly some parameters (known together as the function heading) and some statements to be executed (known as the function body). All the work of a C program is done by these statements.

Among the functions there must be one called "main", and when a C program is run, all that happens is that the "main" function is called.

The smallest possible C program is:

```
main(){}
```

which does absolutely nothing. The smallest program which is normally presented as an example is:

```
main()
{
  printf("Hello, world.");
}
```

which simply displays a message on the screen.

3.7 Layout.

Layout of a C program is not important to the compiler, as you can see from the smallest possible program above, but most people have strong views and prefer a neat indented layout.

A C program is made up of names and constants together with a number of other special symbols usually including a liberal sprinkling of { and }. Collectively, these items are known as tokens.

Each of these symbols can usually be placed right next to another one, but sometimes they have to be separated where confusion might occur, and the program is likely to be easier on the eye if symbols are separated.

The characters used to separate tokens are known as white space. They are: the space character, the newline, form feed, tab.

3.8 Comments.

Comments can also be used anywhere that white space can go. They are used to explain what a program is doing, and to give it a title etc. A comment starts with /* and finishes at the next */. There can be no spaces between the / and the *. Any characters may appear in a comment. Comments DO NOT nest.

3.9 Functions.

syntax

```
function_definition:
     type_specifier_OPT   function_declarator   function_body

function_declarator:
     declarator  (  )
     declarator  (  parameter_list  )

parameter_list:
     identifier
     identifier  ,  parameter_list

function_body:
     type_declaration_LIST          {          declaration_LIST_OPT          statement_LIST          }
```

example

```
unsigned  strlen(string)
  char string[];
{
  int length;

  for(length=0; string[length]; length=length+1) ;
  return length;
}
```

Functions are a very important part of C programs and indeed it can be said that they ARE the entire program. In the example programs above we saw a function definition. We told the compiler what we meant by the "main" function.

In the "Hello, world." example we also saw a function call. We told the compiler that we wanted to perform a function. In this case the function was a built in library function named "printf", which displays a message on the screen. All functions are called in the same way by writing their name followed by a pair of parentheses. In between the parentheses are any arguments which are to be passed to the function.

In C a function always has a result which can be used in an expression. By ignoring the result of a function, we have a procedure. The result of a function has a data type just like a variable, although there are some types which a function cannot return. These restrictions are discussed below in the Data Type section.

Now let's move on to consider how to write functions in detail. When a function is defined, the given identifier (the name of the function) is followed by a pair of parentheses enclosing a "parameter list", which states the number and variety of the arguments a function expects to be passed. The fact that a function has the parentheses and parameter list in its declaration marks it from a variable declaration. The parentheses, '(' and ')', are always there, even for those functions which do not expect parameters.

Note that there is NO semi-colon following the parameter list, as there may be more to come in the declaration - the parameters themselves need to be declared in terms of type. You can see this in the example, where the parameter "string" is shown as an array of characters.

Immediately following a function declaration of this form is the function body. This is enclosed in braces, '{' and '}', and consists of some optional variable declarations followed by any number of statements formed from C keywords and expressions. It is not possible to define functions within other functions.

Parameters to functions are always passed by value, not by reference. The only manipulations possible upon functions are to call them and to take their addresses. A function declaration with no storage class and no data type specified returns an integer.

There is an additional feature in Hisoft C function definitions which provides a way of defining functions which take a variable number of actual arguments (these are called "variadic" functions). The reserved word "auto" can follow the function-declarator before the function-body (eg between the "strlen(...)" and the "char string[]; {...}"). The effect is to cause the compiler to place the number of bytes of actual arguments as an additional argument after the rest. The function can access this argument and use it to work out how many other arguments there were. The standard library functions "min" and "max" make use of this facility and the source code in the library HEADER file "stdio.h" provides an example of how to write such a function.

We will move on now to consider statements and expressions, leaving the discussion of declarations until later. This is because the way that declarations are written is unique to C and depends heavily on how expressions are written. We will consider the "pre-processor" at the end.

3.10 Statements.

A function is comprised of individual statements, all of which are executed in textual sequence unless a "control flow" statement is used to specifically alter this order. We will look at each individual type of statement in turn.

3.11 Compound statements or blocks.

syntax

```
{ statement_LIST_OPT }
```

example

```
{ putc(character, file_pointer); putchar(character); }
```

In every instance where the syntax dictates one statement, we can have as many statements as we see fit by creating a compound statement. A compound statement is formed by placing an arbitrary number of statements inside braces, '{' and '}'. The language treats this as one statement, but executes each statement inside the block as individual statements. A compound statement is enclosed in braces and is not terminated by a semi-colon.

3.12 Expression Statements.

syntax

```
expression ;
```

examples

```
++i;

a = 5;

printf("hello");
```

Any expression can be turned into a statement by putting a semicolon after it. Expression statements are very common and in fact most calculation in C programs is done by expression statements. The expression is evaluated and then the value is discarded; which means that the only effects of the expression are its side-effects! Typically these are a new value assigned to a variable, or some input or output.

3.13 The if or conditional statement.

syntax

```
if ( expression ) statement

if ( expression ) statement1 else statement2
```

examples

```
if (units == 10) { units = 0; ++tens; }

if (i > 0)       printf("positive");
else if (i < 0) printf("negative");
else             printf("zero");
```

This causes the expression to be evaluated. If it has a TRUE (non-zero) value, the first statement is executed. If it is FALSE (zero), then the first form does nothing, while the second executes the second statement. In the case of numerous nested if-else clauses, any ambiguity concerning the attachment of the "else" is resolved by connecting it with the prior "if".

3.14 The while statement.

syntax

```
while ( expression ) statement
```

example

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

This evaluates the expression, and if it is TRUE (non-zero), executes the statement. It then repeats the process, stopping only when the expression becomes FALSE (zero). As the expression which determines the condition is evaluated before the statement is executed, the zero-case is catered for. Thus,

```
while (0) statement
```

will cause the statement to be executed no times, and

```
while (1) statement
```

will cause the statement to be executed for ever. There are ways out of this (see the break statement).

3.15 The do statement.

syntax

```
do statement while ( expression ) ;
```

example

```
do *string1 = *string2++;
while (*string1++);
```

This causes the statement to be executed, and then the expression is evaluated. If it is TRUE (non-zero), the process repeats itself, while if it is FALSE (zero), the statement terminates. Note that the semi-colon following the expression is necessary. The do statement always evaluates its expression after execution of the controlled statement.

3.16 The for statement.

syntax

for (expression_OPT ; expression_OPT ; expression_OPT) statement

examples

```
for (s = string; c=*s; ++s) putchar(c);

for (x=0; s[x]; ) putchar(s[x++]);

for (;;) if ( ! *s++) break;
```

The first expression is evaluated once only to initialise the loop. It then evaluates the second expression, and if this is TRUE (non-zero), executes the statement followed by the third expression before reiterating to evaluate the second expression again. The for statement terminates if the second expression is FALSE (zero); neither the controlled statement or the third expression is executed again.

If the first expression is treated as an initialisation statement, the second as an iteration test and the third as a loop increment, the for statement can be compared easily with other languages' FOR loops:

```
for (x=0; x<10; ++x) statement
```

is equivalent to BASIC's:

```
FOR x = 0 TO 9 STEP 1
statement
NEXT x
```

Any or all of the expressions in the statement may be left out; if the second (testing) expression is left out, it is equivalent to a "while (TRUE)" loop. The semi-colons must be retained even if the expressions are left out.

3.17 The switch statement, case labels, and the default label.

syntax

```
switch ( expression ) statement

case constant_expression : statement

default : statement
```

example

```
switch (day)
  {
    case 0: printf("Monday - back to the grindstone");
            break;
    case 4: printf("Friday - tomorrow's the weekend!");
            break;
    case 5:
    case 6: printf("weekend - don't wake me early");
            break;
    case 1:  case 2:  case 3:
            printf("boring days");
            break;
   default: printf("%d is not a day of the week!!!", day);
  }
```

The statement is almost always a compound statement, so:

```
switch (expression)
  {
    statement_LIST
  }
```

is probably a more useful way to look at the syntax. The switch causes control to be passed to one of several statements inside the block depending on the value of the expression, which must evaluate to an integer. The switch chooses which statement to pass control to by comparing the result of the expression with a "case label" inside the block.

Any statement inside the block may be given a case label, but each case label must have a different value constant expression. A statement may be given more than one case label, in which case numerous values of the switch expression cause control to be transferred to that statement. Each case label is simply a destination for control; it does not imply the Pascal-like "go there, do that statetment, and end". It is a multi-way GOTO contruct. If, after the control has been transferred, other case-labelled statements are met, no extra or special action is taken. This situation can be controlled using the "break" statement (see below).

At most one statement may be labelled with a special case label called the default label. This causes the control to proceed to this labelled statement if the value of the switch expression matches no other case label. If the default label is not present, and the switch expression does match any of the case labels, control is transferred to the statement following the switch statement.

3.18 The break statement.

syntax

break ;

example

see switch statement above

This causes the nearest (smallest textually enclosing) while, do, for or switch statement to be terminated and control is passed to the statement immediately following the terminated statement.

3.19 The continue statement.

syntax

continue ;

example

```
tens = units = 0;
for (i=0; i<100; ++i)
  {
    printf("\n%d %d", tens, units);
    ++units;
    if (units <10) continue;
    units = 0;
    ++tens;
  }
```

The continue statement causes control inside the nearest (smallest textually enclosing) while, do or for statement to pass to the loop iteration part of the statement, causing the iteration test to be performed. In all cases, this means that continue has the same effect as a jump to the very end of the controlled statement. For instance, the example above is equivalent to:

```
tens = units = i = 0;
while (i<100)
  {
    printf("\n%d %d", tens, units);
    ++units;
    if (units <10) goto contin;
    units = 0;
    ++tens;
contin:
    ++i;
  }
```

3.20 The return statement.

syntax

```
return ;

return expression ;
```

example

```
return end - start;
```

This statement causes the termination of a function. The first form, with no following expression, is equivalent to "falling out" of a function by coming to its textual end; and the result of the function is not defined. When return is followed by an expression, this expression is used as the result of the function.

3.21 Labelled statements and goto.

syntax

```
goto label_name ;

label_name : statement
```

example

```
label:
switch(getchar())
  {
    case 'y': puts("YES"); break;
    case 'n': puts("NO");  break;
    default : goto label;
  }
```

Any statement inside a function may be labelled by preceding it with an identifier and a colon. This label serves one purpose only - as a destination for a goto statement. The label is in scope anywhere inside the function in which it was defined. The label must apply to a statement, and the null statement (;) can be used for this purpose if required.

The goto statement causes control to pass to the statement labelled by the given identifier, which must be defined in the same function.

In a language like C it is almost always possible to find a clearer, more efficient way to rewrite a function without using goto statements. Except on rare special occasions they are an indication of careless programming or a poor algorithm.

3.22 The inline statement.

syntax

```
inline ( constant_expression_list ) ;
```

examples

```
/* to jump to location 0: */

inline(0xC3,0,0);   /* 0xC3 is the Z80 'JP' instruction */


  /* to call location 0x1601 with 3 in the A-register */
  /* (open stream 3 on a Spectrum):                   */

#define  CHAN_OPEN  0x1601
#define  LD_A_with  0x3E
#define  CALL       0xCD
inline(
  LD_A_with, 3,
  CALL,      CHAN_OPEN
  );


/* to store an input character in a global variable "c": */

#define LD_HL_into  0x22
inline(
  CALL,       getchar,
  LD_HL_into, &c
  );
```

A new type of statement (which looks like a function call) is used:

```
inline ( k1, k2, k3, ... ) ;
```

k1, k2 etc is a list of constant expressions which will be put into the output code. An expression which has a value in the range 0 to 255 inclusive will cause a single byte to be generated and any other number will cause two bytes to be generated. Any constant expression may be used. Many other examples of the use of inline can be found in the machine specific library "BASIC.LIB".

This statement introduces native machine code directly into a C program. Each constant_expression in the parenthesised list following the "inline" is entered into the compiled program at exactly that point, and each will be executed as machine instructions in the sequence in which they appear in the list. There can be an arbitrary number of constant expressions in the list, but there must be at least one.

Be careful when using this facility. Consult the "Low-Level" section in this manual to find out which registers need saving etc.

3.23 The null statement.

syntax

```
;
```

examples

```
while (getchar() == ' ')  ;

if (some_condition)
  {
    if (another_condition) goto label;
    do_something();
label:  ;
  }
```

The extreme case of a statement is the null statement which is perfectly valid, and causes nothing to be done. It is often useful in delay loops or control flow statements which do all their work in the controlling expression. It is also useful sometimes as the destination statement of a goto.

3.24 Expressions.

Expressions are combinations of constants, variables, operators and function calls which together have a single value. While expressions in C share the common aspects of other languages' expressions, the variety of operators available is far greater.

To understand how to form valid expressions, we need to consider the operators available. The operators are divided into three approximate groups, according to how they are put together with their operands. The "primary" operators form the first group: these each have a special syntax or pattern to write them. The second group comprises the "unary" operators which take a single operand, usually after the operator (as in -27). The third group contains the "binary" operators which are all infix operators, sitting between their two operands (as in 2 + 3). There is actually a single "ternary" operator which takes three operands (? :); and this is best regarded as part of the binary group for most purposes.

Each operator has a 'precedence' and an 'associativity' which determine the meaning of an expression involving several operators. Operators of higher precedence bind more tightly than those of lesser precedence; and operators of equal precedence usually associate (form chains) from left-to-right. These rules are designed to match our previous experience with arithmetic. For example:

```
1 - 2 - 4 * 5
```

evaluates to -2i just like:

```
(1 - 2) - (4 * 5)
```

and not like some of the other possibilites:

```
((1 - 2) - 4) * 5  /* strict left-to-right with no precedence */
1 - (2 - (4 * 5))  /* strict right-to-left with no precedence */
1 - (2 - (4 * 5))  /* right-to-left with * higher precedence  */
(1 - (2 - 4)) * 5  /* right-to-left with * lower precedence   */
```

There are a few exceptions to the left-to-right associativity rule. These are the assignment operators, and the conditional operator (?:). The precedence and associativity of each operator is given in the table below.

Precedence and associativity can be over-ridden with parentheses, and the golden rule is "if in doubt, use parentheses".

It should be noted that the order of evaluation of an expression (the sequence in which the value of the parts of an expression are calculated) is NOT specified by the language; and it certainly varies considerably from one compiler to another. This means that you cannot ever trust an expression like:

```
n = getchar() + (getchar() << 8);
```

and should rewrite it in two separate statements:

```
n  = getchar();
n += getchar() << 8;
```

The &&, ||, and ? : operators are about the only ones which make guarantees about order of evaluation.

3.25 Table of Operator Precedence and Types.

Operator & Precedence	Needs Lvalue	Types of Arguments Left	Right	Result Type
(exp) 16		any		unchanged
func() 15		func returning ...		...
[] 15		ptr to ...	integral	...
. 15	yes	structure	member	type of member
-> 15		ptr	member	type of member
* 14		ptr to ...		...
& 14	yes	...		ptr to ...
- 14		arithmetic		arithmetic
! 14		arithmetic		int
		ptr		int
~ 14		integral		integral
++ 14	yes	...		...
-- 14	yes	...		...
sizeof 14		type_specifier		int
cast 14		type_specifier		type specified
* 13		arithmetic	arithmetic	arithmetic
/ 13		arithmetic	arithmetic	arithmetic
% 13		integral	integral	integral
+ 12		arithmetic	arithmetic	arithmetic
		ptr to ...	integral	ptr to ...
		integral	ptr to ...	ptr to ...
- 12		arithmetic	arithmetic	arithmetic
		ptr to ...	integral	ptr to ...
		ptr to ...	ptr to ...	int
>> 11		integral(L)	integral	integral(L)
<< 11		integral(L)	integral	integral(L)
< 10		arithmetic	arithmetic	int
>		ptr to ...	ptr to ...	int
<=				
>=				
== 9		arithmetic	arithmetic	int
!=		ptr to ...	ptr to ...	int
		ptr to ...	integral	int
& 8		integral	integral	integral
^ 7		integral	integral	integral
| 6		integral	integral	integral
&& 5		arith-ptr	arith-ptr	int
|| 4		arith-ptr	arith-ptr	int
? : 3		arith-ptr	arith-ptr	arith-ptr *grp-r-l
= 2	yes	arith-ptr	arith-ptr	arith-ptr *grp-r-l
op= 2	yes	as for = but also constrained by the operator (op)		

NOTES

arith-ptr	means	arithmetic or pointer type
grp-r-l	means	groups right to left

3.26 Identifiers, Constants, and Strings.

syntax

identifier

constant

string

examples

```
a_very_long_name
i
42
'a'
"Hello, World"
```

The very simplest expressions consist of just one item: either the name of a variable, or a numeric constant, or a string. The value is just the value of the variable, or of the constant (considered as an int). The value of a string is a constant which points to it (of type pointer to char).

3.27 lvalues and array names.

The difference between lvalues and primaries can be very confusing, and the question often arises as to why both exist. The answer lies in the fact that lvalues are basically addresses of some area of memory such as a variable; which allows extra kinds of expressions. For instance, an lvalue is needed as the destination of an assignment.

Most identifiers are lvalues, but in particular array names are not. This is because of the way an array name is used to refer to the array. It is treated as though it were declared as a pointer to the element type, but it is a constant pointing always at the first element of the array. The only operations which can be performed upon an array name are ones in which the notional value of the name is not changed. For instance (array_name + 1) is a perfectly valid expression, yielding a pointer to the second element in the array and we can use it to assign a new value to the element:

```
*(array_name + 1) = 4;
```

But an assignment to the array name, to make IT point at the second element does not make sense and will be rejected by the compiler.

```
array_name = (array_name + 1);
ERROR - 34 - need an lvalue
```

Because of this property no address operator (&) is needed before an array name when a pointer to the array is required. This is also true of the names of functions, but contrasts with normal integral variables and structures. For example:

```
qsort(array, 100, 5, strcmp);  /* sort array using strcmp function */
blt(&structure, array, 128);  /* move contents of array into structure */
scanf(" %d %d", &integer, array); /* reads into first element */
```

3.28 Parenthesised Expressions.

syntax

```
( expression )
```

example

```
(2 + 5) * 7
```

Any expression can be enclosed in parentheses in order to alter precedence, as in normal arithmetic. This does not alter the type or value of the enclosed expression.

3.29 Function Call.

syntax

```
primary ( )
primary ( expression_list )
```

examples

```
printf("Hello")
c = getchar()
scanf(" %d", &n)
(*cmp_func)(a, b)
```

The function call passes the values of any given arguments to the named function, and returns a result of the type given in the function's definition. The values of the arguments passed are passed by value and cannot be changed; but variables outside the function can be changed by passing a pointer to them. A function call is not an lvalue and cannot appear on the left-hand-side of an assignment.

Neither the type nor number of arguments are checked against those specified when the function was defined. This insecurity can be useful on occasion, but is also a large source of errors. The order of evaluation of arguments is not specified by the language. Recursive calls are permitted to any function.

3.30 Subscripts.

syntax

```
primary [ expression ]
```

examples

```
i = array[5]

array[subscript][3] = 27

pointer[index]

2 [location_vector]
```

Subscripts are usually applied to the address of an array, which may be the array name or a pointer to the array. The specified index is multiplied by the size of the base element of the array and added to the base address. In fact the subscript operator is more general than found in most languages and is identical in all respects to writing:

```
* ( ( primary ) + ( expression ) )
```

This flexibility allows all the normal uses of subscripts, and the particularly important extra ability to treat a pointer as though it were the base of an array. Further uses are boundless! Please note that as a consequence of this flexibility there is NO BOUND CHECKING of arrays.

3.31 structure member operator.

syntax

```
lvalue . identifier
```

example

```
file_control_block . record_number
```

This is the structure / union analogy of array subscripting. The result is an lvalue corresponding to the specified member of the named structure / union. The identifier on the right of the dot must be the name of a member of a structure. Normally the lvalue to the left of the dot will refer to a structure of the appropriate type, but ANY lvalue is permitted.

3.32 structure pointer operator.

syntax

```
primary -> identifier
```

example

```
file_pointer -> record_number
```

The structure pointer operator yields an lvalue corresponding to the specified member of the pointed-to structure. The identifier on the right of the arrow must be the name of a member of a structure. Normally the pointer to the left of the arrow will refer to a structure of the appropriate type, but ANY pointer is permitted. The operation is equivalent to writing:

```
( * primary ) . identifier
```

3.33 indirection operator.

syntax

```
* expression
```

examples

```
*pointer + 5

*(pointer + 5)

*s++ = getchar()

*strchr('A', string) = 'B'
```

This operator is applied to a pointer and yields an lvalue corresponding to the item pointed to.

3.34 address operator.

syntax

```
& lvalue
```

examples

```
&variable

scanf(" %d", &integer_variable)

* & * & * & * & do_nothing_just_for_amusement
```

The address operator is the opposite of the indirection operator, in that it yields a pointer to an lvalue. The binary value of this pointer is actually the machine address of the lvalue. As non-lvalues do not have addresses, it is an error to apply the operator to non-lvalues.

3.35 unary minus operator.

syntax

```
- expression
```

examples

```
-5

-(i+j)
```

This operator reverses the sign of an expression, by taking its twos complement.

3.36 logical NOT operator.

syntax

```
! expression
```

examples

```
! flag

! pointer
```

The result of the operator is 1 if its operand has the value 0, and 0 otherwise. So it reverses the boolean state of an expression, such that a FALSE (zero) value will be switched to a TRUE (non-zero) value and vice-versa.

3.37 bitwise NOT operator.

syntax

```
~ expression
```

example

```
~inp() & bitmask
```

The bitwise NOT operator takes an expression and regards it as a binary bit-pattern. It 'flips' the state of all the bits in the expression value, yielding the one's complement. So the bitwise NOT of 0x15AF is 0xEA50 (in binary the two numbers are 0001 0101 1010 1111 and 1110 1010 0101 0000).

3.38 the increment and decrement operators.

syntax

```
++ lvalue
-- lvalue
lvalue ++
lvalue --
```

examples

```
++i

*string_pointer++

(structure_pointer++)->member
```

The ++ operator increments an lvalue while the -- operator decrements it. As it implicitly includes an assignment, the expression that it is applied to must be an lvalue.

The ++ and -- operators can be used either as a prefix or as a postfix. If the operator is written before the lvalue, then the lvalue is incremented and the resulting new value is the value of the expression; while if it follows the lvalue, the value is incremented but its original value is the value of the expression.

Its action is to add 1 to (or subtract 1 from) the lvalue, but this '1' is better regarded as 'sizeof (type_of_lvalue)', as it actually increments in type-sized units. For example, if ptr points to the nth element of an array, then

```
*++ptr
```

refers to the (n+1)th element and results in ptr pointing to this element, while

```
*ptr++
```

refers to the nth element and results in ptr pointing to the (n+1)th array element. If the array is an array of integers (which each occupy two bytes) then the binary value of ptr will increase by TWO each time it is incremented. The decrement operators are analogous to the increment operators but decrease the value of the lvalue by 1 unit.

3.39 the sizeof operator.

syntax

sizeof (type_specifier)

examples

```
sizeof(int)

struct fcb { int field1; struct fcb *field2; } ;
sizeof(struct fcb)

typedef char *char_pointer;
sizeof(char_pointer)
```

The sizeof operator yields the size of the specified type in bytes.

3.40 the type cast operator.

syntax

cast (type_specifier) expression

example

```
cast(char_pointer) 0xABCD
```

This operator performs 'type coercion' upon an expression; forcing its type to be that specified. No changes take place in the binary value of the expression.

3.41 multiplication operator.

syntax

expression * expression

example

```
3 * 5
```

This operator multiplies the first expression by the second, and returns the product as its result. Arithmetic overflow is not considered an error in C, so a value exceeding the data type range of the result may be 'wrapped around'.

3.42 division operator.

syntax

expression / expression

examples

```
 7 / 2  ==  3
-7 / 2  == -3
 7 /-2  == -3
-7 /-2  ==  3
```

Divides the first expression by the second using integer division, so 7/2 returns 3, not 3.5. The remainder has the same sign as the dividend, so truncation is towards zero. Division by zero does not cause a runtime error (in fact it returns the dividend as its answer, but don't rely on this!).

3.43 modulus or remainder operator.

syntax

expression % expression

examples

```
 7 % 2  ==  1
-7 % 2  == -1
 7 %-2  ==  1
-7 %-2  == -1
```

This operator returns the remainder following an integer division of the first expression by the second. The result has the same sign as the dividend. It is always true that (a/b)*b + a%b == a.

3.44 addition operator.

syntax

expression + expression

examples

```
2 + -5
array_pointer + 1
-42 + stringend_pointer
```

This operator yields the sum of the two expressions. For normal integral operands, it behaves as would be expected. But a pointer and an integer can be added, and in this case the integer is multiplied by the size of the object pointed to before the addition. So, if array_pointer points at an array element, then after the addition it points at the next element of the array.

3.45 subtraction operator.

syntax

expression - expression

examples

```
2 - 5
pointer1 - 3
pointer2 - pointer1
```

This operator yields the difference of the two expressions. An integer can be subtracted from a pointer and, in common with addition, the integer is first multiplied by the size of the pointed-to object. The subtraction of two pointers to objects of the same type is quite legal, and yields the number of objects between the first and second pointers. That is to say, it yields the number of bytes divided by the size of the pointed-to type. This is meaningful only if the pointers point to the same array.

3.46 shift operators.

syntax

expression << expression

expression >> expression

examples

```
1 << bit_number
n >> 3
```

These operators are best considered as bitwise operators, as their action is to shift the bit pattern represented by the left-hand expression left (<<) or right (>>) by the number of bits (binary digits) specified by the right-hand expression. Shifting left always fills vacated bits with zeros, while shifting right fills vacated bits with the sign bit if the number is signed, and zero if it is unsigned. Shifting left is equivalent to multiplying the left-hand expression by two to the power of the right-hand expression; and shifting right is equivalent to an analogous division. A negative shift count reverses the direction of the shift.

3.47 relational operators.

syntax

expression < expression

expression > expression

expression <= expression

expression >= expression

examples

```
-5 < 3

lower_limit <= variable  &  variable <= upper_limit
```

These operators return boolean values - TRUE (1) or FALSE (0) depending on whether the first expression is less than (<), greater than (>), less than or equal to (<=), or greater than or equal to ('>=') the second. Pointers may be compared using these operators, and the result indicates which points to a higher memory address. Use unsigned rather than int variables for numerical work on memory addresses.

3.48 equality operators.

syntax

expression == expression

expression != expression

examples

```
if (i == 5) puts("i equals five");

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

These operators return boolean values - TRUE (1) or FALSE (0) depending on whether the first expression is equal to (==) or not equal to (!=) the second. Pointers may be compared using these operators. Pointers may also be compared to integers representing memory addresses. The integer 0 (NULL) is a special case, which is defined as pointing to nothing.

3.49 bitwise AND.

syntax

expression & expression

examples

```
status_value & mask

alphabetic_character & 0x5F
```

This operator performs the bitwise AND of the first operand and the second, setting each bit in the result only if both the corresponding bits in the two expressions are set.

3.50 bitwise exclusive OR.

syntax

expression ^ expression

example

```
alphabetic_character ^ 0x40
```

This operator performs the bitwise XOR (EOR) of the first operand and the second, setting each bit in the result if either but not both of the operands have a set bit in the corresponding bit position.

3.51 bitwise inclusive OR.

syntax

expression | expression

example

```
40 | 10 == 42
```

This operator performs the bitwise OR of the first operand and the second, setting each bit in the result if either or both of the operands have a set bit in the corresponding bit position.

3.52 logical AND operator.

syntax

expression && expression

examples

```
return  0 <= x  &&  x < 10 ;

while (list  &&  list->head != key) list = list->tail;
```

This operator returns TRUE (1) only if both expressions are non-zero; otherwise it returns FALSE (0). Unlike the bitwise AND operator (&) && guarantees a left-to-right evaluation of its operands, and it will not evaluate its second operand if the first is 0 (FALSE or NULL or zero).

3.53 logical OR operator.

syntax

expression || expression

example

```
if (we_know_its_OK  ||  we_can_prove_it_is()) puts("its OK");
```

This operator returns TRUE (1) if either or both of its operands are non-zero. Unlike the bitwise OR operator (|) || guarantees a a left-to-right evaluation of its operands, and it will not evaluate its second operand if the first is non-zero.

3.54 conditional operator.

syntax

expression1 ? expression2 : expression3

examples

```
absvalue = number < 0  ?  -number  :  number

return isdigit(c) ? c-'0' : isxdigit(c) ? 10-'a'+tolower(c) : ERROR ;

f() { puts("error"); }
main()
{
  int i, j;
  i = 1; j = 0;
  ( i ? 0 : 1 ) ? ( f() ? f() : f() ) : ( j ? f() : puts("OK") ) ;
  i = 0;
  i ? 0 : 1   ?   j ? f() : puts("OK")   :   f() ? f() : f()   ;
}
```

This operator is the only 'ternary' operator (it has three operands) that C supports. Its effect is to evaluate the first expression, and if this is non-zero, the result is the value of the second expression, while if the first expression is zero, the result is the value of the third expression. Only one of the second and third expressions is evaluated. If one expression is a pointer and the other an integer (which must be zero), then the result has the type of the pointer.

3.55 Assignment operators.

syntax

lvalue = expression
lvalue += expression
lvalue -= expression

```
lvalue *= expression
lvalue /= expression
lvalue %= expression
lvalue >>= expression
lvalue <<= expression
lvalue &= expression
lvalue ^= expression
lvalue |= expression
```

examples

```
i = 5

a[39] [54*f(5)] += 3

pointer -= 7
```

C has the standard assignment operator, '=', which puts the value of the expression into the lvalue, and it also supports a host of composite assignment operators formed by concatenating an arithmetic or bitwise operator with the assignment operator. The general action of the idiom can be explained:

```
lvalue  op=  expression
```

is essentially equivalent to

```
lvalue  =  lvalue  op  expression
```

but the lvalue is evaluated only once. This method therefore provides a more concise and efficient way of performing common assignment operations, as we can now type

```
lvalue /= 7;
```

to divide a variable by 7, rather than the more cumbersome

```
lvalue = lvalue / 7;
```

In common with simple addition and subtraction, we can use a pointer type lvalue and an integral expression with '-=' and '+=', which behave like their non-composite counterparts.

An important point to notice about C assignments is that they are themselves expressions, and have a value equal to the value of the left-hand operand after the assignment (which may have involved truncation). Thus, if the standard input is at end-of-file, then the sequence:

```
int  int_var1, int_var3;
char char_var2;

int_var1 = char_var2 = int_var3 = getchar();
```

sets int_var3 to -1 (the end of file marker value), char_var2 to 255 (because the value is truncated to fit the char variable), and int_var1 to 255 also.

3.56 Type coercions and the type of the result of an expression.

Every operator expects its operands to have particular types, and the result of the expression formed from the operator and its operands will also have a particular type. Thus, a general expression will have a result type, and unless this is explicitly coerced to another type using the cast operator, it can be deduced from the coercion rules which C compilers follow. The rules for normal arithmetic are:

- First, any operands of char type are converted to ints.
- Then, if either operand is unsigned, the other is changed to unsigned and the result is unsigned.
- Otherwise, both operands must be ints, so that is the type of the result.

Pointers, where they are allowed, produce slightly different result types which are summarised in the Table of Operator Precedence and Types above.

3.57 Constant Expressions.

We've looked at constants earlier on, and an expression formed entirely from constants and certain operators is known as a 'constant expression'. A constant expression always has a particular value, but it may be more convenient to write it as a composite expression rather than as a single constant. Nevertheless, a constant expression will be evaluated at the time the program is compiled, therefore saving on run-time execution speed. Constant expressions are only brought in as a syntactic term to emphasise those places where they are needed.

3.58 Data Declarations.

Data declarations fall into two categories; identifier declarations and constant definitions. We are concerned here with the declaration of identifiers, whether they are simple integer variables, complicated structures and arrays, or functions. Constant definitions are handled by the pre-processor, so each definition is on a line beginning with the "#define" symbol. These are discussed elsewhere in the pre-processor sections.

Identifiers in a C program have two important attributes: the data type and the storage class. The data type defines how the data is to be interpreted and manipulated by the compiler and program. The storage class defines how the data is stored by the compiler: it may be limited to a function's stack frame, or it may be available to the whole program throughout the life of the program by being stored elsewhere.

3.59 The scope of identifiers.

Each identifier declared in a C program has a 'scope', which determines its validity throughout the program. An identifier is in scope only following its textual declaration (which may not necessarily coincide with its definition). If an identifier is in scope at a given time, then it is 'known about' at that time in the program and can be used with perfect validity.

An identifier declared outside any function is in scope throughout the rest of the program, except that any identifiers with the same name which are declared within a function take precedence. An

identifier declared within a function is in scope throughout the function, and any identifiers of the same name declared outside the function will be hidden until the end of the function definition. When an identifier falls out of scope and subsequently falls back in again, its value will not have been altered. Let us give an illustration of all this:

```
int i = 'i';  /* a global variable initialised with the character 'i' */

int f()
{
  i = 'f';              /* new value for the global variable */
  printf(" %c", i);     /* which is printed */
}

int g()
{
  int i;                /* a local variable, also called i */

  i = 'g';              /* is given a value          */
  printf(" %c", i);     /* and the value is printed */
}

int main()              /* this is where execution of the program starts  */
{
  printf(" %c", i);     /* print the initial value of the global variable */
  i = 'm';              /* give it a new value */
  printf(" %c", i);     /* and print that      */
  f();                  /* call the function f and see how that affects i */
  printf(" %c", i);     /* by printing the value of i after calling f     */
  g();                  /* do the same for the function g */
  printf(" %c", i);
}                       /* and this is where the program stops */
```

When this program is run it prints out:

```
i m f f g f
```

3.60 Data Types and type_specifiers.

Variables in C programs are each of a particular type, which may be one of the predefined types (eg int variable), or a user-defined type declared with a typedef declaration (eg FILE *fp), or an anonymous type (eg int *(*f())[]). The C compiler checks that the variables used in an expression are of compatible types in order to help detect programming errors.

The type checking is done by a method known as **structural equivalence.** This means that two variables have the same type if their declarations have the same structure no matter where they are declared. This notion is also employed when operators such as * (indirection) and & (address) are applied. So for example if:

```
int polar[2];
typedef complex[2];
complex z;
int (*ptr_Cartesian)[2];
```

then polar, z, and *ptr_Cartesian are all of equivalent types. This is different to some other languages where they need to have the same named type to be equivalent or to be declared at the same place. Structural type equivalence is more flexible, but can lead to obscure bugs if misused and it is generally good practice to give a name to complicated types by using typedef.

The type specifiers are: char, int, unsigned, struct, union, typedef. The type specifier, if left out, defaults to 'int'.

3.60.1 char.

syntax

char declarator_list ;

example

```
char c, another_character, a_string[50];
```

The char data type is usually used to represent a single character, and for this purpose it can be considered as an unsigned quantity which can hold any single member of the host computer's character set. This implies that a char will occupy at most eight bits on Z80 based machines, so a char is generally equivalent to a byte. It is important to remember that a char is unsigned.

Surprisingly on an 8 bit computer like a Z80, it is actually more efficient to use 2-byte int variables than 1-byte chars; except where there is an array or string of characters.

3.60.2 integer.

syntax

int declarator_list ;

example

```
int a_number, a_character_variable, a_count, a_truthvalue;
```

This data type is used to hold integers, which are signed numbers from a given minimum value (often referred to as 'minint', but don't take it for granted!) through zero to a maximum value, often called 'maxint'. In Hisoft C, minint is -32768 and maxint is 32767: because an integer is the usual Z80 two's complement 16-bit integer.

The int type is also used to hold character variables, because end-of-file is marked by a special value of -1 which will not fit into a single byte. It is also used for truthvalues (TRUE is 1 and FALSE is 0).

The integer is the default data type in instances where a data type definition is optional. So a function will be assumed to return an integer unless specified otherwise. Programs are generally clearer if the type is defined explicitly everywhere, because there is another convention sometimes used, which says that "If I don't declare the type, its because I don't use the result".

3.60.3 unsigned.

syntax

```
unsigned declarator_list ;

unsigned int declarator_list ;
```

example

```
unsigned an_address, a_large_number, a_bit_pattern;
```

The two forms of the syntax above are completely interchangeable. An unsigned int is the same size as an int (ie two bytes), but the number is not considered to be signed two's complement. This means that an unsigned variable on a Z80 system can hold values between 0 and 65535. Although this range is co-incidentally the same as that held by a pointer, it does not imply that the two data types are interchangeable!

3.60.4 pointer.

syntax

```
* declarator
```

examples

```
char *s;
struct tag_name (*f())[];
```

The pointer is not really a type, and is used as the basis for indirection. It holds a value which, if considered as a machine memory address, is the address of a variable of any type. As a pointer can be seen as a valid machine address, it follows that a pointer is always the same size regardless of what type of object it points to. On a Z80 system, memory addresses are unsigned 16-bit numbers from 0 to 65535, but it is unusual and not very sensible to use explicitly the address held in a pointer.

One would normally get to an object through a pointer by taking advantage of the indirection facilities offered by the C language. Note also that there is nothing unusual in having a pointer point to another pointer, which may point to another pointer, and so on. Pointers may point to any other type of object, and this is explored in more detail below under the heading "Building up Data Types". In the examples above "s" is a pointer to a char; while "f" is a function which returns a

pointer to an array of structures. A pointer has the type 'pointer to ...', where '...' is the type of the object pointed to.

3.61 Aggregate data types.

A language would not be complete if it could not store related data grouped together in a sensible fashion. Almost all languages support arrays for just this purpose; C also provides structures and unions.

3.61.1 Arrays.

syntax

declarator [ ]

declarator [constant_expression]

examples

```
char s[50];
int t[10][20];
struct tag_name a[42];
```

An array is a sequence of individual items, each of the same data type, which can be accessed by an index from a base. This is called subscripting, and the index into the array is called the subscript.

If the elements of an array are stored contiguously in memory, which they are in C, then the name of the array is equivalent to its base address. As C arrays start at subscript zero, it follows that this address is also the address of the first element of the array.

When a subscript is applied to an array, it is multiplied by the size of the data item and added to the base, thus giving the address of the required element. However, subscripting also implies indirection, so the data object accessed in this way is actually the value of that member of the array rather than its address.

C arrays can be based on any type, including further arrays, so it follows that arrays can have more than one dimension. Although the name of an array equates to its base address, this name is actually a constant at compilation time. The significance of this will be made clear when we discuss 'lvalues'.

An array is specified by following the identifier concerned with '[' square brackets containing the array dimension: For example "char s[50];" declares "s" as being an array of 50 characters (from index 0 to index 49); and "int t[10][20]" declares 't' as a two-dimensional array (consisting of ten arrays of an array of 20 integers). "a" is an array of forty-two structures.

3.61.2 Structures and Unions.

syntax

```
struct  {  struct_decl_list  }
struct  tag_name  {  struct_decl_list  }
struct  tag_name
union   {  struct_decl_list  }
union   tag_name  {  struct_decl_list  }
union   tag_name
```

struct_decl_list:
type_declaration_LIST

examples

```
struct s_tag {                          /*  1  */
    char s;
    char *t;
    int i;
    struct s_tag *sp;
  };

struct list {                           /*  2  */
    int head;
    struct list *tail;
  }
    a_list_item, *a_list_pointer;


struct s_tag {                          /*  3  */
    char s;
    char *t;
    int i;
    struct s_tag *sp;
  } ;
/* some other declarations */
struct s_tag  item, shelf[20], *this_item;
static struct s_tag  v;
```

Arrays are all well and good, but they have the limitation that each member of the array must be of the same type. So, although we can have an array of arrays, we cannot have an array holding a character, an integer AND a pointer. The Pascal language overcomes this by implementing a 'record' data type; in C, the equivalent of a record is the structure.

A structure can hold any number of items of any variety of types, including further structures, arrays, and arrays of structures. They do share some of the limitations of arrays, in that the only manipulation we can perform on a structure as a whole is to take its address. In much the same way that an array name is a constant, a structure name is also a constant. Like arrays, we can maintain a pointer to a structure and thus facilitate our data manipulations.

Each item in a structure is called a member, and to access a member we need to specify the member name as well as the structure name (or pointer). We must stress that each member can be of any type, and that structures can themselves be the elements of an array.

Structure and union declarations are slightly different to the previous kinds, as a structure is an arbitrary aggregate item. A structure "template" can be declared without any identifiers taking on this type as in the first example above. This associates the "structure tag" ("s_tag" in this example) with the structure format given. No identifiers are defined and no storage space is allocated, but the structure tag can subsequently be used in declarations. Our example above associates s_tag with a structure type containing a character member s, a character pointer member t, an integer i and a pointer to a structure sp. Notice that in this example, the structure pointer sp also points to an s_tag type structure so we can form chains or lists of structures.

We may declare identifiers at the same time as the template as in the second example where we have declared a type of structure called a "list". We then go on immediately to declare one of these structures, and also a pointer to this structure type.

We can make a list of numbers using this structure. The list pointer points to one of the structures which is at the front of the list. The first member of this structure is its "head" which is the first number in our list. The second member of the structure is its "tail" and this points at the next structure in the list, which in turn contains the second number in the list and so on. The process comes to an end when we find a structure whose "tail" is zero (NULL); this marks the end of the chain.

Or we may declare the necessary identifiers sometime after the template definition, as in the third example. In this case we have declared "item" as being a single structure of type s_tag, "shelf" as an array of 20 structures of this type, and "this_item" as a pointer to an s_tag type structure. "v" is a static structure of type s_tag.

Note that the structure tag is entirely optional, but it does not make sense to declare a template alone without a tag, as we would be unable to use the thus-defined template in any declarations.

A **union** is the "do anything" data type of C, and its purpose is to hold any given data type. It can hold one data type only at a given time, but we may specify which data type it is by referring to the member of the union which is of that type. Unions are declared in exactly analogous ways to structures, with the "struct" keyword being replaced by "union".

3.61.3 Pointers to functions.

Functions themselves are typed, meaning that we declare what data type the result of a function will be when we write it. Functions can return all the types we have discussed, but aggregate types can only be returned whole as pointers; the value of an individual non-aggregate member can be returned under all conditions. More importantly, a pointer to a function is a subset of the pointer type, which means that we can create arrays, structures and so on of pointers to functions. This is not quite the same as the (impossible) creation of an array (etc.) of functions.

3.62 Storage Classes.

syntax

```
auto   type_specifier_OPT  declarator_list  ;
static  type_specifier_OPT  declarator_list  ;
extern  type_specifier_OPT  declarator_list  ;
register  type_specifier_OPT  declarator_list  ;
typedef  type_specifier_OPT  declarator_list  ;
```

examples

```
auto local_variable;
static int permanent_local_variable;
extern char *strcpy();
register char another_local_variable;
typedef int *int_ptr;
```

The storage class of an item of data determines both how the compiler will store that item and for how long it will be stored. The storage class specifiers available are: auto, static, extern, and register. We will examine each class in more detail below.

The storage class defaults depend upon the location of the declaration. If the declaration is inside a function and the storage class specifier is left out, it will default to "auto", which introduces an automatic variable. If the declaration is outside of any function, the storage class defaults to "external".

3.62.1 automatic.

This class is the equivalent of most other languages' "local variable", as an automatic variable is created at the point of invocation of the function containing it, and it is "killed" when the function terminates. An automatic variable lives on the function's stack frame, which means that it is not accessible to any other function directly. Naturally, a variable's value (or even its address) can be passed to another function using C's parameter passing facilties.

This storage class is the default within functions, as it is assumed if no other storage class is specified. Also, automatic variables can exist only within functions. A variable can be declared explicitly as automatic by preceding its declaration with the "auto" keyword, but this is only necessary if the type-specifier is omitted.

An automatic variable can of course be of any type, but if an aggregate data type variable, such as an array, is declared within a function and returned as the result of the function, the result will be meaningless as the variable will no longer exist. Individual element values may be returned, but to return a pointer to an array, structure or union which was declared as automatic is an error.

3.62.2 static.

Data which is declared as static exists for the duration of a program, but may be local to a function or available to the whole program, depending upon the location of its declaration in the source program. If it occurs within the body of a function, then it is local to that function. The fact that a static variable exists throughout the life of a program also means that it holds its value between manipulations. This means that a variable may be local to a function, but its value or address, even if it is an aggregate data type, may be returned to the calling function and safely be used.

As static variables are allocated permanent storage, the compiler will always know the address of each one, while it will know only the offset from a function's stack frame of an automatic variable. This means that static variables can be accessed and manipulated with greater speed and fewer bytes of code than automatic variables, as no indexed addressing is involved.

3.62.3 external.

In Hisoft C the "extern" keyword has a restricted use as the entire program is compiled as one unit. It is used to declare a function which has a non-integer type so that other functions which use the function before its definition know what type it is. The extern keyword simply declares a non-integer function to the rest of a program in much the same way that FORWARD is used in Pascal.

3.62.4 register.

A register variable normally instructs a C compiler to use a machine register for the designated variable, but there are not enough registers free on the Z80 to implement this. The keyword "register" is therefore ignored.

3.62.5 typedef.

typedef is syntactically identical to a storage class specifier, but it does not cause a declaration to allocate any storage. It is used to equate a given identifier with a data type, allowing user-defined types, and associates the identifier with the given type name for the duration of that identifier's scope. We could equate the identifier 'int_ptr' with the type 'pointer to integer' as above and subsequently use 'int_ptr' as a type specifier in further declarations.

```
int_ptr a,b;
static int_ptr *c = NULL;
```

which declares 'a' and 'b' as being pointers to integers, and 'c' as being a static pointer to a pointer to an integer with an initial value of NULL (0). The names which are defined with typedef can also be used as arguments to "sizeof" and "cast" in expressions.

3.63 Initialisation

syntax

```
declarator = constant_expression

declarator = { initializer_list }

declarator = { initializer_list , }

initializer_list:
  constant_expression
  initializer_list , initializer_list
  { initializer_list }
```

examples

```
static int i = 3;
char c = 'S', t;
char arr1[] = {'A','r','r','a','y'};
char arr2[] = "Array";
static char arr3[6] = {'A','r','r','a','y','\0'};
```

We can declare a variable and give it a starting value at the same time: this is called "initialising" the variable (or in American "initializing"). Only static and external variables may be initialised, and this is done by putting an "initializer list" after the identifier in the declaration.

The first declaration above declares 'i' as a static integer and gives it the initial value of three; the second declaration declares 'c' and 't' as external character variables and initialises c to the character constant

When initialising an aggregate, the initialiser list must be enclosed in braces as in the third example above. Notice also that we omitted the array bound from between the square brackets. The compiler works this out by counting the number of elements (in this case 5). The compiler will allocate just enough space (and therefore just enough elements) to hold each character in the list.

Character array initialisation can be regarded as a special case, as string expressions and constants in C are nothing more than character arrays terminated with a '\0', so the character array "arr2" above is actually given SIX elements, all of which are initialised. This means that arr2 and arr3 above are initialised with the same contents, while arr1 is one element shorter.

If there are fewer items in the initialiser list than elements in the aggregate, the final elements are filled with zeros. If there are more initialisers than elements, an error is generated.

3.64 The Compiler Preprocessor

This section gives details of the preprocessor part of the Hisoft C compiler. A preprocessor command is one which is preceded by a '#' sign and they are used to provide a way of controlling compiler options (such as whether a listing is displayed) and also to define constants in programs.

Historically, the C preprocessor was a separate program which was run before the compiler to add extra facilities; but now, in **HiSoft C**, the preprocessor is just another part of the compiler. The separate history of the preprocessor means that the preprocessor inteface is not as clearly defined as the language itself, as different systems have different requirements. The commands supported in **HiSoft C** are as follows:

3.65 Constants and Macros

syntax

#define <identifier> <macro>

examples

```
#define EOF -1
#define MAXIMUM 1000
```

This command allows for limited "macro expansion", and is the means by which constants are defined in C programs. The text following the #define command is normally a name which we would choose to use in place of a number or expression, such as `EOF` rather than `-1`. This is the <identifer> part of the command. The <macro> part is the text with which the identifier is to be replaced whenever it is used in the main program. If we had the example preprocesssor commands at the head of a program then whenever we used `EOF` or `MAXIMUM` in the program they would be replaced by `-1` and `1000` respectively.

Remember that the *whole* of the text following the #define <identifier> is used in the replacement, so the very commom error of putting a semi-colon after the <macro> part should be taken particular notice of. If we had typed

```
#define EOF -1;
```

then every occurence of `EOF` would be replaced by `-1;` which is not normally what is required.

3.66 Error Message Sacrifice

syntax and example

```
#error
```

This command effects a once-only removal of compiler error messages, releasing the space taken up by these messages to the compiler. This would normally only be used if a program were particularly large. Subsequent error detection will result in only the error number being reported rather than the full text.

3.67 Listing Control

syntax and examples

```
#list-

#list+
```

This command is followed by either a '+' or a '-' sign and switches the compiler listing on or off as appropriate. These commands may of course be nested, so that the common practice of having a #list- at the start of a header or library file and a #list+ at the end of it has the desired effect.

3.68 Direct Execution

syntax and examples

```
#direct+

#direct-
```

There is an additional feature in the Amstrad CPC version of **HiSoft C** that permits the direct execution of statements compiled by the compiler. This is sometimes called *immediate execution* and is similar to typing a command in BASIC without a line number.

The direct execution mode is almost certainly unique to the **HiSoft C** compiler and allows programs and functions to be tested as they are written. Once a #direct+ command has been issued functions can be invoked, variables can be assigned to, loops can be run and any other statement can be executed. By typing a function name followed by any parameters and a semi-colon the function will be executed. Normally this will only be useful when the function prints its result:

```
#direct+
printf("%u in hex is %x\n",3000,3000);
3000 in hex is BB8
```

will print out the decimal and hex equivalents of 3000 onto the screen.

Remember that an attempt to invoke a non-existent function results in a jump to a random location, which almost certainly means that the machine will crash and the compiler will have to be reloaded.

3.69 Using The Data Origin Directive

syntax

#data number

example

```
#data 0x3000
```

#data nnnn causes the of the global/static fixed data area to be placed at location nnnn. It is mainly used to produce programs to run on CP/M systems with differing memory sizes as described in **Chapter 4**. It enables programs to run unchanged on any size of TPA which is big enough! The compiler suggests a value for use with this directive at the end of every compilation. The suggested value is the possible, and it is better during program development to use a larger value so that the program will still compile successfully as changes are made.

3.70 CPC File inclusion

syntax

```
#include                /* 1 */

#include "filename"     /* 2 */
#include <filename>
#include filename

#include ?filename?     /* 3 */
```

This command is probably most usefully seen as a command to initiate compilation, although its more precise effect is to include a specified file in the compilation of another. It has three major forms, as above.

#include

This compiles the program held as source text in the integral editor's memory space. This program may itself contain further #include directives, but they will not be of this type.

#include filename or
#include "filename" or
#include <filename>

This command causes the named file to be included in the current compilation, which may mean that the named file alone is compiled if it is issued as a stand-alone command. If the command forms part of another file then the named file will be included in the compilation of the calling file. A file included in the compilation of another may not itself contain further #include directives. All three forms given above are equivalent.

#include ?filename?

This variation of the command allows for library file inclusion. In many respects it also gives the user a conditional compilation capability, as it only compiles those functions in the named file which have been used somewhere in the main file. The command would normally be the last instruction in a file, and its effect is to scan through the named library file looking for previously-invoked functions. When one is found it is compiled into the current program. This means that if one were using the standard library to provide certain functions for a program, it would not be necessary to separate all the functions used into a separate file and include that as would otherwise be necessary.

However, as the compiler only includes those functions which have been invoked but not defined, care must be taken to ensure that any library functions which make use of of other library functions are present in the library file *before* the subsidiary ones. Otherwise it will be necessary to search the library again. In other words if a program uses a function f() and this function calls a further function g(), in which both f() and g() are in the library, then the definition of f() must occur before that of g(). The standard library supplied with **HiSoft C** is constructed in this way so no problems should occur in its use.

You must type End-Of-File after compiling everything. **Getting Started** in **Chapter 1** tells you how.

3.71 CP/M File Inclusion

syntax

```
#include "filename"                    /* 1 */
#include <filename> #include filename

#include ?filename?                    /* 2 */
```

This command causes a specified file to be compiled within the compilation of another. It has two major forms, as above.

The first form causes the named file to be included in the current compilation, which may mean that the named file alone is compiled if it is issued as a stand-alone command. If the command forms part of another file then the named file will be included in the compilation of the calling file. All three variations given above are equivalent. A file included in the compilation of another may itself contain further #include directives. The level of nesting is limited by the storage allocated to the input-output system: this is detailed in **Chapter 4.**

#include ?filename?

The second form of the command allows for library file inclusion. In many respects it also gives the user a conditional compilation capability, as it only compiles functions in the named file which have been used somewhere in the main file. The command would normally be the last instruction in a file, and its effect is to scan through the named library file looking for previously-invoked functions. When one is found it is compiled into the current program. This means that if one were using the standard library to provide certain functions for a program, it would not be necessary to separate all the functions used into a separate file and include that as would otherwise be necessary.

However, as the compiler only includes those functions which have been invoked but not defined, care must be taken to ensure that any library functions which make use of of other library functions are present in the library file *before* the subsidiary ones. Otherwise it will be necessary to search the library again. In other words if a program uses a function f() and this function calls a further function g(), in which both f() and g() are in the library, then the definition of f() must occur before that of g(). The standard library supplied with **HiSoft C** is constructed in this way so no problems should occur in its use.

3.72 CPC Stand-Alone Programs

syntax

```
#translate object_file_name
```

The compiler normally compiles a program and expects it to be executed with the compiler still resident in memory. But it also caters for programs which are intended as stand-alone programs which are loaded and run without the necessity to load the compiler first. #translate allows for this, instructing the compiler to save the program and all the required runtime routines to cassette, disc or microdrive with the given file name. #translate can only be used at the start of a program.

In normal operation the compiler stays in memory along with the object code which it is producing. In most applications this will be quite acceptable, but there are instances when it would preferable if the program could run by itself without the need to load the compiler and compile the program before using it. If, for example, one were producing a product intended for commercial sale, it would not be very useful unless it had the ability to run by itself. #translate gives it this power, and the resultant file is a machine code program which can be loaded and run from BASIC as described in **Chapter 1.**

This command would normally only be given once a program has been fully tested, as there is little point in having a non-working stand-alone program! The facility is also useful for large programs with much external data where these cannot fit in the memory at the same time as the compiler.

3.73 CP/M Object Program Names

syntax

```
#translate object_file_name
```

The compiler normally compiles the object program with the same name as the first (or only) source file on the command line. The filetype (ie the filename extension) is changed to .COM. But it also caters for programs which need a different name and #translate instructs the compiler to save the program with the given filename. #translate can only be used at the start of a program.

3.74 Conclusion

That brings us to the end of our survey of **HiSoft C.** We hope that it serves as an easy-to-use, complete, and accurate guide to the language. You will keep making new discoveries about C every time you refer to it.

HiSoft C

Fast Interactive K&R C Compiler

Chapter 4

The Expert's Guide

HiSoft
High Quality Software

4. THE EXPERT'S GUIDE.

He that commands the C is at great liberty,
and may take as much and as little of the war as he will.

Francis Bacon

This chapter aims to provide a detailed reference for the experienced C user; both those who know C already and those who learn it with this compiler.

4.1 Differences from Kernighan & Ritchie.

Note the following overriding restriction not otherwise mentioned :

- Floating point is not yet implemented, which means that use of types float and double will cause the compiler error 'RESTRICTION : floats not implemented' to be produced and compilation will terminate.

1. Introduction

Hisoft C is designed to be very close to the language described in the book 'The C Programming Language' by Kernighan and Ritchie. That book is THE reference work on C and we recommend that all users of Hisoft C should have a copy of it. Because it is close to that definition of C, Hisoft C is also close to other dialects of C that exist which are also based on it. Other possible reference works are listed in the Bibliography in Chapter 1.

Most parts of the Kernighan and Ritchie book apply directly to Hisoft C and we do not repeat the full description here. This chapter describes the differences between Hisoft C and the language described in the 'C Reference Manual' which is Appendix A of Kernighan and Ritchie. That Appendix is a concise definition of the language and this chapter is a concise definition of the differences. More explanation of some aspects is given elsewhere in the manual.

This chapter is given section numbers in just the same way as the 'C Reference Manual' so that the comments here apply to the corresponding section in that book. Where the section heading is given with no comments Hisoft C is just as described in the book. We start now with the comments on section 1 . . .

Hisoft C is implemented on the Sinclair ZX Spectrum, and the Amstrad CPC464 and CPC664 computers.

2. Lexical Conventions

2.1 Comments

2.2 Identifiers (Names)

External identifiers are used only for forward declaration of the type of a function in Hisoft C, otherwise identifiers are just as described.

2.3 Keywords

No differences. Note that keywords MUST be lower case.

2.4 Constants

2.4.1 Integer constants

No differences, but note that long is the same as int (see 2.6).

2.4.2 Explicit long constants

As for 2.4.1.

2.4.3 Character constants

2.4.4 Floating constants

2.5 Strings

2.6 Hardware characteristics

	Z80 ASCII	
char	8 bits	= 1 byte
int	16	2
short	16	2
long	16	2
float		
double		
range		

3. Syntax notation

4. What's in a name?

There are two differences in the storage classes. The first is that register variables are always treated just like automatic variables because there are not enough registers available to allocate any to variables! The keyword 'register' is accepted but ignored so there is actually no difference in the language which the Hisoft compiler accepts. The second difference is that all local variables must be declared at the head of a function body and may not be declared at the head of nested compound statements. This is not a serious restriction in practice and can be defended on the grounds of ease of understanding of programs - but it also helps keep the compiler small!

All the fundamental types are provided, but as noted above short is the same as int, as is long. It is hoped that a future version may include 8 bit short variables and 32 bit long variables.

5. Objects and lvalues

6. Conversions

6.1 Characters and integers

Sign extension is not performed when converting characters to integers, so characters appear as integers in the range 0..255.

6.2 Float and double

Floating point numbers are not yet available so this section has no relevance.

6.3 Floating and integral

6.4 Pointers and integers

6.5 Unsigned

6.6 Arithmetic conversions

7. Expressions

The compiler generally computes expressions from the inside-out and from left to right but, as stated in the reference manual, this should not be relied upon as it may change between releases of the compiler. Integer overflows are ignored as are attempts to divide by zero.

7.1 Primary expressions

Function arguments are evaluated in left to right order and are pushed on the stack as they are evaluated. As the reference manual says, this order should not be relied upon. See the notes for section 7.

7.2 Unary operators

The syntax of the type-conversion operator is different to that described in the reference manual, in order to simplify the compiler (and make the programs more readable). The operator has a name - "cast" - and must be given a predefined type or a "typedef-name" as argument; it will not accept an anonymous "type-name" such as (int **()). To move an existing C program to this compiler, give the type a name in a typedef declaration and add the word "cast" before the parentheses. To compile a program using this syntax with another compiler define the word "cast" to mean nothing thus:

```
#define  cast
```

The sizeof operator has a similar syntax and only accepts a predefined type or a "typedef-name". It will not accept an anonymous type or an expression. These differences do not restrict the programs that can be written, but simply require them to be written in a particular style.

7.3 Multiplicative operators

This compiler is like those described in the reference manual in that the remainder of a division operation has the same sign as the dividend. Thus truncation is towards zero.

7.4 Additive operators

7.5 Shift operators

An arithmetic shift is performed when an int is right-shifted, so that the result has the same sign as the original. A logical (zero-fill) shift is performed when an unsigned is right-shifted. A negative shift count CAN be used, for those occasions when it is not possible to know in advance which direction of shift is needed.

7.6 Relational operators

7.7 Equality operators

Just like most other compilers, the nil pointer (or null pointer) is actually as well as conceptually zero and in consequence pointers which may point at location zero in memory cannot be easily tested against NULL.

7.8 Bitwise AND operator

7.9 Bitwise exclusive OR operator

7.10 Bitwise inclusive OR operator

7.11 Logical AND operator

7.12 Logical OR operator

7.13 Conditional operator

7.14 Assignment operators

The compiler does not allow the assignment of pointers to integers or vice-versa except for the assignment of 0 (zero) to pointers as NULL. This restriction is designed to make it less likely that the wrong value will be assigned to a pointer (eg missing * or & operator) and thus less likely that the store is overwritten in consequence. An explicit type conversion can be used to make pointers point at particular areas of memory:

```
f()
{
  typedef int *location ;
  location  ptr ;
  ptr = cast(location) 0xBC00 ;
}
```

7.15 Comma operator

The comma operator is not implemented.

8. Declarations

In accordance with normal C programming style, any storage class specifier in a declaration must come before any type specifier(s).

8.1 Storage class specifiers

The 'extern' specifier has a restricted use since there are no separate files. It is used in the special case of library functions of non-int type. These functions have not been defined when they are used in a program, because they will be defined when the library file is searched at the end of the program (see section 12.2). However, their type must be declared before they are used otherwise they will be implicitly declared as function-returning-int. So an extern declaration must be made at the start of the program. This declaration is made in the library header file for the standard library functions. This declaration does not cause the function to be loaded from the library, only actual use of the function does that.

The register specifier is accepted but ignored since the compiler has no registers available for variables.

8.2 Type specifiers

The specifiers long and short are accepted but ignored. A long int and a short int are the same as an int (ie 16 bits).

8.3 Declarators

8.4 Meaning of declarators

8.5 Structure and union declarations

Fields, often called bitfields, are not implemented. It is necessary to use bitwise operators and shifts to access particular bits in a byte.

The names of structure tags and members share the same name-space as ordinary variables so a structure cannot have the same name as an integer, for example.

8.6 Initialisation

Initializers are provided for static and external variables but not for automatic variables. The initialisation of automatic scalar variables should just be replaced by an assignment statement. The initialisation of automatic aggregates (ie stack-based local arrays and structures) is not permitted in C; but HISOFT - C provides "blt()" in the function library, and this can be used to initialise a

local array or structure by copying the contents of a static array. Note that a static variable is preferable to an automatic one unless the function is recursive or space allocation dictates an automatic. A structure initializer is a series of constants which generate bytes like "inline" (see 9.14). They are not aligned on field boundaries. Initialisation is performed each time the complete program is run. It is not performed in direct mode.

8.7 Type names

The only type names permitted are those of the predefined types, and those declared in a typedef declaration. Abstract declarators are not allowed. They can be replaced by a typedef declaration where they occur in existing programs, with a resulting increase in clarity and type security.

8.8 Typedef

9. Statements

9.1 Expression statement

9.2 Compound statement

Declarations are only permitted at the head of a function body, and are not allowed in other compound statements.

9.3 Conditional statement

9.4 While statement

9.5 Do statement

9.6 For statement

9.7 Switch statement

There can be no declarations at the head of the enclosed statement.

9.8 Break statement

9.9 Continue statement

9.10 Return statement

9.11 Goto statement

9.12 Labelled statement

9.13 Null statement

9.14 Inline Statement (NEW SECTION)

HiSoft C provides the ability to incorporate machine code into C programs. A new type of statement (inline(k1, k2, k3, ...)) is used and it is fully described in the **Language Summary.**

10. External definitions

10.1 External function definitions

There is an addition to the syntax of function definitions which provides a way of defining functions which take a variable number of actual arguments (these are called "variadic" functions). This is fully described in the **Language Summary.**

10.2 External data definitions

11. Scope rules

In **HiSoft C** an entire program is compiled at once so that there is no linkage of precompiled routines. A source library facility is provided instead and this is described in **section 12.2.**

11.1 Lexical scope

Identifiers associated with ordinary variables and those associated with structure and union members and tags are not disjoint.

11.2 Scope of externals

12. Compiler control lines

12.1 Token replacement

Macro definition is limited to simple token replacement and no arguments are permitted.

#undef control lines are not implemented.

12.2 File inclusion

Files may be included. The named file is searched for on the current device, so the two forms are identical. The " or <> brackets may be omitted. So the following control lines are all equivalent:

```
#include "filename"
#include <filename>
#include filename
```

In addition a library search facility is provided. A control line:

#include ?filename?

causes the named file to be read and searched.

Another variation, without a filename, is used to compile source that has been prepared with the editor and is now in its memory text buffer:

#include

Full details of both these variations are given in the **Language Summary**.

#includes can only be nested once, because of limited file buffer space. So from direct entry level it is possible to #include a program in the editor's RAM which in turn uses #include filename for a header file or for functions, but these files may not use #include. Please see **Chapter** 1 for details of the maximum nesting of includes.

12.3 Conditional compilation

Conditional compilation control lines are not implemented. The library search facility provides a way to achieve conditional compilation.

12.4 Line control

#line control lines are not implemented.

12.5 Listing Control (NEW SECTION)

There is an additional facility in **HiSoft C** to control the production of a listing by the compiler. The compiler will normally produce a listing but:

#list-

will turn it off. To turn it back on use:

#list+

These commands nest, so that as long as an included file has an equal number of #list+ and #list- lines it will not affect the listing of the program that includes it.

12.6 Error Message Sacrifice (NEW SECTION)

The compiler produces explanatory error messages when it detects an error in the program. These messages are held in memory. Larger programs can be compiled if this memory is released to the compiler, and there is a compiler control line to do this:

#error

After this, errors will produce only the error number until the compiler is reloaded.

12.7 Direct Execution (NEW SECTION)

There is an additional feature in CPC **HiSoft C** that permits the direct execution of statements compiled by the compiler. This is sometimes called immediate execution and is similar to typing a command in BASIC without a line number. A preprocessor control line is used to enable this feature:

#direct+

and to disable it:

#direct-

When in direct mode a sequence of statements is compiled, instead of the normal sequence of external definitions. Note also that global variables will *not* be initialised to zero. Full details are in the **Language Summary**.

12.7 Stand-Alone Programs (NEW SECTION)

The compiler normally compiles a program and expects it to be executed with the compiler still resident in memory. But it also caters for programs which are intended as "stand-alone" programs which are loaded and run without the necessity to load the compiler first. See the **Language Summary** for more details.

13. Implicit declarations

14. Types revisited

14.1 Structures and unions

14.2 Functions

14.3 Arrays, pointers, and subscripting

14.4 Explicit pointer conversions

Pointers are 16-bit Z80 memory addresses and can be converted to integers which are also 16-bit. There are no alignment restrictions since the Z80 is a true byte addressed machine. Pointers are stored in the normal Z80 way with the least significant byte at the lower address.

15. Constant expressions

The **HiSoft C** compiler evaluates expressions at compile-time rather than at run-time when it recognises them to be constant. It recognises individual constant operands and it recognises that the result of an operator is constant when all its operands are. But it does not rearrange expressions so that while it will partially evaluate `( 1 + 2 + i )` at compile-time, leaving `( 3 + i )` to be calculated at run-time, it will not evaluate any of `( i + 1 + 2 )` but will calculate `( ( i + 1 ) + 2 )` at run-time.

The compiler accepts as a case constant, or as an array bound, or as an inline, or as an initializer any expression which it recognises as a constant expression.

16. Portability considerations

Function arguments are evaluated left to right. Character constants can only be a single character. The characters in a string are allocated in order of increasing memory address, as are the elements of an array. You are urged not to rely on these features except where absolutely necessary. Details of many implementation considerations are given to increase understanding and for those uses where it is essential. But that is not an invitation to rely on them. For example it is stated above that function arguments are evaluated left to right: they may not be in the next version of the compiler.

17. Anachronisms

The compiler does not support these anachronisms.

4.2 Low-Level Interface

This section gives some details of the code produced by the compiler, with particular reference to the store layout and use of the machine. It is intended to help you interface other programs to C programs and in particular to help you make use of the inline statement.

4.2.1 Source Format

C source is basically just a string of ASCII characters, divided into lines by NEWLINE characters (10, often known as LINE-FEED). There are no CARRIAGE-RETURN characters (13), although these will be accepted in a source file by the compiler, so other editors and even other computers can be used as tools with which to produce **HiSoft C** programs.

4.2.2 ZX Spectrum File Format

Files are provided at the character level (ie fopen, getc, putc, fclose).

On cassette, files are stored as a sequence of data blocks which follows a header block. The header block is a normal Spectrum header. The data blocks are each 514 bytes long, comprised of a character count in the first two bytes and up to 512 characters in the remainder of the block. The top bit of the character count is set to indicate the last block in the file.

Normal Spectrum CODE files are used on Microdrive, except that a start address and length of 0 are used. This means that most of the extended cataloguing programs available for the ZX Spectrum microdrives will return a file length of zero.

The compiler, and compiled programs, can open any type of Microdrive file for reading. This is done automatically.

4.2.3 Amstrad Cassette and AMSDOS File Format

Normal Amstrad file formats are used throughout. This section applies to cassette files and to disc files written under AMSDOS; details of the CP/M version will be given in the next section.

Source files produced using the editor are unprotected ASCII files, as are files written by C programs. An object file written by the #translate command is an unprotected binary file.

The compiler, and compiled programs, can open any cassette or AMSDOS file for reading. This is done automatically.

4.2.4 CP/M File Format

Normal CP/M file formats are used throughout. Both text and binary access to files is allowed as described in **Chapter 1** and in **Chapter 5**. Random access file handling is provided by seek() and tell() which are described in **Chapter 5**.

4.2.5 Function Linkage and the Stack

The Z80 processor stack (SP) is used for function linkage and local variables.

The caller evaluates each argument in left-to-right order and pushes them on the stack in turn before calling a function (so the last argument is on the top of the stack). *There must be exactly as many arguments as the function expects.* The caller then enters the function with a CALL instruction to the start of the function.

The called function then takes over and it first pushes the IX register on the stack and loads IX with the current value of the stack pointer (SP). Space is then allocated for any automatic local variables by decrementing SP. The function now executes, using IX to access the arguments and its locals. Finally it recovers the previous value of IX (for the caller), salvages the return link, and discards the local variables, linkage, *and arguments*, from the stack.

The result of the function is returned in HL, and also in BC.

There is a variation on this mechanism which is used for variadic functions (ie those that take a variable number of arguments). In this case, after pushing all the arguments on the stack, the caller pushes the total number of bytes of arguments including two bytes for the total itself. This total appears as the last argument to the called function. The called function can use the total to access its other arguments and *must* use it to discard the arguments from the stack before returning.

4.2.6 Register Usage

Neither the compiler nor the compiled code use the alternate register set, the IY register, or the I or R registers. At least some of these registers *are* used by the ROM firmware on many computers. The alternate register set is used by the Amstrad and the Spectrum; and on the Spectrum the IY register must always point at ERR_NR.

The compiler and the compiled code run with interrupts enabled.

The stack pointer (SP) is used normally and stack discipline should be observed. The IX register is used as a frame pointer as described above. The HL register is used to return the value of expressions and particularly function results. The BC, DE, A and F registers are used as general working registers.

4.2.7 Data Storage

There are three kinds of data storage: constant, fixed-address, and stack-based.

Storage for constants is allocated inline with the generated code; this includes numbers, characters, and strings.

Fixed-address storage is used for global (external) variables and for all static variables. This storage is allocated starting at the top of memory at RAMTOP and working downwards. The stack is below the fixed-address storage, and is moved down when necessary to keep it so. Fixed-address storage is accessed directly by addresses in the compiled instructions.

Stack-based storage is used for automatic local variables, arguments and function linkage, and temporary working store. It is allocated on the processor stack and accessed using SP and IX.

Storage for individual variables is allocated in the same way, regardless of whether stack-based or fixed-address storage is used. The allocation is shown below, firstly storage for the basic types, then for derived types:

char	1 byte.
int	2 bytes, least significant byte at lower address.
unsigned	2 bytes, least significant byte at lower address.
pointer	2 bytes, least significant byte at lower address. Contains address of pointed-to object.
array	n * s bytes, where n is the array bound and s is the size of each element. The first element (`a[0]`) is at the lowest address. A multi-dimensional array such as `a[m][n]` is treated as an array with bound m of arrays with bound n of the elements. So in the case of a character array element `a[i][j]` is at address `a + i*n + j`.

structure s bytes, where s is the sum of the sizes of all the members of the structure. The first member of the structure to be declared is at the lowest address (as described in Kernighan & Ritchie).

union s bytes, where s is the maximum of the sizes of all the members of the union. All members will be aligned at the lowest address (ie any spare space for a particular member will be at the high end of the union).

4.2.8 Spectrum Memory Layout

The compiler and stand-alone programs translated using the compiler all load at 25200 and are entered there also. This address leaves space below the compiler for two Microdrive channels, or one Microdrive channel and a cassette pseudo-channel. There is also room for a *very* small BASIC program.

The compiler will use store up to RAMTOP, and it moves RAMTOP down beneath itself for protection. This means that it is necessary to CLEAR to a larger value if you want to use the Spectrum for BASIC again after leaving the compiler.

Translated programs use the memory between 25200 and the RAMTOP value which was set before using the compiler to translate the program. However, these programs do not move RAMTOP themselves.

4.2.9 Amstrad Memory Layout

The compiler and translated programs load at the bottom of memory. Amsoft and Locomotive recommend that ROMs and RSXs do not use the option of taking low memory workspace, so this should not cause any conflicts.

The compiler obtains its high memory limit from the firmware by calling KL_CHOKE_OFF followed by KL_ROM_WALK.

4.2.10 CP/M Memory Layout

The compiler and translated programs are normal CP/M transient programs. The compiler assumes a conventional CP/M system where the transient program area (TPA) begins at location 0x100. The compiler and all compiled programs start at this location.

There is a runtime library at the base of the compiled code. This contains such things as 16-bit multiply and divide, printf and scanf, and support for input-output to files and other devices.

After this comes the compiled Z80 machine code of your program.

Next is initialisation data (if you have used initializers). This is stored in the .COM object file and loaded with the program. It is used once as the program starts up and the memory may then be used for other purposes (eg for heap or global data). The program *cannot* be rerun except by reloading from disc if the initialisation data area has been reused.

The area above this is used for the global variables, the stack, and the heap (ie alloc and free). The way in which it is used depends on whether or not a #data directive was used in the program.

If *no* #data directive was used then the compiler has to make an assumption about the location of the global variables. It assumes that the program will be run on the same system with the same size of TPA (so don't try to single step it with MON80 or DDT because they reduce the TPA size!!). The globals are placed at the top of the TPA, and the runtime stack starts just below the globals and grows down towards the code and intializer data.

The top of the TPA is read from location 6 at compile time by the compiler and is also used as the top of its own workspace. It is necessary for the compiler to know the address of the top of the global data area when it starts compiling. This is a consequence of the compiler's internal organisation, and partly accounts for the speed of both the compiler and the compiled code. But it can cause difficulty when trying to compile a program for debug using a monitor such as MON80, and when trying to compile to run on as many CP/M systems as possible. In both these cases it is important that the program uses as little memory as possible, leaving high memory free.

The way to achieve this is to use #data, which is a compiler directive to set the top of available memory explicitly. First compile the program without bothering about this problem and then recompile it using #data. The required address can be found by using the information shown in the memory usage report which is produced by the first compilation. It will not be necessary to repeat the double compilation very often because the address will not usually change very much.

The compiled program is still able to make use of the whole of the memory in the TPA automatically (whether it is an unknown system or a TPA reduced by a monitor in high memory).

If a #data directive is used then the runtime memory is organised a little differently. The global variables are placed where directed as explained above; and the compiler causes the runtime stack to be placed above this area rather than underneath it. It does this because the globals will have been placed as low as possible and whatever area of memory remains is now between the top of the globals and the top of the TPA (wherever that may be on the target system). The program determines the actual size of the TPA available to it on each occasion that it is run (by looking in location 6) and sets the stack to that address.

The memory layout both with and without a #data directive are illustrated in **Chapter 1.**

4.3 CP/M Input-Output System Buffers

The input-output system used with the CP/M version of the compiler requires an area of store for use as file buffers and control blocks. The area is allocated at the base of the stack just as the compiler or compiled program is starting up. The size of this area of store depends upon the number of streams (files or CP/M devices) which are opened by the compiler and/or the compiled program.

It is possible to alter the size of this area by patching the binary object file and instructions for doing this are presented after we explain how to calculate the size. Storage should be allowed for each stream as follows:

disc files (eg #include)	580 bytes
TRM: line buffered keyboard input (eg stdin)	160 bytes
other CP/M device (eg stdout, stderr)	30 bytes

Storage is allocated when a file is opened and is only reclaimed when files are closed in the reverse order to that in which they were opened. The compiler does this and most programs can be arranged to do so. For example the left hand sequence below needs i/o space of 1160 bytes whilst that on the right requires 1740 bytes.

```
f = fopen("FILE1", "r");
g = fopen("FILE2", "w");
copy_file(f, g);
fclose(g);
fclose(f);
h = fopen("FILE3", "a");
```

```
f = fopen("FILE1", "r");
g = fopen("FILE2", "w");
copy_file(f, g);
fclose(f);
fclose(g);
h = fopen("FILE3", "a");
```

The compiler is supplied with a default file area of 1960 (0x7A8) bytes calculated as:

```
stdin + stdout + sterr  + 3 levels of include file nesting
160 +  30   +   30   + 3 * 580
```

This size is stored in the compiler at location 0x133, but as a negative number -1960 (0xF858) and the size allocated as I/O space can be altered just by patching this number with a different value. HiSoft MON80 is a very useful program to perform this patch or a combination of the CP/M commands DDT and SAVE can be used.

The default size of 1960 bytes also applies to compiled programs so that by default they will be able to open 3 disc files simultaneously as well as the default console streams stdin, stdout, and stderr.

HiSoft C

Fast Interactive K&R C Compiler

Chapter 5

Standard Function Library

HiSoft

High Quality Software

5. THE HISOFT C STANDARD FUNCTION LIBRARY.

To a most dangerous C; the beauteous scanf

Shakespeare

This chapter of the Hisoft C Reference Manual describes the functions that are provided with the compiler.

The function library provided with a C compiler is very important since it adds to the power of the language. This library, like those of most other compilers, is patterned after that of the Unix C compiler. Many of the functions are also described in Kernighan & Ritchie (some in great detail). The library also serves to illustrate some of the features of the language, and of course includes many functions which take advantage of the graphics and sound capabilities of your computer.

The descriptions of related functions are grouped together and there is also an index of all functions. The library comes in several parts: the built-ins, the header, the standard library, and the BASIC library.

The built-ins are functions which are in the run-time package for efficiency and are therefore always contained in your program and can simply be called. The most important built-in is "printf" - for formatted output.

The header is a C source file called "stdio.h" provided on tape or disc. It contains constant and type definitions for the library and also contains the "min" and "max" functions. It should be included (#include "stdio.h") at the start of all programs which use the library.

The standard library is also supplied as C source in the file "stdio.lib" on tape or disc. It contains the source of most of the machine-independent library functions. These functions are all patterned after the Unix function library. This file should be selectively included at the end of each program which uses the library by means of a library-search control line (#include ?stdio.lib?).

The function library for the HISOFT-C compiler will continue to grow and become more powerful with time. One of the best ways for the library to grow is by new functions created by the people who use the compiler. If you write functions that you think will be useful to other people then send them to us. We will collect these functions and distribute them on low-priced library cassettes so that they can be made widely available. Some of these library functions may also be distributed in "stdio.lib" with the compiler. An up-to-date copy of the library will be sent to everybody whose functions are included in the library. If you wish to contribute to the library, please send a cassette containing the C source of your functions and the documentation for them (because we don't have enough time to type it all in!). Ideally, send a printed copy as well. Put your name into the documentation and as a comment in the source. You can put your address as well if you are happy to get comments from other users.

Arithmetic functions

5.1 int max(n, ...) auto

Returns the value of the greatest of its integer arguments. The function takes any number of arguments (it is "variadic") as indicated by the word "auto".

5.2 int min(n, ...) auto

Returns the value of the smallest of its integer arguments. The function takes any number of arguments (it is "variadic").

NB - min and max are in "stdio.h" because they take any number of arguments; and so need to be declared BEFORE they are used. If you include "stdio.h" these functions will always be compiled into your program. You should make a version of "stdio.h" without them (by using the editor) if you want to compile a lot of code and don't need them.

5.3 int abs(n)

Returns the absolute value of its argument.

5.4 int sign(n)

Returns -1 if the argument is less than zero, 0 if the argument is zero, and 1 if the argument is greater than zero.

An Illustration of How to Grub Around

peek and poke are provided to show how to access absolute locations in store from C programs. It is often possible to write specific functions for a particular program which are more efficient and easier to use. See the example "dump" function in chapter 1 for instance. A useful technique is to define a C structure which represents the layout of the piece of store being used and then assign the address of the store area to a pointer to the structure. These routines do this for the simple case of a single byte.

5.5 char peek(address)

Returns the value of the byte of store at location "address".

5.6 void poke(address, value)

Puts the low eight bits of "value" into store at location "address". Note that the function has no result, which is denoted by the "void" type.

Simple input and output to ports is also provided. Note that the Z80 instructions "in r,(c)" and "out (c),r" are used so that full 16 bit port addresses can be used on those machines which require them.

5.7 int inp(port_number)

Returns the 8 bit input value from the i/o port specified by the 16 bit "port_number".

5.8 int out(data, port_number)

Sends the bottom eight bits of "data" to the i/o port specified by the 16 bit "port_number".

Format conversion routine - ASCII to binary integer

5.9 int atoi(s)

```
char *s;
```

Scans the string "s" and returns the binary value of the ASCII number in it. The function first scans over any whitespace (space, tab, or newline characters) and then converts the number. The conversion stops when it finds the first non-digit character. The value 0 is returned if no number is found. The number may have a '+' or a '-' sign in front of it.

Sorting function - a Shell sort

5.10 void qsort(list, num_items, size, cmp_func)

```
char *list;
int  num_items, size;
int (*cmp_func)();
```

Sorts a list of items into ascending order using a Shell sort. (The function is called qsort because the Unix original used Hoare's quicksort). The items are all the same size - "size" bytes long. There are "num_items" of them. They appear one after the other starting at "list". "cmp_func" is a pointer to a function which will compare two items in the list. For example the standard function "strcmp" can be used if the items are strings. The function should take two pointer arguments so a call looks like:

```
(*cmp_func)(x,y);
```

and the function should return an integer:

```
<0  if  *x <  *y
 0  if  *x == *y
>0  if  *x >  *y
```

A common structure for the list is a two-dimensional array, num_items long and size bytes wide:

```
char   list [num_items] [size];
```

The function is described in detail in Kernighan & Ritchie.

String Handling Functions

Remember that strings in C are arrays of characters which "end" at the first zero byte. The array may well have more physical store after the end. Equally, if the array is not big enough then whatever follows the array in memory will be overwritten. YOU HAVE BEEN WARNED.

5.11 char *strcat(base, add)

```
char *base, *add;
```

Inserts a copy of the string "add" at the end of string "base". This is a physical copying and it is your responsibility to make sure that there are enough bytes at the end of "base" to take the copy of "add"; otherwise whatever is next will be overwritten!! The function returns a pointer to the start of the "base" string as its result.

5.12 char *strncat(base, add, number)

```
char *base, *add;
int    number;
```

Behaves like strcat except that it copies at most "number" characters. The resulting string is null-terminated.

5.13 int strcmp(s, t)

```
char *s, *t;
```

Compares two strings, byte for byte, and returns 0 if the two are identical. It returns a value >0 if s>t and a value <0 if s<t. A string is greater if the first character that differs is later in ASCII code sequence.

5.14 int strncmp(s, t, n)

```
char *s, *t;
int  n;
```

Behaves like strcmp except that it checks at most the first "n" characters of the strings. It stops earlier if either of the strings are shorter.

5.15 char *strcpy(dest, source)

```
char *dest, *source;
```

Makes a physical copy of the "source" string in the "dest" string.

5.16 char *strncpy(dest, source, number)

```
char *dest, *source;
int  number;
```

Copies exactly "number" characters into "dest". If "source" contains less than "number" characters, then it is copied in its entirety to the beginning of "dest", and the remaining characters are filled with nulls (ie zero). If "source" contains "number" or more characters, then the first "number" are copied to "dest", and "dest" is NOT null-terminated (ie its not really a string anymore, just a character array).

5.17 unsigned strlen(s)

```
char *s;
```

Returns the length of a string. That is the number of characters before the terminating zero.

5.18 char *strchr(string, ch)

```
char *string, ch;
```

Returns a pointer to the first occurrence of the character "ch" in the "string" or NULL (zero) if the character does not occur. You can get a pointer to the end of a string by looking for the NULL character:

```
pointer_to_end = strchr(string, 0);
```

You can use this function in many situations where you would use the SET type in Pascal. For instance to loop round whilst hexadecimal digits are input we might write:

C

```
while (strchr("0123456789abcdefABCDEF", (character = getchar())))
   do_something_with_character() ;
```

Pascal

```
while input^ in [ '0'..'9', 'a'..'f', 'A'..'F'] do
   begin
      DoSomethingWithCharacter ;
      get(input)
   end ;
```

Of course C provides a much neater way of solving this particular problem.

```
while (isxdigit(character = getchar()))
   do_something_with_character() ;
```

5.19 char *strrchr(string, ch)

```
char *string, ch;
```

Behaves like strchr except that it returns the last occurrence of "ch" in "string" rather than the first.

5.20 char *strpbrk(s1, s2)

```
char *s1, *s2;
```

Returns a pointer to the first occurrence in string "s1" of any character from string "s2", or NULL if no character from "s2" exists in "s1". For instance:

```
strpbrk("the quick brown fox jumps over the lazy dog", "wolf");
```

Will return a pointer to the place marked with a '^'.

5.21 **int strspn(s1, s2)**

```
char *s1, *s2;
```

Returns the length of the initial segment of string "s1" which consists entirely of characters from string "s2".

5.22 **int strcspn(s1, s2)**

```
char *s1, *s2;
```

Returns the length of the initial segment of string "s1" which consists entirely of characters NOT from string "s2".

Character Test and Manipulate Functions

5.23 **int isalnum(c)**

```
char c;
```

Returns TRUE (ie 1) if the character is an alphanumeric (ie a letter or a digit) and returns FALSE (ie 0) if it is not.

5.24 **int isalpha(c)**

```
char c;
```

Returns TRUE if the character is a letter and FALSE if it is not. This function is built-in.

5.25 **int isascii(c)**

```
char c;
```

Returns TRUE if the character is ASCII (ie less than 0x80).

5.26 **int iscntrl(c)**

```
char c;
```

Returns TRUE if the character is a control character.

5.27 int isdigit(c)

```
char c;
```

Returns TRUE if the character is a digit. This function is built-in.

5.28 int isgraph(c)

```
char c;
```

Returns TRUE if the character is a graphic printing character (greater than space and less than 0x7F).

5.29 int islower(c)

```
char c;
```

Returns TRUE if the character is a lower-case letter ('a' - 'z'). This function is built-in.

5.30 int isprint(c)

```
char c;
```

Returns TRUE if the character is a printing one.

5.31 int ispunct(c)

```
char c;
```

Returns TRUE if the character is punctuation (ie printable and not a letter or a digit).

5.32 int isspace(c)

```
char c;
```

Returns TRUE if the character is whitespace. That is, if it is the space character, the newline character or the tab character. This function is built-in.

5.33 int isupper(c)

```
char c;
```

Returns TRUE if the character is an upper-case letter ('A' - 'Z'). This function is built-in.

5.34 int isxdigit(c)

```
char c;
```

Returns TRUE if the character is a hexadecimal digit ('0' - '9' or 'a' - 'f' or 'A' - 'F').

5.35 char tolower(c)

```
char c;
```

If the character is an upper-case letter then it returns its lower-case equivalent, and otherwise it returns the character unchanged. This function is built-in.

5.36 char toupper(c)

```
char c;
```

If the character is a lower-case letter then it returns its upper-case equivalent, otherwise it returns the character unchanged. This function is built-in.

5.37 char toascii(c)

```
char c;
```

Forces the character into the range 0x00 to 0x7F by "and"-ing it with 0x7F.

Storage Allocation and Freeing (Heap Management)

These functions are explained in detail in Kernighan & Ritchie. Note that there is a small control region allocated in "stdio.h" for use by these functions (it is the head of the free-store chain).

5.38 char *calloc(n, size)

```
unsigned  n, size;
```

Allocate space for "n" items of "size" bytes each. It returns a pointer to the start of the space or else it returns NULL if there is no space. For example:

```
p = calloc(100, sizeof(int));
```

allocates 200 bytes, enough for an array of 100 integers. There are actually some more bytes hidden before the block which are used by "free" when the block is finished with. These hidden bytes must not be changed. The function corresponds to "new" in Pascal.

5.39 void free(block)

```
char *block;
```

Return a block of store to the free-store chain for re-allocation later by "calloc". You must return (a copy of) the pointer supplied by "alloc" when the storage was obtained, and the hidden bytes must be intact. The function corresponds to "dispose" in Pascal.

5.40 char *sbrk(n)

```
unsigned n;
```

This is another function associated with storage allocation, and the reasons for its existence are a little obscure. What it does is to allocate n bytes of physical memory for use by calloc(). It is not normally called from anywhere else. Why is this extra function necessary, when calloc could do it directly? Well, it allows YOU to decide which area of memory should be used to provide the space for the heap. On Unix and many other larger systems sbrk calls the operating system which makes memory available by moving things around or by taking it from another user (etc etc). On smaller computers it is up to you to decide on a safe area of memory to be used. There are several possible places to get the memory, and we have included an sbrk() function which uses the safest of these. You can rewrite it if you want to use one of the other places.

Miscellaneous Functions

5.41 void swap(p, q, length)

```
char *p, *q;
unsigned  length;
```

This function swaps the contents of two areas of store each "length" bytes long and pointed to by "p" and "q". It is used by "qsort" in particular. This function is built-in.

5.42 void blt(dest, source, length)

```
char *dest, *source;
unsigned  length;
```

This function moves the contents of the area of store starting at "source" into the area starting at "dest". It moves "length" bytes. The copy is done in the non-destructive direction if the areas of store overlap (ie starting at the low end if dest is below source and the high end if dest is above source). The strange name of the function stands for "BLock Transfer"; it arises from Unix history. This operation is similar to "strcpy" but it always copies the given number of bytes including any zero bytes it finds. This function is also faster because it is built-in.

INPUT - OUTPUT FUNCTIONS

INPUT - OUTPUT FUNCTIONS

These functions implement C/UNIX type file input-output. They are all similar to those described in Kernighan & Ritchie, and thus that is a good place to find more details.

There are three main groups of functions: the character-level functions, the complex-level functions and the raw-level functions. The character-level functions are those most used in C programs, they provide buffered input and output of single characters. The complex-level functions use the character-level functions to provide more facilities ranging from output of a string (puts) to flexible formatted printing (printf). The raw-level functions provide a direct interface to the facilities available in the computer.

Input-output in C is done via "files" which is a fairly general concept. Files are "just a stream of bytes" which can be read using getc or written using putc. There are three standard files which all C programs have: these are the standard input stdin, the standard output stdout, and the error output stderr. stdin is usually the keyboard and the two output files are the screen. A program can also open other files on cassette, disc, microdrive etc as available. All the files can only be accessed one character at a time (serial access).

Character-level Input-Output Functions

HiSoft C
CPC SPECIFIC

5.43 FILE * fopen(name, mode)

```
char *name, *mode;
```

Opens a file for character-level input-output. The string name is the name of the file to be opened and the string mode tells whether the file is to be read or written. The file will be opened for reading if the string is 'r' and it will be opened for writing if the string is 'w'. It is not possible to append to an existing file, and if you open an existing disc or Microdrive file for writing it will first be *erased. Be careful* - mode *is a string not a character.* fopen returns a file-pointer for use with the other functions to tell them which file to write to or read from. The file-pointer will be NULL (ie 0) if there is any error. An example of the use of this function can be found in the editor example section at the end of **Chapter 2.** This function is built-in.

HiSoft C
CPC SPECIFIC

5.44 int fclose(fp)

```
FILE *fp;
```

Close the file indicated by the file-pointer fp. If the file is being written to this ensures that the last block of data is written. If the file is being read there is no action on the device. In both cases the control and buffer storage becomes available to open another file. An example of the use of this function can be found in the editor example session at the end of **Chapter 2.** This function is built-in.

5.43 FILE * fopen(name, mode)

```
char *name, *mode;
```

Opens a file (or a device) for input and/or output. It returns a file-pointer for use with the other functions to tell them which file to write to or read from. The file-pointer will be NULL (ie 0) if there is any error.

The name string is the name of the file to be opened. It can be the name of a real file or one of the predefined device names listed in **Chapter 1**.

The mode string specifies the kind of access required to the file. *Be careful* - mode *is always a string, not a character.* The mode string is made up of a sequence of characters as follows. The first character in the string must be one of:

'r' open for reading.
'w' create a new file for writing.
'a' append (ie open for writing at the end of a file or create a new file).

The next character may be a 'b' which specifies that the file should be opened for binary access rather than text access which is the default. Finally, a '+' character may be the last character in the string, which allows both read and write access to the file.

If you open a file for writing with "w", "wb", "w+", or "wb+" any existing file with that name will first be erased. *Be careful when using append; refer to* ***Chapter 1.7.*** This function is built-in.

5.44 int fclose(fp)

```
FILE *fp;
```

Close the file indicated by the file-pointer fp. If the file is being written to this ensures that the last block of data is written. If the file is being read there is no action on the device. This function is built-in

5.45 int getc(fp)

```
FILE *fp;
```

The basic character-level input function. It reads the next character using the file-pointer fp. It returns EOF (-1) if the end of the file has been reached. *Note that it returns an integer, not a character.* If the result is assigned to a character variable EOF will never be seen because the top byte is thrown away, leaving +255 instead of -1!! This function is built-in.

5.46 int ungetc(c, fp)

```
int     c;
FILE    *fp;
```

This function puts the character c back onto the file fp, so that it is the next character to be read with getc() (or getchar()). There are a couple of points to note about the use of ungetc: you can only put one character back at a time on each file; and scanf uses ungetc so you cannot use ungetc after scanf without an intervening call to getc. This function is built-in.

5.47 int putc(c, fp)

```
int     c;
FILE    *fp;
```

The basic character-level output function. Sends the character c using the file-pointer fp. It returns the character as its result also. This function is built-in.

5.48 int getchar()

Get a character from the standard input - stdin. This function does buffered input from the keyboard. Incoming characters are collected into a line buffer until [ENTER] is pressed. The [DELETE] key can be used to edit the characters as they are typed. A cursor is displayed and characters are echoed on the display as they are typed. This function is built-in.

5.49 int putchar(c)

```
int c;
```

Put a character to the standard output - stdout. The character is displayed on the screen. The function returns the character as its result also. This function is built-in.

Complex-level I-O Functions

This section continues on page 101.

This page is intentionally left blank.

5.50 void exit(n)

This function is a mixture of I-O function and system function. It closes all files which the program has open and then exits from the program by calling "_exit" (see below). The parameter "n" is passed out as the result of the program and it is used to indicate whether or not the program was successful.

A return value of 0 means success, other values indicate an error on the Spectrum by causing the corresponding Spectrum error report to be displayed. On the Amstrad computers, a message is displayed and the compiler or translated program is restarted.

5.51 char *fgets(s, n, fp)

```
char *s;
int  n;
FILE *fp;
```

Read string "s" from file-pointer "fp". The reading will stop when a NEWLINE character is read or when "n"-1 characters have been read, whichever occurs first. (So "n" is the size of "s"). The string will be terminated by a '\0' character which is added after the newline character. The return value is normally "s" but if the end of file has already been reached when "fgets" is called then the return value is NULL (0).

5.52 void fputs(s, fp)

```
char *s;
FILE *fp;
```

Outputs the string "s" to the file-pointer "fp".

5.53 char *gets(s)

```
char *s;
```

Reads string "s" from the standard input (the keyboard). It is similar to fgets except that it has no maximum character count, and also the newline character is OVERWRITTEN by the terminating '\0'.

5.54 void puts(s)

```
char *s;
```

Outputs the string "s" to the standard output (the display) and appends a newline character to it.

5.55 void printf(control, arg1, arg2, ...)

```
char *control;
```

This is the most important output function, as it is used for almost all kinds of printing - text, numbers, characters, strings etc. "printf" converts, formats, and prints its arguments on the standard output "stdout" under control of the string "control". It behaves as described in Kernighan & Ritchie. The control string is printed as it stands except that all conversion-specifications in it are used to print the other arguments.

A conversion specification starts with a '%' character, then follow some optional modifiers and finally the conversion character. All the conversion specifications are supported (except the floating point ones):

d	signed decimal number.
o	unsigned octal number.
x	unsigned hexadecimal number.
u	unsigned decimal number (eg store address).
c	a single character.
s	a string terminated by a '\0' character.

The character specification modifiers are also all supported:

-	left justify field (default is right justify).
0	(a leading zero on following field) use '0' instead of ' ' for padding.
999	(a digit string) minimum field width.
.999	the precision - max number of characters from a string.
L	long data - has no effect.

Note that a % character is printed by putting %% in the control string. This function is built-in.

5.56 void fprintf(fp, control, arg1, arg2 ...)

```
FILE *fp;
char *control;
```

Behaves like "printf" except that output is performed using the file-pointer "fp" instead of "stdout". This function is built-in.

5.57 void sprintf(s, control, arg1, arg2 ...)

```
char *s;
char *control;
```

Behaves like "printf" except that output is placed in the string "s" instead of being sent to "stdout". The character array pointed to by "s" must be large enough to receive the output or store will be overwritten. This function is built-in.

5.58 int scanf(control, arg1, arg2 ...)

```
char *control;
ALL OTHER args MUST BE POINTERS;
```

This function is the input analogue of printf, providing many of the same conversion facilities in the opposite direction. It reads characters from the standard input, interprets them according to the format specified in control and stores the results in the remaining arguments WHICH MUST ALL BE POINTERS!!! The control string usually contains conversion specifications which are used to direct interpretation of input sequences. It may contain:

- Blanks ("white space") which are matched by any amount of white space in the equivalent position in the input stream. Any amount is from zero to an indefinite maximum.

- Ordinary characters (not %) which must match the next input character.

- Conversion specifications consisting of the character %, an optional assignment suppression character *, an optional number specifying a maximum field width, and a conversion character.

A conversion specification determines how the next input field is interpreted. Normally the result is placed in the variable POINTED TO by the corresponding argument. This means that the argument usually starts with an & operator. If assignment suppression is selected by the * character then the input field is simply skipped and no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. Hence, [ENTER], the [TAB] character or [SPACE] can be used to separate input fields to scanf.

Note however that unlike the scanf described in Kernighan & Ritchie, white space in the input field will only be accepted when it matches a white space character in the control string. This is so that a control string, say "alpha", will not match an input string with embedded space, say "al pha".
This was an oversight which made the original scanf act rather unintelligently in certain circumstances, but later versions of Unix C follow the same form as we have here. The conversion character indicates the interpretation of the input field and the corresponding argument must be a pointer to satisfy the 'call by value' rules of C. The following are the legal conversion characters:

d A decimal integer is expected; the corresponding argument should be an integer pointer.

o An octal integer, with or without the leading zero, is expected; the corresponding argument should be an integer pointer.

x A hexadecimal integer, with or without the leading 0x, is expected; the corresponding argument should be an integer pointer.

h A short integer is expected; in this implementation the corresponding argument should be an integer pointer.

c A single character is expected; this reads the next input character regardless of whether it is white space or not and assigns it to the char variable pointed to by the corresponding argument.

s A character string is expected; the corresponding argument should be a pointer to a character array large enough to hold the string and the terminating \0 which will be added.

The conversion characters d, o and x may be preceded by 'l', which in compilers supporting long integers would mean that the argument points to a long integer rather than a normal integer. Here it is ignored. The function returns when the control string is exhausted or when some input does not match the control specification. It returns as its value the number of successfully assigned input items, or EOF if end of file was reached first. A short example of scanf is shown here:

```
int  n;
char    s[20];
scanf(" %d  , %19s", &n,  s);
```

This will read an integer into n (&n POINTS to n) and a string (which must not be longer than the array s) into s. The two fields must be separated by a comma (and maybe some white space). Note the use of the maximum width 19 to protect against an overlong input string. Note that there was no ampersand (&) preceding 's' in the scanf call as s is already a pointer to the array. Also note the leading space in the control string which will match (and discard) white space on the input if there is any. This is particularly important to match the [ENTER] on the end of previous input to scanf.

5.59 int fscanf(fp, control, arg1, arg2 ...)

```
FILE *fp;
char *control;
```

This function behaves just like 'scanf' except that its input is obtained from the file attached to fp rather than stdin.

5.60 int sscanf(s, control, arg1, arg2 ...)

```
char *s;
char *control;
```

This function behaves just like 'scanf' except that it uses the string pointed to by s as its source of input.

Raw-Level I/O Functions

5.61 int rawin()

Inputs a character directly from the keyboard, with no conversion of character codes. There is no cursor and nothing is echoed to the display. This function is intended for special applications such as games. It is built-in.

On the Spectrum it waits for bit 5 of the FLAGS system variable to be set, then reads LAST_K and resets the flag.

On the Amstrad machines it uses KM_WAIT_CHAR.

5.62 int keyhit()

Tells whether a key has been pressed on the keyboard, returning TRUE (1) if so and FALSE (0) if not. The function does not read the key, and if it returns TRUE then you must read the keyboard (using rawin() perhaps) to get the key and reset the keyboard before you try to use keyhit() again. If you do not, it will continue to return TRUE every time. This function is built-in.

On the Spectrum it uses the ROM KEY_SCAN routine.

On the Amstrad computers it uses KM_READ_CHAR followed by KM_CHAR_RETURN.

5.63 void rawout(c)

```
char c;
```

Outputs the character directly to the screen, with no conversion of character codes. It is designed to allow full control over the display screen. This function is built-in.

System Interface

5.64 void _exit(n)

This function immediately exits from the program and returns to the system. On the Spectrum the argument is printed as the corresponding BASIC report (eg _exit(0) is OK and _exit(4) is Out of memory). On the Amstrad a message is printed.

Some Functions for 32 bit number arithmetic

These functions are not intended to provide full facilities for long arithmetic but are here because they are needed by the random number generator. They can provide a base if 32 bit numbers are needed. The numbers are represented by an array of four characters (or a pointer to such an array). The least significant eight bits are held in array[0] and so on to the most significant eight bits in array[3]. The numbers are unsigned (this only affects the multiplication routine).

5.65 void long_multiply(c, a, b)

```
char *a, *b, *c;
```

Multiply two 32 bit numbers. c = a * b;

5.66 void long_add(c, a, b)

```
char *a, *b, *c;
```

Add two 32 bit numbers. c = a + b;

5.67 void long_init(a, n1, n0)

```
char *a;
unsigned n1,n0;
```

Initialise a 32 bit number. n1 provides the most significant 16 bits and n0 provides the least significant 16 bits. So for example after:

```
long_init(a, 0x1234, 0x5678);   "a" has the value   0x12345678
```

5.68 void long_set(a, n, d)

```
char *a;
unsigned n, d;
```

Initialise a 32 bit number. n provides 16 bits to initialise and d tells where to place them in the number. For example after:

```
long_set(a, 0x1234, 1);      "a" has the value 0x00123400,
```
and
```
long_set(a, 0x5678, 3);  gives "a" the value 0x78000000
```

5.69 void long_copy(c, a)

```
char *a, *c;
```

Copy one 32 bit value to another place. Equivalent to c = a;

Pseudo-Random Number Generator

This generator is adapted from "Learning to Program in C" by Thomas Plum. It generates 16 bit numbers with a period of 2^32. So it generates a sequence of numbers which repeats itself after every 2^32 calls. Note that the same numbers will occur again and again in the sequence. No guarantees are given about the distribution of the random numbers, so don't be disappointed.

5.70 int rand()

Returns a 16-bit pseudo-random number.

5.71 void srand(n)

"Seeds" the generator. Used at the start of a program to begin at a different place in the sequence each time. This makes the program's behaviour change in different runs. If the same seed n is used then the sequence will be exactly the same as the previous time - this can be useful in some statistics programs.

Auxiliary Input-Output Functions

Although these functions are UNIX compatible it is necessary to be aware of the underlying operating system and the consequent implementation issues to derive maximum benefit in all circumstances. Well it's certainly easier to use them if you understand what's happening, so here goes!

5.72 General Points to Remember About CP/M Input-Output

The CP/M I-O structure provides several particular problems for an implementation of the UNIX-like C I-O. The short word-length and addressing range of the Z80 processor provide additional difficulties, similar to those found in some PDP-11 versions of UNIX.

Firstly, it has been necessary to differentiate between text access and binary access to a file, because of the difference between CP/M line and file end markers and those in UNIX. This is explained in **Chapter 1**.

Secondly, we are using a system of 16 bit numbers (-32768 to +32767 or 0 to 65535) and it is possible that file sizes can be bigger than this. This could cause a conflict between compatibility with UNIX and practicality under CP/M, particularly for the random access functions seek and tell. The solution we have adopted is to provide one basic built-in function for each and then to construct additional functions in the C source library. These are compatible with their UNIX counterparts having the same name. We have invented different names for extra functions needed to write practical programs in a Z80 CP/M environment. This approach also means that it is very easy for the user to customise a particular function to suit her own needs.

5.73 int read(stream, buffer, bytes)

```
FILE * stream;
char * buffer;
unsigned bytes;
```

This is a standard UNIX function. It reads the specified number of bytes from the given stream into the buffer whose address is supplied; and returns the number of bytes actually read. This will normally be the number of bytes requested but may be less if the End-Of-File is encountered first (or even zero if the stream had already reached the End-Of-File). The return value will be -1 (== ERROR == EOF) if an error occurred. Note that the stream parameter would be a file_descriptor and not a file_pointer on UNIX systems which make the distinction. This function is the building block used in fread(). The function is built-in.

5.74 int fread(buffer, item_size, num_items, stream)

```
char * buffer;
unsigned item_size, num_items;
FILE * stream;
```

This is a standard UNIX function. It reads the specified number of items, each of the given size, from the given stream into the buffer whose address is supplied. The number of items actually read is returned unless an error occurred in which case NULL (0) is returned. This function is not much different from read and is supplied mainly for compatibility.

5.75 int write(stream, buffer, bytes)

```
FILE * stream;
char * buffer;
unsigned bytes;
```

This is a standard UNIX function. It writes the specified number of bytes to the given stream from the buffer whose address is supplied; and returns the number of bytes actually written. This will normally be the number of bytes requested but will be -1 (== ERROR == EOF) if an error occurred. Note that the stream parameter would be a file_descriptor and not a file_pointer on UNIX systems which make the distinction. This function is the building block used in fwrite(). The function is built-in.

5.76 int fwrite(buffer, item_size, num_items, stream)

```
char * buffer;
unsigned item_size, num_items;
FILE * stream;
```

This is a standard UNIX function. It writes the specified number of items each of the given size, to the given stream from the buffer whose address is supplied. The number of items actually written is returned unless an error occurred in which case NULL (0) is returned. This function is not much different from write and is supplied mainly for compatibility.

5.77 int fflush(stream)

```
FILE * stream;
```

This is a standard UNIX function. It is used on an output stream to cause the buffer associated with the stream to be flushed. Flushing is used with disc files and means that the contents of the buffer are written to the disc immediately instead of waiting until the buffer is full. This is only necessary where it is important to guarantee the data against some catastrophe such as power failure, or where record locking is being used on a multiple access file. Sparing use of fflush() is recommended since the extra I-O makes a significant difference to the speed of a program. The use of fflush() does not affect the current position in the file, and the same data will be written to the file again when the buffer is filled. The function is built-in.

5.78 **int _seek(stream, hi_offset, lo_offset, mode).**

```
FILE * stream;
int    hi_offset, lo_offset, mode;
```

This function provides random access to a file. It moves the current position in the disc file associated with the given stream, according to the **offset** and **mode** parameters. This function is designed to allow immediate access to the facilities of the runtime system. The offset should be considered as a single signed 32 bit number, although it is presented as two 16 bit integers. The most significant 16 bits of the offset are given in **hi_offset** and the least significant 16 bits are in **lo_offset.** The file is repositioned in a way that depends on the value of mode.

mode

0 The current position is set **offset** bytes from the beginning of the file. So for example

```
_seek(stream, 0,0, 0);
```

will put a stream back to the start of the file.

1 The position is moved **offset** bytes from the existing position. Remember that offset is a signed number so that mode 1 can be used to re-examine the most recent part of an input file or to re-write some of an output file. The first example below ignores the next 10 bytes in a file, and the second one rewinds the file by one byte (approximately like ungetc).

```
_seek(stream, 0,10, 1);
_seek(stream, 0xFFFF,0xFFFF, 1);
```

2 The offset is measured from the end of the file, and should be negative or zero (an offset of zero allows appending to a file assuming writing to the file is permitted).

```
_seek(stream, 0,0, 2);
```

Seeking from the end of file is much more tricky under CP/M than it is under UNIX because of the inadequacy of the CP/M file system. In practice seeks relative to the end of file should only be made on well-behaved text files. The usual techniques used with binary files under UNIX and other real operating systems do not work under CP/M 2; and it will be necessary to keep a manual track of the exact file length if this is required. The comments made in section 7 of chapter 1 are particularly relevant here.

One final, but very important point - **the byte position within the file is the physical CP/M byte number** and so does not relate immediately to the count of characters seen by a C program using text mode access (because of the end-of-line handling). In UNIX parlance "the number returned from ftell is a **Magic Cookie** useful only for handing to fseek". So now we all know and are one step closer to the arcane mysteries! The function is built-in.

5.79 int seek(stream, offset, mode).

```
FILE * stream;
int    offset, mode;
```

This function also allows a file to be repositioned by use of an offset and mode (see _seek above). This is a UNIX function which was supplied with pre-version 7 UNIX systems. It is useful to us because the PDP-11 on which these systems ran also had a limited word length. The solution which was adopted works equally well here. In addition to modes 0,1, and 2 which work as described above; another three modes are supplied which work in the same way except that **offset** is multiplied by 512 first (the length of a PDP-11 disc block!). This enables offset to be a simple 16 bit value and yet still be able to seek the full length of any CP/M file. Modes 3,4, and 5 correspond to modes 0, 1, 2 respectively. So to seek backwards 16384 bytes from the current position in a file we can use:

```
seek(stream, -16384/512, 4);
```

5.80 int fseek(stream, offset, mode).

```
FILE * stream;
long   offset;
int    mode;
```

This is a standard UNIX function which is provided mainly for compatibility. It performs a seek (see _seek above) with modes 0, 1, or 2 and a **long** offset. It works exactly as it should, but is not able to seek the full length of a very large file because the offset is a signed 16 bit number (since long is sixteen bits).

5.81 int _tell(stream).

```
FILE * stream;
```

This function provides random access to a file. It returns the current position in the disc file associated with the given stream. This function provides immediate access to the facilities of the runtime system. The current position is returned as a 32 bit physical byte number in the de and hl registers, with the most significant 16 bits in de and the least significant 16 bits in hl. The value in de cannot be accessed in normal use, while hl contains the return value of the function. But the contents of de can be accessed by inline code immediately after a call to _tell; and this is done by functions below. The function is built-in.

5.82 long ftell(stream).

```
FILE * stream;
```

This is a standard UNIX function. It returns the current position in the disc file associated with the given stream. Because longs are 16 bits, the value returned is only useful for files of size less than 65536 bytes. Use tell32 (below) on larger files. This function uses _tell (qv).

5.83 **void tell32(stream, pos_ptr).**

```
FILE * stream;
char * pos_ptr;
```

This function returns the current position in the disc file associated with the given stream. The position is written into the four byte area addressed by pos_ptr and should be considered as a single signed 32 bit number. It is compatible with the format used by long_add etc and by regarding it as two 16 bit integers can be passed back to _seek. This is shown in the example below. Once again note that the position is a **physical byte number** and not a count of characters in text mode.

```
/* example of 32 bit file position manipulation */

typedef union  {
    char pos_bytes[4];

    struct  {
        int pos_lo;
        int pos_hi;
      } _word;
  }
    file_pos;

main()
{
  file_pos p, q, r;
  FILE * f;

  f = fopen("filename", "rb");          /* open a binary file                */
  do_some_reading_from(f);              /* part of the file is processed     */
  tell32(f, p);                         /* find out where we are in the file */
  long_init(q, 0x003B,0x5A71);          /* q = 0x3B5A71; its a big file!     */
  long_add(r, p, q);                    /* r = p + q; skip through the file  */
  _seek(f, r._word.pos_hi,r._word.pos_lo, 0);
  do_some_reading_from(f);              /* and resume processing later       */
}

/* in this case we could actually avoid the calls to tell32, long_init,     */
/* and long_add by using mode 1 in the call to _seek as follows:            */

main()
{
  FILE * f;

  f = fopen("filename", "rb");          /* open a binary file                */
  do_some_reading_from(f);              /* part of the file is processed     */
  _seek(f, 0x003B,0x5A71, 1);           /* skip through the BIG file         */
  do_some_reading_from(f';              /* and resume processing later       */
}
```

5.84 int fname(stream, buffer).

```
FILE * stream;
char buffer[15];
```

This function returns the name of the file which is associated with a stream. For example, it can be used to detect the actual destination of standard output which has been redirected. The name is copied into an actual buffer supplied by the user. It is your responsibility to ensure that buffer is indeed the address of an array of at least 15 characters. The buffer is filled with a normal C zero-terminated string representing the name of the file in the usual CP/M way as drive:filename.extension. A physical device name will be returned if appropriate (eg "CON:"). The function is built-in.

5.85 freopen(filename, mode_string, stream).

```
char *filename, mode_string;
FILE * stream;
```

This is a standard UNIX function which is used to redirect the input or output on an existing stream. It is commonly used to redirect the standard streams **stdin, stdout** and **stderr.** An example of its use can be found in the I/O redirection facilities implemented in the **cpm_cmd_line** library function. The function is built-in.

5.86 int getw(stream).

```
FILE * stream;
```

Returns a 16 bit integer read from the file associated with **stream.** The integer is made up from the next two bytes read from the stream with getc. The first byte read provides the low order 8 bits of the integer. This function is usually appropriate only to a binary file opened in "rb" mode and particularly to a file written using **putw** below.

5.87 void putw(w, stream).

```
int  w;
FILE * stream;
```

Writes the 16 bit integer w to the file associated with **stream.** The integer is written as two successive bytes using putc. The low order 8 bits are written first. This function is usually appropriate only to a binary file opened in "wb" mode.

HiSoft C

Fast Interactive K&R C Compiler

Chapter 6

The CPM.LIB Library

6. THE CPM.LIB LIBRARY FOR CP/M COMPUTERS.

You never enjoy the world aright,
till the C itself floweth in your veins,
till you are clothed with the heavens,
and crowned with the stars.

Thomas Traherne

The CPM.LIB library file provides access to the facilities of a computer running CP/M-80 or CP/M-Plus (also known as CP/M 3). It also contains some useful machine-independent functions for handling files and strings which are not found on UNIX systems. These supplement stdio.lib which contains machine-independent functions usually supplied with C compilers on UNIX systems.

6.1 int cpm_bdos(func, param).

```
int func, param;
```

func is the BDOS function number required; **param** is the parameter required by BDOS (or use 0 if none required); and the **return value** is that returned by BDOS (or garbage).

This function simply calls the CP/M BDOS via the jump at location 5. It is the recommended way of using CP/M functions, if it is possible to do what you need by using it. In particular, some of the other ways of doing things (eg directly reading and writing the iobyte or disc number) may not work on some CP/M look-alikes or on CP/M Plus; but then not all BDOS calls are implemented on some so-called CP/M look-alikes either!

As an added convenience many of the BDOS functions are also supplied as specific C functions. This also allows easy reworking of individual functions where necessary on "not-quite-CP/M" systems.

The BDOS function supplied makes some assumptions about your version of CP/M. The implications and consequences of these assumptions if you are not running CP/M 2.2 or CP/M-Plus are listed below. It is assumed that your CP/M:

1. Has a jump vector at location 5 (this had better be true!). If the jump isn't there, then where is your stack pointer being loaded from? You may need to do a lot of poking into the runtimes!

2. Takes the function number in the C register. This is almost always true, and just needs altering if it should be different.

3. Takes any argument in the DE register pair. This is almost always true, and just needs altering if it should be different.

4. Returns any result in the HL register pair. This is true for CP/M 2.2, CP/M-Plus, and MP/M; but not for certain other systems. The most common difference is that single-byte results might be returned just in the A register, and not in the HL registers as well. If this should be so uncomment the LD_A_into instruction and use the global variable reg_a in the specific calls that are affected.

5. Preserves SP and PC but may destroy any other register. The only register we care about is IX which is used as our stack frame pointer so we take care to preserve this.

6.2 int cpm22_bios(func, bc_param, de_param);

```
int func, bc_param, de_param;
```

This function provides access to the CP/M 2.2 BIOS. You are recommended not to use the BIOS entry points (or low-store addresses such as the IOBYTE supported (or not) by the BIOS). You should use the corresponding feature in the BDOS wherever possible. Note in particular that the BDOS and the BIOS may each hold information about the current disc position, and they may get confused and cease working if both BDOS and BIOS functions are used to access discs.

The cpm22_bios function takes a function number **func** (from 0 to 17), a main parameter **bc_param** which is loaded into the **bc** register, and an auxiliary parameter **de_param** which is loaded into **de** (used only for SECTRAN), it returns an int result, normally from the **a** register but from **hl** in the case of the SELDSK and SECTRAN functions. Note that the auxiliary (**de**) parameter MUST be present for all calls; use a value of zero for calls where it is not used.

A summary table of the BIOS functions and the registers used is shown below:

FUNC	NAME	INPUT	OUTPUT
00	BOOT	--	--
01	WBOOT	--	--
02	CONST	--	A
03	CONIN	--	A
04	CONOUT	C	--
05	LIST	C	--
06	PUNCH	C	--
08	READER	--	A
09	HOME	--	--
10	SELDSK	C	HL
11	SETTRK	BC	--
12	SETSEC	BC	--
13	SETDMA	BC	--
14	READ	--	A
15	WRITE	--	A
16	LISTST	--	A
17	SECTRAN	BC,DE	HL

6.3 int cpm3_bios(func, a_param, bc_param, de_param, hl_param).

```
int func, a_param, bc_param, de_param, hl_param;
```

This function provides access to the CP/M 3 BIOS. You are recommended not to use the BIOS entry points (or low-store addresses such as the IOBYTE supported (or not) by the BIOS). You should use the corresponding feature in the BDOS wherever possible. Note in particular that the BDOS and the BIOS may each hold information about the current disc position, and they may get confused and cease working if both BDOS and BIOS functions are used to access discs.

The cpm3_bios function takes a function number func and parameters for each of the registers A, BC, DE, and HL: a_param, bc_param, de_param, and hl_param. It builds the required BIOS-parameter-block and calls CP/M 3 BDOS function 50 to access the BIOS. It returns the result of this call.

6.4 Command Line Support and I/O Redirection for Compiled Programs.

Command line handling is provided by HiSoft C, but there are certain differences from UNIX command line handling as described in Kernighan & Ritchie. We have however tried to retain as much compatibility as possible.

The compiled program can access a series of arguments supplied on the command line; and the standard input (stdin) and standard output (stdout) are automatically redirected by command lines such as:

```
A>program arg1 <input arg2 arg3 >>append arg4
```

Full details of what is allowed on a command line are given below. The main difference from UNIX is that programs must call a library function cpm_cmd_line(). This is because the facility takes a significant amount of memory. Those programs that don't use command line handling need not call the function, and avoid wasting this space.

The second difference is that CP/M does not provide access to the command name (ie the name of the program which is running). This name is normally available as argv[0] and to retain UNIX compatibility we return the string "HiSoft" as the name - this can be altered in "cpm_cmd_line()".

The user program declares argc and argv in the usual way as parameters. It also declares an area of buffer space which is used to hold the argument vector and the text of the command line arguments. The constant MAXARGS is used to determine the maximum number of arguments which can be handled, and 132 bytes are allowed for the text (the maximum CP/M line length).

It calls cpm_cmd_line() **before** it uses argc or argv - usually first thing in the program. The call always has the same arguments.

We have provided an example of command line handling below. It is a modified version of the echo program given on page 111 of Kernighan & Ritchie. The modifications cause it to print out the first argument argv[0] to reinforce the point that under CP/M this **cannot** be the command name. Note that CP/M also forces the entire command line to upper case. [Thats why the compiler accepts directives in upper case as well!]

Arguments can be enclosed in double quotes if they need to contain spaces.

```
/* * * * * *      C O M M A N D   L I N E   E X A M P L E      * * * * * */
/*
    Example of Command Line Handling - a modified "echo" program which
    simply displays its command line arguments on the standard output.

        source file:    echo.c

  first comes the part common to all programs which use the command line
*/

#include stdio.h

main(argc,argv)
  int   argc;
  char *argv[];
{
  FAST char argv_buffer[MAXARGS * 2 + 132];

  cpm_cmd_line(&argc, &argv, argv_buffer);

  /* now for the example program */
  while (argc--)  printf(argc > 0  ?  "%s "  :  "%s\n", *argv++);
}

#include ?cpm.lib?
#include ?stdio.lib?

/* * * * * *      E N D   O F   E X A M P L E      * * * * * */
```

The compilation, running, and output from this program looks like this:

```
  A>hc echo.c
  HiSoft C Compiler V1.3. Copyright (C) 1985
  Line File

  MEMORY MAP     Start  End

  Runtimes       0100   11FF
  Code           1200   1635
  Initialisers   1636   1639
  Fixed Data     C247   C306

  Smallest #data 0x1702
  A>echo  hello,  world
  HiSoft HELLO WORLD

  A>echo >junk now      send     "   the output   " to a file
  A>echo and send some more >>junk
  A>type junk
  HiSoft NOW SEND    THE OUTPUT    TO A FILE
  HiSoft AND SEND SOME MORE
```

6.5 cpm_dir(drive, user, afn, sp, fp, width).

```
int drive, user, width;
char *afn, *sp;
FILE *fp;
```

cpm_dir produces a directory listing of a disc. It just produces a list of filenames so headings can be added as required by programs; this also makes it easier for application programs which intend to manipulate the directory in some way. For similar reasons, its arguments specify which files on which disk are to be listed. The application program should obtain this information via its user interface.

The directory is produced of files on **drive** (0 for drive A through to 15 for drive P) in user area **user** (0 to 15) which match the ambiguous file specification **afn** (normal CP/M style - such as "ed?tor*.com" or the usual "*.*"). If the file-pointer parameter **fp** is not NULL then the output is sent to the file. If the string pointer parameter **sp** is not NULL then output is sent to the string "sp". Other combinations result in either both types of output or none at all. Output is formatted across **width** columns.

You must reset the DMA address after calling this function if you intend doing any direct access to discs. This is because CP/M provides no way of discovering the current DMA address so this function cannot reset it. The runtime system resets the DMA address for subsequent normal file accesses.

6.6 int cpm_drive(new_drive).

```
int new_drive;
```

The CP/M operations of selecting a drive and interrogating the current drive number are combined in the function "cpm_drive". The current drive number is ascertained and returned after selecting the new drive passed as argument. Passing -1 as argument suppresses the selection and just returns the current drive number. The function can be used to select a drive temporarily and then restore the original drive (see "cpm_dir()").

6.7 cpm_pfcb(fcb, afn).

```
char fcb[36], *afn;
```

This function constructs a 36 byte CP/M file-control-block **fcb** from the ambiguous-file-name **afn.** **fcb** must be the address of a 36 byte area. The function takes a null-terminated C string which represents a CP/M filename in the usual way as a disk, filename, and extension. The wildcard characters '?' and '*' may be used. Example filenames are:

```
"A:FULLNAME.EXT"  "hc.*"  "a-file"  "P:ED*.COM"  "*.*"  "DATA??97.DAT"
```

6.8 int cpm_user(new_user).

```
int new_user
```

This function both sets a new_user number and returns the previous user number. This allows the old user number to be re-established later.

6.9 char *instr(main_string, sub_string)

```
char *main_string, *sub_string;
```

Find ocurrence of one string inside another. This function returns a pointer to the first occurrence of sub_string within main_string. It returns NULL if there is no such occurrence. This function must be declared before use as "extern char *instr();". Special purpose versions are easily written for greater speed.

6.10 itob(n, string, precision)

```
char *string;
```

Convert a number to a binary string. n is the number to be converted. **string** is the string to place the result in and must be a character array of length precision+1 or bigger. **precision** is the number of binary digits to place in the string.

eg itob(10, string, 7) gives string the value "0001010".

6.11 read_file(filename, address)

```
char    *filename, *address;
```

Read a named file into a block of store. Takes a string containing the filename to be used for the input file, the address of the start of a block of memory. It returns ERROR (-1) if the file could not be opened and TRUE (+1) if everything went OK.

6.12 char *strlower(string)

```
char *string;
```

Transform each upper case letter in a string to its lower case equivalent. Takes a string s and transforms it in place (the string is permanently changed - use strcpy first if necessary). Returns a pointer to the string. This function must be declared before use as: extern char *strlower();

6.13 char *strupper(string)

```
char *string;
```

Transform each lower case letter in a string to its upper case equivalent. Takes a string s and transforms it in place (the string is permanently changed - use strcpy first if necessary). Returns a pointer to the string. This function must be declared before use as: extern char *strupper();

6.14 write_file(filename, address, length)

```
char        *filename, *address;
unsigned    length;
```

Write a block of store to a named file. Takes a string containing the filename to be used for the output file, the address of the start of a block of memory, and the length (in bytes) of the block. It returns ERROR (-1) if the file could not be opened and TRUE (+1) if everything went OK.

HiSoft C

Fast Interactive K&R C Compiler

Chapter 6

The BASIC.LIB Library

HiSoft C

CPC SPECIFIC

6. THE BASIC.LIB FUNCTION LIBRARY FOR AMSTRAD COMPUTERS.

You never enjoy the world aright,
till the C itself floweth in your veins,
till you are clothed with the heavens,
and crowned with the stars.

Thomas Traherne

The BASIC.LIB library file provides access to the features of the Amstrad computers in a way which is intended to be familiar to the BASIC programmer. It supplements stdio.lib which contains machine-independent functions usually supplied with C compilers. The library is actually supplied as two files (BASIC1.LIB and BASIC2.LIB) so that each part can be loaded into the editor as required.

Another library called "firmware.lib" is also available which provides direct access to all the jumpblock entries. This can be bought direct from Hisoft.

Some of the BASIC commands provide facilities in a way which is completely different to C - for example string handling is normally done in a different style using pointers and the many library functions. This means that some of the examples given here are a little contrived, but they do at least give some idea of how to achieve a particular result using C.

This chapter of the manual is organised into two sections. The first lists all the Locmotive BASIC keywords in alphabetic order, and gives the name of the C function which corresponds to it, or shows how to achieve the same result if there is no directly corresponding function. The second section gives descriptions of all the functions actually contained in the "BASIC.LIB" files, so that they can be used properly.

6.1 BASIC Keywords And Their C Equivalents.

ABS is abs() in stdio.lib.

AFTER is after() in basic1.lib.

ASC is just *string_name.
eg
```
string = "Hello";
printf("%d", *string);
72
```

ATN is not available (floating point).

AUTO is the editor command 'i'.

BIN$ is itob() in basic1.lib.

BORDER is border() in basic1.lib.

CALL is just inline(0xCD, address);
eg
#define CALL 0xCD
inline(CALL, 0xBD19);

CAT is catalog() in basic1.lib or is done by getting a non-existent file using the editor.

CHAIN and CHAIN MERGE are not available.

CHR$ is not needed at all.
eg

PRINT CHR$(72);	becomes	putchar(72);
PRINT CHR$(13);	becomes	rawout(13);

putchar() is normally used to output a single character but rawout() is available for characters which are otherwise not directly output by the C input-output functions.

CINT is not needed (floating point).

CLEAR is automatic when you recompile.

CLG is G_clear_window() in basic2.lib (and firmware.lib).

CLOSEIN is fclose() built-in.

CLOSEOUT is fclose() built-in.

CLS is cls() in basic1.lib (which is rawout('\f');).

CONT is not needed (see STOP).

COS is not available (floating point).

CREAL is not needed (floating point).

DATA are initializers.

DEC$ is done by a call to sprintf.
eg
```
sprintf(string, "%d", 27);
puts(string);
```
27

DEF FN is just a function defintion.

DEFINT and DEFSTR and DEFREAL are not used in C; all variables are declared.

DEG is not available (floating point).

DELETE is the editor "d" command.

DI is event_disable() in basic1.lib. A more severe disabling of interrupts can be done by:

```
inline(0xF3);
```

but this will prevent keyboard scanning, stop timers, etc and must be reversed by a later:

```
inline(0xFB);
```

See also the discussion of events.

DIM is just a normal array declaration.
eg
```
char string_variable[3761];
```

DRAW is G_line_absolute() in basic2.lib (and firmware.lib). Colour is set separately with G_set_pen().

DRAWR is G_line_relative() in basic2.lib (and firmware.lib). Colour is set separately with G_set_pen().

EDIT is the editor "e" command.

EI is event_enable() in basic1.lib. See also DI.

END is just the end of the "main" function.

ENT is s_tone_env() in basic1.lib.

ENV is s_ampl_env() in basic1.lib.

EOF is done by testing each input character against constant EOF (-1).

```
eg
  /*  display each character in a cassette file  */

    while ((next_char = getc(cassette)) != EOF)
      putchar(next_char);
```

ERASE is done by allocating the array with "calloc()" and then reclaiming the memory with "free()".

ERR and ERL are not needed because C has no errors which cause program termination apart from "stack overflow" which is a disaster.

ERROR is not needed. See ERR.

EVERY is every() in basic1.lib.

EXP is not available (floating point).

FIX is not needed (floating point).

FOR is just a particular case of C's for statement.

```
for(simple_variable = start;
      simple_variable <= end;
      simple_variable += step_size)
    controlled_statement ;
```

FRE is not available, but see the editor's 'V' command.

GOSUB is just a function call
eg
```
qsort(list, length, size, func);
```

GOTO is just a goto statement.
eg
```
goto label_name;
```

HEX$ is done by a call to printf, or sprintf.
eg
```
printf("%x", 65534);
FFFE

sprintf(string, "%x %04x", 42, 42);
puts(string);
2A 002A
```

HIMEM the top of memory is 0xB0FF in a raw CPC464, and 0xABFC with DD1 discs or on a CPC664.

IF is just an if statement.
eg
```
if (expression_is_non_zero)
   then_execute_this_statement;
else
   execute_another_statement;
```

INK is ink() in basic1.lib.

INKEY is inkey() in basic1.lib.

INKEY$ is a combination of keyhit() and rawin(). Both are built in.
eg
```
rawout(12);
do {
   printf("Are you clever (y or n) ?");
   while ( ! keyhit() ) ; /* loop until a key press */
   c = toupper(rawin());
   if (c == 'N') printf("You must have been to buy me!");
   if (c == 'Y') printf("You're too modest!!!");
   }
while (c != 'Y' && c != 'N');
```

INP is inp() in stdio.lib.

INPUT is fscanf() which is built in.

INSTR is instr() in basic1.lib. Note that it returns a pointer to the substring (it corresponds more closely to a hypothetical INSTR$). An offset can be calculated if required.

```
eg:
   puts(instr("BANANAS"+2, "AN"));
   ANAS
   s = "There are many ways of writing special versions";
   printf("OFFSET IS %d", instr(s, "special") - s);
   OFFSET IS 31
```

INT is not needed (floating point).

JOY is joy() in basic1.lib.

KEY is key_function() in basic1.lib.

```
eg
   key_function(139,"c\n#include\n");
```

sets up the small [ENTER] key (without [CTRL]) to compile the C program held in RAM by the editor by just one keystroke.

KEY DEF is key_translation() in basic1.lib.

LEFT$ is strncpy() in stdio.lib.

```
eg:
   printf("%2s", strncpy(s,"HISOFT",2));
   HI
```

LEN is strlen() in stdio.lib.

LET is an assignment statement.

LINE INPUT is fgets() in stdio.lib.

LIST is the editor "l" command.

LOAD Source is read into the editor's text buffer with the "G" command. It can be compiled directly from disc or tape using the "#include filename" compiler directive. Your programs can read a complete file into memory using read_file() in basic1.lib.

LOCATE is just

```
rawout(31);
rawout(x);
rawout(y);
```

where x is the column number
and y is the row number
rawout() is built in

LOG is not available (floating point).

LOG10 is not available (floating point).

LOWER$ is strlower() in basic1.lib.

MAX is max() in stdio.h.

MEMORY is not needed.

MERGE is not available.

MID$ is done by strcpy() or strncpy() or by print format control as required.
eg

```
s = "High Quality Software";
printf("%.2s%.4s!", s, s+13);
```

HiSoft!

```
strncpy(t, s+5, 7);
t[7] = 0;
puts(t);
```

Quality

MIN is min() in stdio.h.

MODE is just

```
rawout(4);
rawout(new_mode_0_1_or_2);
```

MOVE is G_move_absolute() in basic2.lib (and firmware.lib).

MOVER is G_move_relative() in basic2.lib (and firmware.lib).

NEW is roughly the same as dl,32767 in the editor.

NEXT is not required (see FOR).

ON GOSUB and ON GOTO are just a switch statement.

eg

```
switch (day) {
   case 0: printf("Monday - back to the grindstone");
           break;
   case 4: printf("Friday - tomorrow's the weekend!");
           break;
   case 5:
   case 6: printf("weekend - don't wake me early");
           break;
   case 1:
   case 2:
   case 3: printf("boring");
           break;
  default: printf("%d is not a day of the week!!!", day);
 }
```

ON BREAK GOSUB and ON BREAK STOP are achieved by using K_arm_break() in basic1.lib (and firmware.lib).

ON ERROR GOTO has no equivalent (because there are no errors!).

ON SQ GOSUB is replaced by the sound handling facilities.

OPENIN is fopen(filename,"r"); which is built in.

OPENOUT is fopen(filename,"w"); which is built in.

ORIGIN is G_set_origin() in basic2.lib (and firmware.lib). Window size is set separately by G_win_width() and G_win_height().

OUT is out() in stdio.lib.

PAPER is

```
rawout(14);
rawout(ink);
```

PEEK is peek() in stdio.lib, see also dump() in the examples in the manual.

PEN is

```
rawout(15);
rawout(ink);
```

PI is not available (floating point).

PLOT is G_plot_absolute() in basic2.lib (and firmware.lib).

PLOTR is G_plot_relative() in basic2.lib (and firmware.lib).

POKE is poke() in stdio.lib but also often can just be done with pointers.

POS is T_get_cursor() in basic2.lib (and firmware.lib).

PRINT is normally printf() but fprintf(), sprintf(), puts(), putchar(), putc() rawout() etc are also available.

RAD is not available.

RANDOMIZE is srand() in stdio.lib.

READ is like an initializer.
eg

```
char *s[] =
      {"Adam", "Dave", "Mabel", "Mark", "Richard"};
```

RELEASE is S_release() in basic1.lib (and firmware.lib).

REM is /* a comment like this */.

REMAIN is not needed when using the event facilities of BASIC.LIB. The required time can be read directly from the tick block.

RENUM is the editor's N command.

RESTORE is not needed in C programs (because we have arrays of strings etc).
eg

```
int   times, word;
char *s[] =
      { "initialized", "data", "is", "accessible" };

for(times=0; times<3000; ++times)
   for(word=0; word<4; ++word) puts(s[word]);
```

RESUME is not needed (see ON ERROR GOTO).

RETURN is done automatically at the end of a function but the return statement can also return a value.

RIGHT$ is just an addition.
eg

```
puts("Hello world" + 6);
world
```

RND is rand() in stdio.lib.

ROUND is not needed (floating point).

RUN The BASIC RUN command is used to run a translated program. After a normal compilation press [CTRL]-Z and then Y to run a program.

SAVE Source is saved using the editor's P command. #translate automatically saves a binary program. You can save an area of store with write_file() in basic1.lib.
eg

```
write_file("SCREEN", 0xC000, 0x4000);
```

SGN is sign() in stdio.lib.

SIN is not available (floating point).

SOUND is part of the sound facilities.

SPACE$ is done with blt() which is built in.
eg
```
#define LENGTH 190
char who_uses_this[LENGTH + 1];

*who_uses_this = ' ';
blt(who_uses_this + 1, who_uses_this, LENGTH - 1);
who_uses_this[LENGTH] = 0;
```

SPEED INK is flash_speed() in basic1.lib.

SPEED KEY is key_speed() in basic1.lib.

SPEED WRITE is cass_speed() in basic1.lib.

SQ is sound_check() in basic1.lib.

SQR is not available (floating point).

STOP is done by:
```
printf("At debugging point 3 variable is %d", variable);
rawin();  /* wait for a key press */
```

STR$ is done with sprintf() which is built-in.
eg
```
char s[20];
sprintf(s, "%d", 0x766);
puts(s);
1894
```

STRING$ is done with blt() exactly like SPACE$.

SYMBOL is symbol() in basic1.lib. Note that C does not set any default user symbol area, so you must always use symbol_after before using symbol.
eg

```
#define ARROW 94

char circumflex[] =
   {0, 24, 60, 102, 102, 0, 0, 0};
char bytes[(256 - ARROW) * 8];

#direct+

symbol_after(ARROW, bytes);
symbol(ARROW, circumflex);
```

Redefines the up arrow key on the keyboard to display as a circumflex.

SYMBOL AFTER is symbol_after() in basic1.lib.

TAG is T_set_graphic() in basic2.lib (and firmware.lib).

TAGOFF is T_set_graphic() in basic2.lib (and firmware.lib).

TAN is not available (floating point).

TEST is G_test_absolute() in basic2.lib (and firmware.lib).

TESTR is G_test_relative() in basic2.lib (and firmware.lib).

TIME is time() in basic1.lib.

TRON and TROFF are normally achieved by using putchar() to print a character at significant places in the program, or printf to print a message and perhaps the value of some variables. See STOP.

UNT is not needed. Its all done automatically by C.

UPPER$ is strupper() in basic1.lib.

VAL is done with sscanf() which is built-in. See the entry for scanf in the Library chapter of the Manual.

VPOS is T_get_cursor() in basic2.lib (and firmware.lib).

WAIT is just:

```
do ; while ((inp(port_number) ^ inversion) & mask) ;
```

You can escape from the loop just by pressing [ESC] twice.

WEND is not needed (see WHILE).

WHILE is a while or do loop.

eg

```
square_root = 1;
while (square_root * square_root  < square)
  ++square_root;
```

or

```
do c = getchar(); while (c == ' '  ||  c == '\n');
```

WIDTH is not available. Control of printing format is very flexible using fprintf() and you can normally rely on your printer to wrap round to the next line automatically if you don't send it the right formatting information.

WINDOW is T_win_enable() in basic2.lib (and firmware.lib).

WINDOW SWAP is T_swap_streams() in basic2.lib (and firmware.lib).

WRITE is done by using printf(), but its normally better in C to write a list of values without commas because they are easier to read back in again.

XPOS is G_ask_cursor() in basic2.lib (and firmware.lib).

YPOS is G_ask_cursor() in basic2.lib (and firmware.lib).

ZONE is not needed.

6.2 Events and C.

The firmware operating system in ROM on the Amstrad computers provides a wide variety of event processing facilities which let the user perform many tasks at once. The most general access to these features is provided by means of direct calls to the ROM software using the functions in firmware.lib. These calls are not really suitable for the normal requirements because they involve much work in setting up parameters and can have disastrous consequences if incorrectly set up. So several functions are provided in basic1.lib which allow commonly used patterns of event handling to be set up easily.

The functions provided are similar to those found in Locomotive BASIC but they do differ in some areas to fit in more easily with the C style. In particular, for those who are interested, asynchronous events are used rather than synchronous ones; and extra data are placed on the end of all event blocks.

The most important thing to remember when using these events is that either all display printing should be done by the event functions, or it should be done by the normal program, but not by a mixture. The reason for this is because event functions can interrupt the firmware whilst it is running and the firmware is not re-entrant. So take the decision that one or the other set of functions (event functions or normal program functions) will do all the output to avoid accidentally re-entering the firmware. Event functions do not interrupt each other, so it doesn't matter how many of them use the firmware.

The event blocks used by C are larger than those used by the firmware in order that arrays of events can be constructed. The layout is as follows:

byte	contents
0,1	event chain
2	event count
3	event class - asynchronous (0x80)
4,5	pointer to byte 7
6	ROM select - all RAM (0xFF)
7	PUSH_HL (0xE5)
8	CALL (0xCD)
9,10	event function address
11	RET (0xC9)
12+	onwards available for user fields

Thus each event block is 12 bytes long, of which the first seven are a normal firmware event block. The event routine address is always set to point at byte 7 of the block; which contains a short

machine code program. This program pushes HL onto the stack, calls the real event function whose address is contained in bytes 9 and 10, and returns afterwards. HL has been loaded by the firmware with a pointer to the event block, and it is placed on the stack so that the event function can access it as a parameter. This enables the event function to access the block, in case it is necessary to alter any of the fields, or to use data contained in the optional user fields.

The sound functions in the library illustrates the techniques. Note that HL points to byte 4 of the event block; NOT as variously described in the firmware specification either byte 5 or byte 7 ("the address of the user fields"). Note also that these asynchronous event functions are at "far" addresses, and NOT at "near" addresses as suggested in the firmware manual. This is because the function could be in the lowest 16K of RAM; and the lower ROM is always switched in for a near address.

The functions used as event functions are normal C functions but they are subject to the restriction that they should have exactly one parameter which is the address of byte 4 of the event block as discussed above. The function can ignore this parameter if it has no use for it, but it must still be declared.

6.3 Sounds of the C.

BASIC.LIB contains a group of functions providing a simple but very flexible interface to the extensive sound facilities of the AMSTRAD. A typical use of the routines is shown below.

```
#include stdio.h

main()      /* try this out */
{
  char *string;

  string = "T\170 O\3 F. G. A'. Bb'. C'\8.";
  setup_sound();
  play(string, 1);
  play("T\360 O\4 C'\2.W\2.G\2.W\1.G\1.A'.G\2.W\6.B'\2.W\2.C'\8.", 1);
}

#include ?basic1.lib?
```

The function setup_sound() must be called before using the main function play(). play() has two parameters. The first is of type pointer-to-char and is the string to be played and the second is a char and indicates (among other things) which channel the string is to be played on. It is identical in all respects with the Channel Status parameter to the BASIC command SOUND.

The string used in the play statements consists of a sequence of notes and sound primitives that are obeyed in turn. There are seven controlling primitives each of which should be followed by a parameter byte (using the backslash character to put bytes into strings as in standard C. Remember that the byte is written as OCTAL NOT DECIMAL).

The meaning of the parameters to follow mirror exactly those for the BASIC command SOUND.

S\x The parameter (\x) sets the status byte. This change lasts for one note only after which the top five bits will be reset to 0 for subsequent notes on the same channel. Any attempt to change the channel number (ie the bottom 3 bits) will be ignored. The channel that the string is played on is decided at the start by the status parameter given when play() is called and lasts for the whole string. (The action of the status parameter is to insert an S command at the beginning of the string).

The next 6 commands all apply on the specified channel until subsequent commands of the same type are issued on that channel.

V\x Set the volume. The range is 0 to 15 with default 12.

N\x Set the amount of noise with each note. The range is 0 to 31 with default 0.

Y\x Specify which volume envelope to use. The range is 0 to 15 with default 0. Volume envelope 0 leaves the volume constant for 2 seconds.

Z\x Specify which tone envelope to use. The range is 0 to 15 with default 0. Tone envelope 0 does not modulate the tone.

O\x Set the base octave. The range is 0 to 8 with default 3 (which is the octave containing middle C on the piano).

T\x Set the tempo. The range is 0 to 255 with default 60. This parameter is used exactly as in a music score. For example a tempo of 60 means one beat per second or a tempo of 120 means two beats per second.

Any character other than those above is taken to be the start of a note specifier as detailed below.

A note consists of a certain pitch and a certain duration (both of which are relative to the O and T commands above). The pitch of a note is specified simply by its name - a letter between A and G (capitals only). Sharp signs (#) or flat signs (b) following the note will raise or lower the pitch of the note by a semitone. Apostrophes (') preceding the note will each lower the note by one octave and apostrophes following will each raise the note by one octave. Following apostrophes should be placed after and '#' or 'b' signs. Apart from the letters A to G, a W (standing for Wait) may be used to produce silence. As in the example above it may be followed by a duration byte (see below) and must be played like any other note (by use of a full-stop as below).

The duration of the note may be set by placing a byte of value less than \40 (32 decimal) after all of the pitch specifiers. If this is omitted a value of \4 is used (which in combination with the default tempo of 60 gives the note a duration of one second). Remember that bytes after the backslash are OCTAL in C (eg \16 = 14 decimal).

Finally, a full-stop (.) will play the last note to have been set up (including a W for silence). If this is omitted, the note will be ignored.

If the note is preceded by any character other than those detailed above (eg spaces or underlines) then these are taken to be separators and are ignored.

If you are in doubt about the meaning or use of any of the parameters, consult the BASIC command SOUND (or the firmware routine SOUND_QUEUE) as SOUND.LIB is designed to emulate the effect of these.

NOTE TO CPC464 USERS as opposed to CPC664 users.

Owing to a bug in the firmware of the 464 (corrected in the 664) concerning asynchronous sound events, the program as supplied will not work correctly. A very small change is required in the routine init_event() as follows.

```
reg_bc = 0x80FF;        should become
reg_bc = 0x02FF;
```

For those "in the know" about events, this makes the routine _sound_event() into a synchronous event (rather than asynchronous). This is then kicked by a routine in the compiler which itself is an asynchronous event called 10 times per second. Thus a translated program using these routines will not work at all on the 464 without other code, which is also supplied as "fix_464_sound_interrupts()" in BASIC2.LIB. The change to init_event() applies only for use with play(). In other circumstances, the bug in the firmware may well not be visible and the change need not be made.

6.4 Graphics and C.

Various functions are provided to access the graphics on the computers. In general these mirror the way that BASIC uses the firmware, but in addition this library provides a function "draw" which displays complex line graphics. The drawing is specified by a control string (like printf) which is similar to the Unix GPS or the MSX GML.

A string to be drawn consists of a sequence of graphical primitives which each consist of a single character, possibly followed by parameters. The string is more properly thought of as a character array, since it is likely to contain '\0' characters in co-ordinate values etc. It is terminated by a '\0' which is not part of a parameter, since "draw" can recognise this, but watch out when trying to use normal string manipulation functions.

The most common sort of parameter is a four byte sequence containing a binary x-y relative co-ordinate pair (dx, dy) in the form:

```
low byte  of x-increase
high byte of x-increase
low byte  of y-increase
high byte of y-increase
```

The graphical primitives are all relative to the current graphics cursor position (except 0).

char	name	bytes	description.
0	zero	0	move graphics cursor to absolute position (0,0) (bottom left).
m	move	4	move graphics cursor from current position by (dx, dy).
p	plot	4	as move then plot at point in current pen ink.
l	line	4	draw line from current position (x,y) to (x+dx, y+dy).

t	text	n	print text from string starting at current position. Text is terminated by a newline character '\n'.
c	colour	2	if next character is 'f' then change pen ink to next byte else [next byte is 'b'] change paper ink to next byte.
s	scale	4	change horizontal and vertical scaling to dx and dy respectively. The scales are shifted by DRAW_SCALE (2) so a unity scale is represented by 4.
r	rotate	0	rotate axes 90 degrees clockwise.

FUNCTIONS IN BASIC1.LIB (GENERAL AND SOUND).

6.5 after(delay_in_ticks, control_block, function_name)

```
int  delay_in_ticks;
char *control_block[18];
int  (*function_name)();
```

Invoke a function after a given time period has elapsed. See the section on "Events and C" above.

delay_in_ticks indicates the time in units of 0.02 sec (50 make one second).
control_block is the address of an 18 byte area for system use.
function_name is the name of the function to call.

6.6 every(period_in_ticks, control_block, function_name)

```
int  period_in_ticks;
char *control_block[18];
int  (*function_name)();
```

Invoke a function at regular intervals. See the section on "Events and C" above.

period_in_ticks indicates the repetition interval in units of 0.02 sec.
control_block is the address of an 18 byte area for system use.
function_name is the name of the function to call.

6.7 add_ticker(ctrl_block, initial_time_delay, recharge_delay, function_name).

```
char *ctrl_block[18];
int  initial_time_delay, recharge_delay;
int  (*function_name)();
```

Add a new block to the 1/50th second ticker queue. This function is provided for use by after() and every() but can also be called by the confident user. See the section on "Events and C" above.

ctrl_block is the address of an 18 byte area for system use.

initial_time_delay indicates the time to first run in units of 0.02 sec.
recharge_delay indicates the repetition interval in units of 0.02 sec.
function_name is the name of the function to call.

6.8 init_event(event_block, function_name)

```
char *event_block[12];
int  (*function_name)();
```

Initialise an event block. This function initialises a particular kind of event block, specialised for use in C programs. See the section on "Events and C" above.

event_block is the address of a 12 byte area for system use.
function_name is the name of the event routine.

6.9 cass_speed(speed)

Set the speed at which data is written to the cassette. Use 0 for 1000 baud (approx) or 1 for 2000 baud (approx) like BASIC.

6.10 border(colour1, colour2)

Change the colour of the border on the screen. Two colours must always be supplied, but they may be the same to avoid flashing.

6.11 catalog()

Produce a catalogue of a cassette tape.

6.12 cls()

Clear the Current Text Window.

6.13 event_disable()

Disable Synchronous Events ("interrupts"). This does not find as much use in C as it does in BASIC, since asynchronous events are used rather than synchronous ones.

6.14 event_enable()

Enable Synchronous Events ("interrupts") This does not find as much use in C as it does in BASIC, since asynchronous events are used rather than synchronous ones.

6.15 flash_speed(time1, time2)

Set up the rate at which flashing inks flash. Takes two times, each measured in frame flyback periods. (ie a time of 50 is one second in the UK and other 50Hz areas, and a time of 60 is one second in the USA and other 60Hz areas). The inks are displayed in their first colour for time1 and in their second colour for time2.

6.16 ink(ink_to_setup, colour1, colour2)

Associate colours with an ink. There must always be two colours, but they may be the same to avoid flashing.

6.17 int inkey(key_number)

See whether a particular key is pressed, and maybe [SHIFT] or [CTRL]. The return value is just the same as BASIC:

KEY	[SHIFT]	[CTRL]
-1	UP	?	?
0	DOWN	UP	UP
32	DOWN	DOWN	UP
128	DOWN	UP	DOWN
160	DOWN	DOWN	DOWN

6.18 char *instr(main_string, sub_string)

```
char *main_string, *sub_string;
```

Find ocurrence of one string inside another. This function returns a pointer to the first occurrence of sub_string within main_string. It returns NULL if there is no such occurrence. This function must be declared before use as "extern char *instr();". Special purpose versions are easily written for greater speed.

6.19 itob(n, string, precision)

```
char *string;
```

Convert a number to a binary string.

n	is the number to be converted.
string	is the string to place the result in.
precision	is the number of binary digits to place in the string.

eg itob(10, string, 7) gives string the value "0001010". "string" must be a character array of length precision+1 or bigger.

6.20 joy(joystick_number)

Fetch the current state of the joysticks. Returns the state of joystick 0 or 1, depending on the value of joystick_number. The return value is the sum of individual bits just as in BASIC.

bit	meaning	decimal
0	up	1
1	down	2
2	left	4
3	right	8
4	fire 2	16
5	fire 1	32

6.21 int key_function(translated_key_number, expansion_string)

```
char * expansion_string;
```

Set up a new function key definition. Accepts a translated_key_number (an expansion token) and an expansion_string. Returns TRUE (1) if the expansion was set up OK or FALSE (0) if there was a problem.

6.22 key_speed(start_up_delay, time_between_repeats)

Set the speed at which keys repeat. Set the start_up_delay which is the time a key must be held down before it starts to repeat, and the time_between_repeats. Both times are measured in keyboard scans. The keyboard is normally scanned every 1/50th of a second.

6.23 key_translation(key_number, translated_key_number)

Set a new translated value for a given key number.

6.24 play(string, channel)

```
char *string;
int channel;
```

This sound function plays a series of notes described in "string" on the channel specified. The channel number is bit significant, so should be 1, 2, or 4. See the "Sounds of the C" section above.

6.25 read_file(filename, address)

```
char    *filename, *address;
```

Read a named file into a block of store. Takes a string containing the filename to be used for the input file, the address of the start of a block of memory. It returns ERROR (-1) if the file could

not be opened and TRUE (+1) if everything went OK.

6.26 S_release(channel_bits)

Release all sounds that have been held up by S_hold, and those held by hold status bits on the channels specified.

6.27 S_ampi_envelope(number, envelope)

Set up an amplitude envelope. The envelope parameter will normally be the address of (or a pointer to) a structure like that below.

```
typedef union {
  struct {
      char count, size, pause_time;
    } software_section;
  struct {
      char      shape;
      unsigned period;
    } hardware_section;
  } section;

typedef struct {
  char    count_of_sections;
  section the_sections[5];
  } sound_envelope;
```

6.28 S_tone_envelope(number, envelope)

Set up a tone envelope. The envelope parameter will normally be the address of a structure as above.

6.29 S_hold()

Suspend all sounds immediately.

6.30 S_continue()

Resume sounds that have been suspended by S_hold.

6.31 setup_sound()

Initialises the C sound mechanism. Must be called before play() is used. See "Sounds of the C" above.

6.32 sound_check(channel)

Check the state of a sound queue. The result is bit significant just like SQ in BASIC.

6.33 char *strlower(string)

```
char *string;
```

Transform each upper case letter in a string to its lower case equivalent. Takes a string s and transforms it in place (the string is permanently changed - use strcpy first if necessary). Returns a pointer to the string. This function must be declared before use as: extern char *strlower();

6.34 char *strupper(string)

```
char *string;
```

Transform each lower case letter in a string to its upper case equivalent. Takes a string s and transforms it in place (the string is permanently changed - use strcpy first if necessary). Returns a pointer to the string. This function must be declared before use as: extern char *strupper();

6.35 symbol(character_number, matrix)

```
char matrix[8];
```

Set up a user-defined symbol. Takes the number of the symbol to be redefined (call symbol_after first), and an eight-character array containing the new matrix.

6.36 symbol_after(number, table_memory)

```
int  number;
char table_memory[];
```

Allocate an area for user defined symbols. Takes a number of permissible user symbols, together with an area of memory to hold the table of their matrices. The area of memory must be at least (256 - number) * 8 bytes long. In C you are given a choice of where to put this area by appropriate declaration of table_memory, or by a call to calloc().

6.37 time(array)

```
int array[2];
```

Report the elapsed time. The elapsed time is reset at switch-on and by KL_CHOKE_OFF which is called when the [ESC] key is used to abort a listing, compilation, or program. The time is returned in a four byte array which is compatible with those used by long_add() etc in the standard library. The array would normally be declared as "char time_array[4]", then the most significant byte is in time_array[3] and the least significant byte in time_array[0].

6.38 write_file(filename, address, length)

```
char    *filename, *address;
unsigned length;
```

Write a block of store to a named file. Takes a string containing the filename to be used for the output file, the address of the start of a block of memory, and the length (in bytes) of the block. It returns ERROR (-1) if the file could not be opened and TRUE (+1) if everything went OK.

FUNCTIONS IN BASIC2.LIB (GRAPHICS).

6.39 draw(control_string)

```
char *control_string;
```

See the section "Graphics and C" above.

6.40 T_set_graphic(on)

Turn on or off the graphics VDU write character option. The parameter should be TRUE(1) to turn the option on or FALSE(0) to turn it off.

6.41 T_win_enable(x1,x2,y1,y2)

Set the size of the current text window.

6.42 T_swap_streams(stream_number, another_stream_number)

Swap the states of two streams.

6.43 T_get_cursor(px_column, py_row, p_roll_count)

```
int *px_column, *py_row, *p_roll_count;
```

Get current cursor position.

6.44 G_ask_cursor(pdx, pdy)

```
int *pdx, *pdy;
```

Get the current graphics position.

6.45 G_set_origin(x,y)

Set the origin of the user co-ordinates.

6.46 G_win_width(x1,x2)

Set the left and right edges of the graphics window.

6.47 G_win_height(y1,y2)

Set the top and bottom edges of the graphics window.

6.48 G_clear_window()

Clear the graphics window.

6.49 G_set_pen(ink)

Set the graphics plotting ink.

6.50 G_set_paper(ink)

Set the graphics background ink.

6.51 G_wr_char(c)

Put a character on the screen at the current graphics position.

6.52 G_move_absolute(x,y)

Move to an absolute graphics position.

6.53 G_move_relative(dx, dy)

Move relative to current graphics position.

6.54 G_plot_absolute(x, y)

Plot a point at an absolute position.

6.55 G_plot_relative(dx, dy)

Plot a point relative to the current position.

6.56 int G_test_absolute(x, y)

Test which ink is at an absolute position.

6.57 int G_test_relative(dx, dy)

Test which ink is at a position relative to the current one.

6.58 G_line_absolute(x, y)

Draw a line to an absolute position.

6.59 G_line_relative(dx, dy)

Draw a line relative to the current position.

EXTERNAL COMMANDS FROM C PROGRAMS.

The file EXTCMD.H appears after stdio.lib on cassette and contains two functions which let you use external commands from inside your C programs; giving you similar facilities to the vertical stroke "|" command in the editor and BASIC. As external commands have varying numbers of parameters the main function is variadic and so must be defined before it is used. This file also needs some of the #define statements in stdio.h so you should use #include extcmd.h after #include stdio.h .

6.60 extcmd(string, args ....) auto

```
char *string;
```

This takes a variable number of parameters. The first is a string (the name of the external command) and any more are arguments to be passed to the external commands. Integer arguments are passed directly, but string arguments should be passed using makestr (see below). eg.

```
char d[3], e[3];
extcmd("dir");
extcmd("user", 1);
extcmd("era", makestr("*.bak", d));
extcmd("ren", makestr("newfile", d), makestr("oldfile", e));
```

6.61 int makestr(string, descriptor)

```
char *string, descriptor[3];
```

makestr is needed because the firmware requires the string parameters to external commands to be passed as a three byte string discriptor. The first parameter of makestr is an ordinary string and the second is the name of a three character array for the descriptor. The value returned is the address of the descriptor so that makestr can be used as an argument to extcmd as in the examples above.

HiSoft C

Fast Interactive K&R C Compiler

Chapter 7

Errors

HiSoft

High Quality Software

7. ERRORS.

O hear us when we cry to Thee
For those in peril on the C.

William Whiting

Errors are an important and all but inevitable subject when writing programs. In this chapter of the manual we discuss errors in general, then provide a detailed list of the error messages that can be produced by the compiler and what they mean. Finally we discuss some common errors in C programs, and their consequences.

7.1 Introduction.

When you ask the compiler to compile your program, it will scan through your program recognizing the various constructs that you have used and generating appropriate code for them. Whilst doing this it may find an error in your program. If the compiler does find an error it will display an error message like this:

```
ERROR nn
error message text
```

There are three parts to the error message. First the word ERROR just lets you know that this is an error message. Secondly, the error number "nn" tells what the compiler thinks is wrong and lets you find more information about the error in the next section of this manual. Thirdly is the error message text, which is a brief indication of the cause of the error. In many cases it will be all that you need to work out what is wrong and how to correct it.

After displaying the error message, the compiler will wait for you to press any key and will then return to its sign-on message, unless you choose to do an automatic edit of the line (see the Editor Chapter).

The error message texts take up a significant portion of the computer memory (approx 2 kilobytes) and there is a facility to reclaim this. The control line:

```
#error
```

will cause the compiler to discard the error messages and put the memory to general use so that you can write larger programs. After the compiler has done this an error message looks like this:

```
ERROR nn
```

The only way to get the error messages back is to reload the compiler. You can always look up the error numbers in this chapter, of course. Some error messages result from a mistake in the sequence of characters that you have typed (eg a missing semi-colon). These are often called SYNTAX errors. Another kind of error results from an inappropriate use of the language and these are called SEMANTIC errors (eg trying to assign a value to an array identifier). The compiler makes no distinction between

these two kinds of errors.

A third kind of error message occurs when your program is too large for the compiler to handle. These are compiler LIMITS and are identified by the word LIMIT at the start of the error message text. They have been chosen so that the great majority of programs will compile without reaching the limit. If you do encounter a limit then try to break up the part of the program concerned. More detailed suggestions are given for each individual limit.

A fourth kind of error message occurs if you write a program which includes a legal C construct that is not implemented by this compiler. This is called a RESTRICTION and is indicated by the word RESTRICTION at the start of the error message text. Details of restrictions are given in the Expert's Guide. Some restrictions may be lifted in future versions of the compiler.

A fifth kind of error is a COMPILER ERROR. We hope that these will be rare but include advice here in case you find one. It may be indicated by an error message starting with the word COMPILER or with a number not in the list below. No details of these messages are given since they may vary from time to time. They result from internal checks in the compiler. Of course a compiler error may not be so well behaved as to produce an error message but may cause the compiler to crash or loop etc. If you think that this has happened, first try again after reloading the compiler from tape in case it has been corrupted. If the problem persists then please contact HISOFT - we will be pleased to help you. It will be easier to help you if you have clear details.

There is also a sixth kind of error - a RUNTIME error. This is an error which occurs whilst your program is actually running. The overall philosophy of C is not to produce these errors, and so avoid the need for code to test for them! Thus, for example, arithmetic overflow is not an error, and neither is going beyond the bounds of an array. There is just one error condition which is actually checked for at runtime in Hisoft-C, and that is a stack overflow (see below).

The List of Error Messages

7.2 stack overflow (runtime error).

This is the only error message which can occur at runtime. It means that the stack used by the program has grown until it has filled all the available space. This may simply be because the program and its data are too large, but it might be because of an infinite recursion or some such.

7.3 ERROR - 0 - missing 'x'.

This is a slightly special message, because the 'x' character can vary. The message says that the compiler believes that the 'x' character should come next in the program, and it doesn't. Examples are missing ';' at the end of a statement or a missing ')' in a function call. Some care is needed to understand the message properly. The compiler is reading the program from the beginning and cannot see the rest of the program beyond the place that it has reached, so it has to decide what error message to produce on the basis of what it has seen so far. This limitation, together with our desire to keep the compiler as small as possible, means that to people looking at the whole program the message may seem strange.

For example, if you write f(a b); then the compiler will say "missing ')'" after the "a". The explanation is lengthy but straightforward: the compiler knows it is compiling a function argument list, and starts compiling the first argument expression "a". It then reads the next token "b" and decides that the first argument has now finished because two variables cannot be adjacent in an expression (they must have a '+' or some other operator between them). Now if there are more arguments in the list then there ought to be a comma next and clearly there isn't so the compiler decides that it must have reached the end of the argument list. At the end of the argument list there must be a ')' - but instead there is a "b". The compiler generates the "missing ')'" message.

7.4 ERROR - 1 - RESTRICTION : floats not implemented.

Sorry!

7.5 ERROR - 2 - bad character constant.

The compiler didn't find the closing '. Check the syntax in the "Language Summary" chapter.

7.6 ERROR - 3 - not a preprocessor command.

This line looks like a preprocessor command (it starts with a #) but it isn't one that the compiler recognises. Check in the "Language Reference" chapter.

7.7 ERROR - 4 - LIMIT: macro buffer full.

The intention is that this limit won't be reached in normal use. The compiler has a buffer where it remembers the definitions from #define lines which are currently being expanded. This buffer is now full. Try to simplify the macro definitions. Note that a circular macro definition can cause this error.

7.8 ERROR - 5 - can only define identifiers as macros.

Check what is allowed on a #define line in Kernighan & Ritchie.

7.9 ERROR - 6 - RESTRICTION: macros may not have parameters.

The compiler can only handle macros which don't have parameters (ie those which are a straight token replacement). Alternatively you may be trying to use bracketted replacement text and have left out the necessary space:

```
#define FIFTH_ELEMENT(array+4)  /* is not C */
#define SIXTH_ELEMENT (array+5) /* is C - the parenthesis can be useful */
```

7.10 ERROR - 7 - cannot open file.

The compiler cannot find an include file.

7.11 ERROR - 8 - RESTRICTION: cannot nest includes.

The compiler permits one level of #include for the main program and a further level for header files and functions and libraries. These files cannot have a further level included in them.

7.12 ERROR - 9 - missing 'while'.

At the end of a "do" statement there must be the word "while" :

```
do statement while (expression) ;
```

The compiler is looking for that "while" and hasn't found it.

7.13 ERROR - 10 - not in loop or switch.

A "break" statement is used to exit from a "switch" statement or from a loop such as a "do" or a "while" or a "for". This "break" isn't inside one.

7.14 ERROR - 11 - not in loop.

A "continue" statement is used to go back to the top of a loop such as a "do" or a "while" or a "for". This "continue" isn't inside one.

7.15 ERROR - 12 - not in switch.

"case" and "default" introduce the action for particular values in a "switch" statement. They can't be used outside a "switch".

7.16 ERROR - 13 - LIMIT : too many case statements.

There is a limit of 50 "case" statements active at once. A "case" is active from when it occurs until the end of the "switch" enclosing it. Several switch statements in succession provide a way round this limit. Eg: switch(c) { case 'a':...; case 'b':...;} switch(c) { case 'd':...; }

7.17 ERROR - 14 - multiple default statements.

Each "switch" statement can only have one "default" statement inside it.

7.18 ERROR - 15 - goto needs a label.

Every "goto" statement must have a corresponding labelled statement somewhere in the same function body.

7.19 ERROR - 16 - multiple use of identifier.

An identifier that is used to label a statement, or as the label in a "goto" statement, cannot be used as a variable name (either global or local to the same function as the label) as well. The converse is also true.

7.20 ERROR - 17 - direct execution not possible when translating.

When you are translating a program for stand-alone execution it is not possible to use #direct.

7.21 ERROR - 18 - LIMIT : name table full.

The compiler has a table which holds the name of every active variable, macro, and label in the program. This table is now full. Global variables and macros (ie #define names) take space from their declaration to the end of the program but local names only take space whilst their function is being compiled. So reducing the number of global names will save space and so will breaking up large functions into two or more smaller ones which don't have as many variables each.

7.22 ERROR - 19 - LIMIT : too many types.

The compiler has a table of all the types used in the program, and this table is now full. These are all types (eg int ****ptr) and not just named types. Reducing the number of types is the only way to save space, perhaps by using the type-cast operator to break up more complex pointer chains and so on.

7.23 ERROR - 20 - duplicate declaration - type mismatch.

This usually means that the name has been declared twice. Choose another name for one of the variables. It may be an intentional double declaration such as using a function before it is defined; in which case the type rules have been broken. For example a function used before it is defined is implicitly declared to return an int, and the subsequent definition must confirm this.

7.24 ERROR - 21 - duplicate declaration - storage class mismatch.

This usually just means that the variable name has been declared twice. Choose another name for one of the variables. Remember that in C a name can be declared twice sometimes (eg when a function is used before it is defined, or when the same name is used for members of two different structures) and in that case the second part of the message is important. It means that in this case the conditions required (eg that the members have the same offset in the structures) have not been met.

7.25 ERROR - 22 - LIMIT : global symbol table full.

The compiler has a table which holds details of each global variable in the program. The only way to save space in this table is to reduce the number of global variables.

7.26 ERROR - 23 - LIMIT : too much global data.

There is not enough space in memory for everything that has to fit, and more space is needed for global variables. Reducing the size of the global variables, or the size of the generated code (by not compiling as much) or the size of the in-memory source program (by #include) will all help to give more room. In addition, using the #error control line to sacrifice error messages will give more room.

7.27 ERROR - 24 - duplicate declaration.

This name has been declared twice in this function as a local variable or a parameter. Choose another name for one of the variables.

7.28 ERROR - 25 - LIMIT : local symbol table full.

The compiler has a table which holds details of each local variable and label in a function. This table is full. Space can be saved by reducing the number of local variables in the function which caused the problem, perhaps by splitting the function into parts which use fewer variables, or by using fewer "goto" labels (none!!) by rewriting using loops and conditionals.

7.29 ERROR - 26 - this variable was not in parameter list.

Only those variables which were in the parameter list between the ( ) of a function definition can appear again in the declarations before the start of the function body.

7.30 ERROR - 27 - undefined variable(s).

At the end of compiling a program, the compiler checks that all the variables used in the program have been properly defined and it lists any which have not. These are usually functions and if they are all library functions they can now be conditionally included by using a library search. This error does NOT restart the compiler. You can carry on and type more input to the compiler afterwards (such as #include ?stdio.lib?).

7.31 ERROR - 28 - bad function return type.

There are some restrictions in C on the types which a function may return as its value. The details are given in section 8.4 ("The Meaning of Declarators") of "the C Reference Manual" in Kernighan & Ritchie.

7.32 ERROR - 29 - no arrays of functions.

But it is possible to have arrays of pointers to functions. Perhaps that is what is needed.

7.33 ERROR - 30 - LIMIT : expression too complicated - too many arguments.

The intention is that this limit won't be reached in normal use. If it is break up the expression by an intermediate assignment or rearrange it with fewer parentheses.

7.34 ERROR - 31 - LIMIT : expression too complicated - too many operators.

The intention is that this limit won't be reached in normal use. If it is break up the expression by an intermediate assignment or rearrange it with fewer parentheses.

7.35 ERROR - 32 - bad type combination.

There are rules for each operator in an expression about what types of operands it can take and somewhere in this expression those rules have been broken. The compiler displays the error message as soon as it can but it has to evaluate the operands before it knows their types and can check them. Details of the combinations which are allowed are in section 7 ("Expressions") in the C Reference Manual. This message refers to a binary or ternary operator (eg + or ?:).

7.36 ERROR - 33 - bad operand type.

Similar to the previous error, but this refers to the operand of a unary operator such as * & ! etc.

7.37 ERROR - 34 - need an lvalue.

Some operators can only take arguments which are lvalues (eg = ++ -- &). The rules about when an operator needs an lvalue and what an lvalue is are given in the C Reference Manual, but roughly speaking an lvalue is a memory address which can be stored into. Remember in particular that an array name and a function name are NOT lvalues.

7.38 ERROR - 35 - not a defined member of a structure.

Only structure (or union) member names can appear on the right of a -> or a . operator.

7.39 ERROR - 36 - expecting a primary here.

The compiler is looking for a primary (such as a variable name) in an expression. This is another error that can be confusing because it is really a negative statement. The compiler believes it is in a function body (so expressions are allowed) and has decided that the current input is not a declaration or a specific statement (eg "while" or "if") so it must be an expression. It has further decided that the current input is not an operator and so it must be a primary! One particularly

obscure case which can cause this message sometimes is an unclosed comment (if you think through it when it happens you can probably understand why).

7.40 ERROR - 37 - undefined variable.

The name of an undefined variable has been used in an expression. Define it as a global or at the top of the function.

7.41 ERROR - 38 - need a type name.

The "sizeof" and the "cast" operators will only work on the name of a type, either one of the basic types (eg unsigned int) or one declared using typedef. They won't accept an "anonymous type" such as "int **(*(*())[])()" and the "sizeof" operator won't accept a variable name (because you can write the program with named types more clearly and help keep the compiler small).

7.42 ERROR - 39 - need a constant expression.

The compiler will accept and evaluate any constant expression here, but it has just encountered something which isn't constant.

7.43 ERROR - 40 - can only call functions.

A left parenthesis '(' after a variable name looks like an attempt to call that variable as a function, and to do that the variable must be declared as a function returning something. Maybe there is a missing '+' or somesuch between the variable and the parenthesis or a missing indirection '*' in front of the variable.

7.44 ERROR - 41 - : does not follow a ? properly.

The compiler has found a colon where it shouldn't be. A ':' can appear after a label name or after a "case" expression, or it can appear in an expression as part of the e1 ? e2 : e3 conditional expression. This one seems to be part of an expression, but doesn't match a '?'. Check the precedence of the expression if you think it does match a '?' !! The simplest way is to put extra parentheses in to make sure. Remember that conditional expressions group right-to-left if they are nested.

7.45 ERROR - 42 - Destination of an assignment must be an lvalue.

A special case error message to let you know as soon as possible that the left-hand side of this assignment expression is not an lvalue. The rules about what an lvalue is are given in the C Reference Manual, but roughly speaking an lvalue is a memory address which can be stored into. Remember in particular that an array name and a function name are NOT lvalues.

7.46 ERROR - 43 - need a : to follow a ? - check bracketting.

The compiler expected to find a colon as part of an e1 ? e2 : e3 conditional statement and it hasn't done so. Remember in particular that e2 cannot be an assignment expression unless it is bracketted and that conditional expressions group right-to-left if they are nested.

7.47 ERROR - 44 - need a pointer.

In order to use indirection on an expression, it must be a pointer-to something. The indirection may be an explicit '*' operator, but it could also be implicit in a '->' operator or an array element reference e1 [e2]. Note that the Hisoft-C compiler does NOT permit an integer before a '->' as discussed in section 14.1 of the C Reference Manual. If you want to access a structure at an absolute store address, use a "cast" to write the expression:

```
typedef  char *char_ptr;
(cast(char_ptr) 0x005C) -> fcb_filename [ 0 ] = 'A';
```

The idea is to help prevent mistakes which can corrupt memory, and to make it easier to transfer programs from one type of machine to another.

7.48 ERROR - 45 - illegal parameter type.

There are certain types which cannot be used as parameters to functions. These types are structures, unions, and other functions. You CAN pass a pointer to any of these types.

7.49 ERROR - 46 - RESTRICTION: Floating Point not implemented.

Sorry!

7.50 ERROR - 47 - cannot use this operator with float arguments.

This message will not occur as long as the previous one does.

7.51 ERROR - 48 - bad declaration.

Check what is allowed in a declaration.

7.52 ERROR - 49 - storage class not valid in this context.

In particular, there are no register or automatic globals.

7.53 ERROR - 50.

There is no error 50.

7.54 ERROR - 51 - duplicate declaration of structure tag.

The same identifier has been used twice as a structure tag.

7.55 ERROR - 52 - use a predeclared structure for parameters.

Instead of declaring the contents of a stucture in the parameter list, declare the structure first and give it a name (either a tag or by using typedef). Then use the name to declare the parameters. You will need the structure again to supply the actual arguments to the function, in any case.

7.56 ERROR - 53 - structure cannot contain itself.

You can include one kind of structure directly inside another, but to build a list using structures use pointers to structures, such as:

```
struct list
{
  int                   value;
  struct list  * next;
}
```

7.57 ERROR - 54 - bad declarator.

Check the syntax of a declarator in The C Reference Manual.

7.58 ERROR - 55 - missing ')' in function declaration.

A function declaration - as opposed to a function definition - must have a "()" after the name and may not have a parameter list. A function declaration just says that a particular identifier is of type function-returning-something, whilst a definition contains the body of the function as well. Functions can only be defined at the outermost level of a program whereas a function declaration can occur as a member of a structure or a parameter etc.

7.59 ERROR - 56 - bad formal parameter list.

The formal parameter list of a function is just a list of identifiers. Additional type information is given in a separate declaration afterwards, just before the body of the function.

7.60 ERROR - 57 - type should be function.

The compiler believes that this is a function definition, and has discovered that the function name is not of function-returning-something type. Check what is allowed in the C Reference Manual.

7.61 ERROR - 58.

There is no error 58.

7.62 ERROR - 59.

There is no error 59.

7.63 ERROR - 60 - LIMIT: no more memory.

The compiler uses the free space between the top of itself and the end of free memory (RAMTOP) to keep many things. The editor's text is kept here, and the compiled machine code of your program, and the text of the error messages, and workspace for the compiler. This space is now all full. You can make some room in several ways:

- use the #error control line to discard the error messages
- save your program to tape/microdrive/etc and use #include file
- compile the entire program and run it, rather than using #direct+
- simplify your program

7.64 ERROR - 61 - RESTRICTION: use assignment or blt() to initialise automatics.

The compiler does not support initialisation of automatic local variables. You should simply use an assignment statement instead. Please note that you CAN initialise static local variables and these should normally be used in preference to automatics except where the function is recursive or re-entrant. To initialise large objects like arrays you can also set up a static array with the required data and then use the built-in blt() function to copy the data into the automatic variable. Note that you cannot initialise automatic arrays or structures within C and this is not a "RESTRICTION".

7.65 ERROR - 62 - Cannot initialise this (disallowed storage class).

You cannot initialise names of types, structures etc as opposed to variables of these types.

7.66 ERROR - 63 - Cannot initialise this (disallowed type).

It is not permitted to initialise variables which are unions, functions etc.

7.67 ERROR - 64 - too much initialisation data.

There are more constants in the initializer-list than are needed to initialise the variable.

7.68 ERROR - 65 - bad initializer (need a '{').

Initializers for structures and for arrays must be enclosed in curly braces, even if they are only a single constant.

7.69 Common Mistakes in C Programs.

C is intended to be a systems programming language and is designed first and foremost to provide power to the user without runtime overheads. Because of this there are numerous ways to make mistakes in C programs and these can have startling effects, particularly if you are used to the protection of languages like Pascal. Debugging C is more like debugging assembler - save your source text before running a compiled program! This section lists some likely causes of programs that don't work, concentrating on those that won't show up immediately as error messages when the program is compiled.

First the disasters, crashes and what to look for:

1. Using an array index with a bad subscript and so storing into an arbitrary location. Note that C does not have subscript checking.

2. Assigning a bad value to a pointer and then using it to store a value; or incrementing a pointer past the region it should point to. Very similar to the array errors. Hisoft-C tries to prevent some errors by insisting on the types being correct.

3. Passing the wrong number of arguments to a function when it is called. This may cause a crash by using the value of an argument that wasn't there or by unbalancing the stack when the function returns. C doesn't check the number of arguments.

4. Calling a pointer-to-function variable (eg int (*ptr_to_func)()) which has a bad value, or hasn't been initialised. This is just a random jump.

5. Calling a function which hasn't been defined in direct mode.

Of course the bad values in these cases may not be obvious: they might be caused by any of the errors listed below.

Now for a general list of likely problems:

1. Capital letters are different to lower-case letters in identifiers.

2. Check that all comments are terminated.

3. Check that all string constants are terminated.

4. Check that all variables are initialised, particularly that a pointer points where it should before using it to store through (ie don't write *ptr = x; before writing ptr = y;).

5. Check CAREFULLY that "==" is used for equality tests, and "=" only for assignment.

6. Remember that all arrays start from index 0. An array declared as a[N] has just N elements going from index 0 to index N-1.

7. An array name evaluates to a pointer to the array, and in particular does not cause a copy of the array to be passed to a function.

8. There are no complete array or structure assignments. It must be done element-by-element (or by the built-in library function "move"). In particular the test (a == b) where "a" and "b" are arrays just tests whether they have the same base address, not the same contents. The test (s == "text") is almost certainly an error.

9. Remember that local variables and arguments of functions are allocated on the stack and are thrown away as soon as the function finishes executing.

10. All arguments to functions are passed by value. An explicit pointer must be used to pass a variable parameter.

11. Check the number and types of arguments supplied to functions, as no checking is performed. Check also that the arguments are in the right order.

12. Check that pointers are supplied where required (so that the argument can be changed by the function). An "&" operator may be required. In particular, an "&" is always required when passing the name of a structure as an argument to a function.

13. Even if a function has no arguments it must have "()" after it's name to call it. This will normally be caught as a type error, but the statement:

```
func_with_no_args ;
```

just causes the address of the function to be evaluated and thrown away. To call the function it is necessary to write:

```
func_with_no_args();
```

14. Remember that function return types may be changed automatically by the rules of C. In particular int may be truncated to char.

15. Remember that the end-of-file value EOF is -1 and cannot be tested against a char variable, int must be used for the result of getchar, getc etc.

16. Check whether --i or i-- is appropriate. In particular note that a normal loop of N times goes from 0 to N-1 and is written as:

```
for ( i=0; i<N; ++i)  do_something();
```

17. Remember that incrementing a pointer causes its binary value to be increased by the size of the object that it points to so that it points to the next object in an array. In particular remember that pointers to "int" increase by two bytes at a time. Remember also that any value added to a pointer will be multiplied by the size of the object first.

18. Arithmetic overflow is not an error, and is not tested for. This can be useful sometimes but means that you must include explicit checks if you need them.

19. A string is a pointer to an array of char (ie an address). You cannot use Pascal-like tests of equality on strings. In particular beware of typing "x" when you mean 'x'. The first is the address of a two byte array, the second is the ASCII value of a character.

20. A string has a zero byte on the end. Remember that when calculating array lengths, and remember to put one there if your program makes strings.

21. The precedence of operators can be surprising at times. Check the relational operations (== < etc) and remember that shift operators are of lower precedence than addition so that (hi<<8 + lo) doesn't mean what it seems to.

22. The order of evaluation of an expression is NOT specified and must NEVER be relied on.

23. An "else" belongs to the immediately preceding "if".

24. Execution flows through case statements into the next one. A "break" is needed to exit from the "switch" statement.

25. If a "switch" variable does not match any of the "case" expressions, then the statement is just bypassed.

26. There is no semicolon between a control statement (while, if, for) and the statement that it controls. Otherwise the empty statement is controlled.

27. It is normally wrong for a #define line to finish with a semicolon.

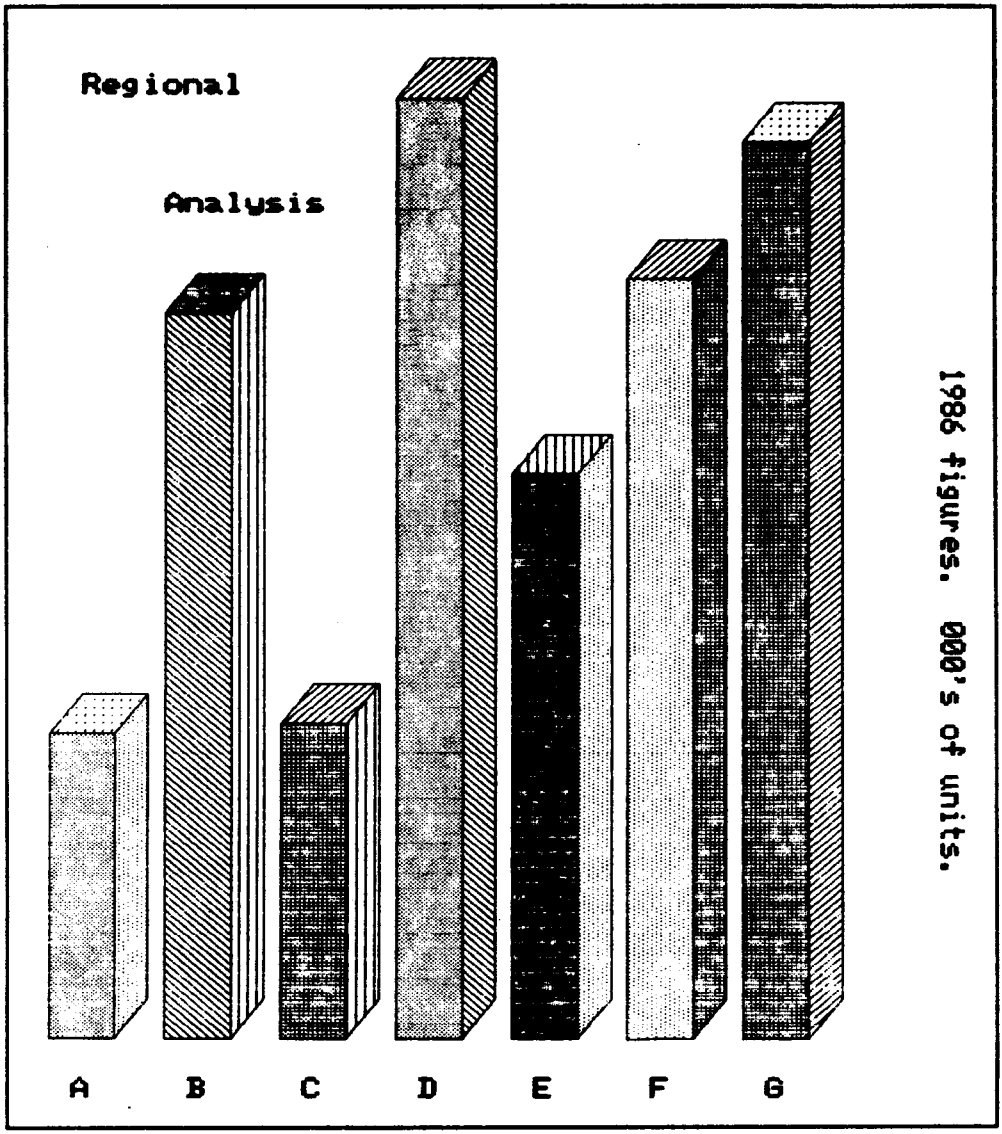
28. All arguments to scanf must be pointers. That is, they must be the ADDRESS of the variable where the result is to be stored.

29. There is no %u conversion for scanf; %d is used for both cases. Using one by mistake can produce obscure symptoms; because scanf tries to match against a 'u' character, fails and gives up early, which leaves junk in the remainder of the input variables.

30. Most scanf control strings should start with a space character. This is because it is likely that the previous input to scanf was terminated by pressing [ENTER] (because getchar() is buffered) and the newline character is still sitting in the input buffer.

    Don't put the space at the end of the control string because then scanf will carry on scanning until it finds something which isn't white space, before you have prompted for the next input.

HiSoft GSX Graphics



HISOFT GSX C LIBRARY

What is GSX ?

GSX is an extension to the CP/M Plus operating system that gives you a powerful graphics interface on your computer through the use of standard functions that are the same on all implementations. Both the Amstrad CPC 6128 and the Amstrad PCW8256/8512 have GSX supplied as standard under CP/M+ but the problem is that the calls to GSX are difficult to use and there is virtually no documentation available for them. This manual and the associated C GSX functions attempt to help by supplying a high-level C interface to GSX along with substantial documentation on using the routines.

Please read this manual all the way through at least twice and do try out the example programs before taking the plunge yourself.

What does GSX let you do?

With GSX you can:

1. Draw lines and any sort of polygon in 6 different styles.
2. Fill areas in 12 different fill styles.
3. Move a special graphics cursor and read the keyboard with one call.
4. Print text at any pixel position on the screen.
5. Write text in transparent or XOR mode.
6. Perform most of the CP/M Plus terminal emulation codes.
7. Display 5 sorts of 'markers' that are useful for drawing graphs.
8. Use colour on the CPC 6128.

If you are using a PCW or have an Epson compatible, Shinwa or Amstrad DMP-1 printer for your 6128 you also have the following features on the printer only:

1. Draw text in 12 different sizes, vertically and upside down on the printer.
2. Use 12 sizes of markers.

On Amstrad computers the CP/M operating system consists of just one file with extension .EMS but to program with GSX you need several files:

1. GSX.SYS
the machine independent part of GSX : about 2K long.
2. GENGRAF.COM the program that makes your programs load GSX before they run: about 2K long.
3. ASSIGN.SYS the file that tells GSX which driver to use : less than 1K long.
4. At least one device driver from the following:

DDSCREEN.PRL
The PCW series screen driver, 5K long: use the one on the disc with the GSX library on it; it will be better than the one on your system disc!

DDMODE0.PRL
The CPC mode 0 driver: 160 x 200 pixels with 16 colours. 20 characters per line. 4K long.

DDMODE1.PRL
The CPC mode 1 driver: 320 x 400 pixels with 4 colours. 40 characters per line. 4K long.

DDMODE2.PRL
The CPC mode 0 driver: 640 x 400 pixels with 2 colours. 80 characters per line. 4K long.

DDFXHR8.PRL
High-resolution printer driver for PCW printers. 960 x 1368 resolution. 15K long.

DDFXLR8.PRL
Low-resolution printer driver for PCW printers 480 x 672 resolution. 12K long.

Or one of the printer drivers on your system disc. If you have a CPC 6128 see side 3 of your system disc. Type the file DRIVERS.GSX. This gives information on the various printer drivers.

5. GSX.LIB
contains the C functions for calling GSX. This does not call any other libraries; so for some programs it is the only one you need.

6. GSX.H
contains the declarations of the C variables that are used by GSX.LIB

The Format of ASSIGN.SYS

ASSIGN.SYS is an ordinary text file which *must* be present when using GSX; it is composed of lines with the following format:

```
nn d:fname
```

where nn is the number of the device. Conventionally
01-09 are used for screens.
10-19 are used for plotters.
20-29 are used for printers.

d is an extended drive name specifier. This can be just A: or B: or M: or alternatively @: which means the default drive. So to use the screen on the PCW it is normal to use

```
01 @:DDSCREEN
```

and to use the High-resolution mode on the CPC

```
01 @:DDMODE2
```

or

```
21 @:DDFXHR8
```
to use the printer on the PCW.

You can use more than one device at a time, but this uses more memory and you can easily end up with a message 'file too large' because the device drivers must load in the top 16K of memory.

HOW TO COMPILE AND RUN C PROGRAMS USING THE GSX LIBRARIES

When compiling your program you need to use the compiler's `#data` pre-processor command. This is necessary because GSX runs at the top of memory and your program normally uses this area for its variables and so would corrupt GSX. The minimum value you can use in `#data` is given when you compile, but for small programs use

```
#data 0x4000
```

at the front of your program.

The format of your file should be of the form:

```
#data 0x4000

#include stdio.h
#include gsx.h

/* any other includes of .h files */

/* your functions */

main ()
{
/* your main program */
}

#include ?gsx.lib?
```

```
/* other includes of libraries including

#include ?stdio.lib?

 if required */
```

It is **essential** that the `#data` is present and **before** the `#include` directives.

Then to compile a typical program like BAR.C (which is on your master disc) use:

```
HC BAR.C[ENTER]
```

When your program compiles correctly use:

```
GENGRAF BAR [ENTER]
```

to add the GSX loader to `BAR.COM`, then

```
BAR [ENTER]
```

will run your program. Try this now!

It is often a good idea to create a `.SUB` file to do the compilation and `GENGRAF` particularly if you are using the `M:` drive on the PCW.

GSX CONCEPTS

Before describing the function calls in detail its necessary to be aware of the concepts behind GSX.

At the lowest level your program 'talks' to GSX by passing the address of an array called `contrl_ptr`. This in turn contains the addresses of 5 further arrays: `contrl, intin, intout, ptsin, ptsout`. Normally you don't need to know about these, but they must be present in your program and are defined in `GSX.H`.

Normally a GSX program opens a '**workstation**', does some GSX calls and then closes the workstation. So that GSX can tell different workstations apart, opening the workstation returns a '**handle**' that must be used in all subsequent calls.

The **close workstation** call is used to clear the screen and put the cursor etc. back to normal for the screen and to actually do the printing on printers.

As far as the user is concerned the screen or printer page is considered to have (0,0) in the bottom left to (32767,32767) in the top right, regardless of the screen or printer physical resolution.

COLOURS

This section is mostly irrelevant when using a PCW.
Colours are referred to by colour indices. These are 0,1,2.. depending on how many colours can be on the screen at any one time.
In mode 0 this is 16 colours; in mode 1, 4 colours and in mode 0, 2 colours.

Colour index 0 is the normal background colour and Colour index 1 is the foreground colour.

The default colours are as follows:

0 black
1 red
2 green
3 blue
4 cyan
5 yellow
6 magenta
7 white

8-15 are also initially white.

Different 'inks' can be associated with a colour by specifying the proportions of red, green and blue. The device driver then does its best to match this colour from the available pallette on the CPC 6128. When using mode 2 it is generally a good idea to change the ink of index 1 as this is dark red and thus is difficult to read against a black background on a colour monitor and almost impossible to read on a monochrome one.

How and what GSX prints on the screen depends on several 'variables'. These are:

1. Line type:

There are five different sorts of line type:

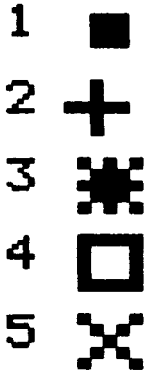
1. solid
2. dash
3. dot
4. dash,dot
5. long dash

There is also the **line colour index**, which gives the colour in which the lines are drawn.

2. **Marker type:**

Markers give you the ability to plot graphs in a more interesting way than just using dots. The different types are as follows:

PICTURE C1



There is also the **marker colour index** which gives the colour in which the markers are drawn.

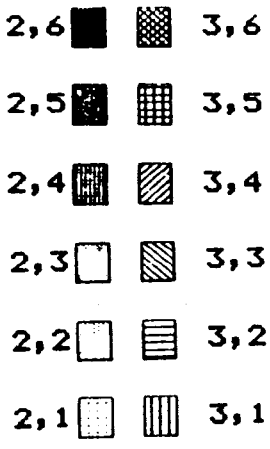
3. Fill interior style

0. hollow
1. solid
2. pattern
3. hatch

Associated with **fill interior styles** pattern and hatch there is a **'fill style index'**.

The following table shows the patterns generated for different styles and indices. 2,1 indicates an **fill interior style** of 2 and a **fill style index** of 1.

PICTURE 82



There is also the **fill colour index** which gives the colour that areas are filled with and the **text colour index** which is the colour in which graphics text is drawn.

For printers there is also the **text rotation angle.**

Writing mode

There are four different writing modes:

1	Replace	normal over- printing
2	Transparent	leaves any already printed pixels
3	XOR	reverses the bits
4	Reverse transparent	

The current **writing mode** is used in all the graph print routines.

THE GSX FUNCTIONS

1. OPEN WORKSTATION

```
v_opnwk( work_in, handle, work_out)
int *work_in,*handle,*work_out;
```

This is in fact the most complicated function to call. It opens a workstation given the initial settings of the colours etc. given in the array `work_in`. Normally 1 is used for each value. The handle to use is returned in the variable `handle`.

The elements of the array work_in are as follows:

`intin[0]`	device number.
`intin[1]`	line type.
`intin[2]`	line colour index.
`intin[3]`	marker type.
`intin[4]`	marker colour index.
`intin[5]`	text face. Must always be 1 on Amstrads.
`intin[6]`	text colour index.
`intin[7]`	fill interior style.
`intin[8]`	fill style index.
`intin[9]`	fill colour index.
`intin[10]`	not used.

e.g.

```
FAST int work_in[11], work_out[57];
FAST int handle;

 work_in[0] = 1;   /* device number 1 the screen */
 for (i=1; i<10; ++i)
        work_in[i] =1;

 v_opnwk(work_in,&handle, work_out);
```

Note that the device is the number given in the `ASSIGN.SYS` file and that textface will always be 1 since the Amstrad drivers do not support any other typefaces.

The open workstation call also return the following in the `work_out` array:

`work_out[0]`	pixel width of device -1
`work_out[1]`	pixel height of device -1
`work_out[2]`	0= device is capable of producing a continously scaled image. 1= otherwise.
`work_out[3]`	width of one pixel in microns
`work_out[4]`	height of one pixel in microns
`work_out[5]`	number of character heights
`work_out[6]`	number of line types
`work_out[7]`	number of line widths
`work_out[8]`	number of marker types
`work_out[9]`	number of marker sizes
`work_out[10]`	number of typefaces
`work_out[11]`	number of fill patterns
`work_out[12]`	number of hatch styles
`work_out[13]`	number of colours that can be displayed on the device at once.
`work_out[14]`	number of Generalised Drawing Primitives (GDPs). Sadly for the Amstrad device drivers this is always 1.
`work_out[15]`- `work_out[24]`	list of the GDPs supported. This is always just the Bar.
`work_out[25]`- `work_out[34]`	list of attributes associated with each GDP. Not useful on Amstrads.
`work_out[35]`	1= device supports colour. 0= No colour capability
`work_out[36]`	1= text rotation available 0= no text rotation

`work_out[37]`	1= fill area available 0= no area fill possible
`work_out[38]`	1= cell array capability 0= no cell array capability
`work_out[39]`	total number of colours in palette (2= monochrome)
`work_out[40]`	number of locator devices (0= just keyboard)
`work_out[41]`	number of valuator devices. Not applicable to Amstrads.
`work_out[42]`	number of choice devices. n/a
`work_out[43]`	number of string devices. n/a
`work_out[45]`	minimum character width.
`work_out[46]`	minimum character height.
`work_out[47]`	maximum character width.
`work_out[48]`	maximum character height.
`work_out[49]`	minimum line width.
`work_out[51]`	maximum line width.
`work_out[53]`	minimum marker width.
`work_out[54]`	minimum marker height.
`work_out[55]`	maximum marker width.
`work_out[56]`	maximum marker height.

CLOSE WORKSTATION

```
v_clswk(handle)
int handle;
```

For screen devices clears the screen and re-displays the cursor etc. For printers this causes the current page to be output. Should be used before finishing your program. Because this clears the screen it is usual to place a `getchar();` before it so that the screen can be viewed and then the screen cleared when `[RETURN]` is typed.

e.g.

```
 v_clswk(handle);
```

CLEAR WORKSTATION

```
v_clrwk(handle)
int handle;
```

Clears the device. On printers all data output but not printed is 'forgotten'.

e.g.

```
 v_clrwk(handle);
```

UPDATE WORKSTATION

```
v_updwk(handle)
int handle;
```

On screens, does nothing as the screen is always up to date. On printers, causes the current page to be printed.

e.g.

```
v_updwk(handle);
```

POLYLINE

```
v_pline(handle,count,pxyarray)
int handle,count,*pxyarray;
```

Draws a series of lines on the screen; the number of points is given by the parameter `count`. `pxyarray` is a pointer to an array that contains the points. The current **writing mode, line type** and **line colour** index are used.

e.g. Given the definitions:

```
int p[6];
```

then

```
        p[0]=0;     p[1]=0;
        p[2]=10000;p[3]=0;
        p[4]=0;     p[5]=10000;

   v_pline(handle,3,p);
```

will then draw a line from (0,0) to (10000,0) and another one from (10000,0) to (0,10000).

POLYMARKER

```
v_pmarker(handle,count,pxyarray)
int handle,count,*pxyarray;
```

Draws a series of markers on the screen; the number of markers is given by the parameter `count`. `pxyarray` is a pointer to an array that contains the points on which the markers are based. The current **writing mode, marker colour, marker height** and **marker type** are used.

This is used in a way exactly analagous to `v_pline`. Note that on the screen there is only one marker height.

WARNING: There appear to bugs in the printer drivers that cause markers not to be printed unless `v_gtext` (see below) is called previously.

GRAPHIC TEXT

```
v_gtext(handle,x,y,string)
int handle,x,y;
char *string;
```

This function writes a string starting at graphics position `(x,y)`. `string` is a pointer is to be printed. The string must be null terminated.

This uses the current **writing mode**, and **text colour**. On printers the current **rotation angle** is used.

e.g.

```
gtext(handle,16000,16000,"HiSoft");
```

writes `HiSoft` near the middle of the screen/page.

FILL AREA

```
v_fillarea(handle,count,pxyarray)
int handle,count,*pxyarray;
```

This fills a polygon specified in the `pxyarray`. The number of vertices is specified by the parameter `count` and `pxyarray` is a pointer to the array containing the points, in exactly the same the form as `v_pline`. The device driver draws a line from the last point to the first point so that the polygon is always closed.

The area is filled using the current **fill area colour**, **fill style**, and **writing mode**, and the outline of the polygon is drawn using a solid line in the current **writing mode** of the current **fill area colour**.

BAR

```
v_bar(handle,pxyarray)
int handle,*pxyarray;
```

This draws a bar given `(pxyarray[0],pxyarray[1])` as one corner and `(pxyarray[2],pxyarray[3])` as the opposite corner.

The area fill attributes are used as with `fillarea`.

e.g. given the definition:

```
int p[4];
```

then

```
p[0]=p[1]=10000;
p[2]=p[3]=20000;
v_bar(handle,p);
```

draws a rectangle near the middle of the screen.

SET CHARACTER HEIGHT

```
vst_height(handle,height,char_width,
                  char_height,cell_width,cell_height)
int  handle,height,*char_width,
                *char_height,*cell_width,*cell_height;
```

This sets the size of the text characters printed on a printer. There are 12 different heights.

The `height` parameter is the size of the characters given in terms of graphics points.

The `char_width` and `char_height` parameters are maximum size of the actual characters and the `cell_width` and `cell_height` are the size of the cells in which the characters are printed.

The following table gives the different heights given by the PCW printer using the High resolution driver:

yheight	char size(mm)	cell size(mm)
1 - 383	1 x 1	1 x 1
384 - 574	2 x 3	3 x 3
575 - 766	3 x 4	4 x 4
767 - 958	4 x 6	5 x 6
959 - 1149	5 x 7	6 x 7
1150 - 1341	6 x 8	8 x 8
1342 - 1533	7 x 10	9 x 10
1534 - 1724	8 x 11	10 x 11
1725 - 1916	10 x 13	11 x 13
1917 - 2107	11 x 14	13 x 14
2108 - 2299	12 x 15	14 x 15
2300 - 10000	13 x 17	15 x 17

The following table gives the different heights available using the Low resolution driver:

yheight:	char size(mm)	cell size(mm)
1 - 780	2 x 3	3 x 3
781 - 1170	4 x 6	5 x 6
1171 - 1560	6 x 8	8 x 8
1561 - 1950	8 x 11	10 x 11
1951 - 2340	11 x 14	13 x 14
2341 - 2730	13 x 17	15 x 17
2731 - 3120	15 x 20	18 x 20
3121 - 3510	17 x 22	20 x 22
3511 - 3900	19 x 25	23 x 25
3901 - 4291	21 x 28	25 x 28
4292 - 4681	23 x 31	28 x 31
4682 - 1000	25 x 34	30 x 34

SET ROTATION

```
int vst_rotation(handle,angle)
int handle,angle;
```

Again this is a function that is only applicable to printers. The `angle` parameter is a measure of the angle at which text will be printed. Normally this is 0 corresponding to the usual East to West. 900 corresponds to upwards; 1800 to upside down and 2700 to down the paper. If a different value is given then the closest match is taken.

The value returned is the set value.

e.g.

```
vst_rotation(handle,1800);

v_gtext(handle,15000,16000,"This is Upside down");
```

prints

```
uʍop ǝpısdn sı sıɥ⊥
```

COLOUR

```
vs_color(handle,index,rgb_in)
int handle,index, *rgb_in;
```

This is only useful on CPC 6128 screens; the `index` can be:

0 - 1	in mode 2
0 - 3	in mode 1
0 - 15	in mode 0

`rgb_in` is a pointer to an array containing the red, green and blue proportions of

the various colours defined.These vary from 0 to 1000. Thus the default for index 0 is 1000,0,0.

To change index 1 to be white use:

```
int white[3]={1000,1000,1000};
vs_color(handle,white);
```

and to set ink 8 to be 'dark' blue use:

```
int dark_blue[3]={0,0,300};
vs_color(handle,dark_blue);
```

The default colours are:

0 black
1 red
2 green
3 blue
4 cyan
5 yellow
6 magenta
7 white

8-15 are also initially white.

SET LINE TYPE

```
int vsl_type(handle,style)
int handle,style;
```

This function sets the line type that is used. Valid `style` arguments are:

1. solid
2. dash
3. dot
4. dash,dot

5. long dash

This function returns the style that has been set. If an illegal value is used solid (1) is set.

e.g. After

```
vs_ltype(handle,4);
```

then lines will be drawn using dashes and dots.

SET LINE COLOUR

```
int vsl_color(handle,color_index)
int handle,color_index;
```

This functions sets the `color_index` in which lines will be displayed. Normally the default is index 1 (normally red on the CPC and 'white' on the PCW) (this is set by the open workstation call). It returns the colour that has actually been set, so that if you try to set a colour index that does not exist for a particular screen you will get the highest value possible.
Thus if using either MODE0 or MODE1 on a CPC machine and assuming that colour index 3 has not been redefined using `vs_color` (see above) then

```
vsl_color(handle,3);
```

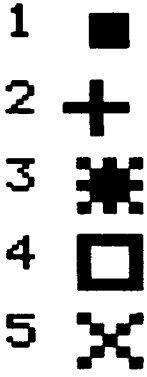
will cause lines to be drawn in blue.

SET MARKER TYPE

```
int vsm_type(handle,symbol)
int handle,symbol;
```

This function sets the current marker type to be one of

PICTURE C2



As usual this returns the type that is set; this is asterisk (3) if an illegal type is passed.

e.g.

```
vsm_type(handle,5);
```

sets the marker type to be a diagonal cross.

SET MARKER HEIGHT

```
int vsm_height(handle,height)
int handle,height;
```

This sets the size of markers to be displayed on the printer. There are 12 different sizes available with the two drivers. The PCW High resolution driver gives the following sizes:

yheight	approx height in mm
1 - 383	1
384 - 574	3
575 - 766	4
767 - 958	6
959 - 1149	7
1150 - 1341	8
1342 - 1533	10
1534 - 1724	11
1725 - 1916	13
1917 - 2107	14
2108 - 2299	15
2300 - 10000	17

The PCW Low resolution driver gives the following sizes:

yheight	approx height in mm
1 - 780	3
781 - 1170	6
1171 - 1560	8
1561 - 1950	11
1951 - 2340	14
2341 - 2730	17
2731 - 3120	20
3121 - 3510	22
3511 - 3900	25
3901 - 4291	28
4292 - 4681	31
4682 - 10000	34

SET MARKER COLOUR

```
int vsm_color(handle,color_index)
int handle,color_index;
```

This functions sets the `color_index` in which markers will be displayed. It returns the colour that has actually been set, so that if you try to set a colour index that does not exist for a particular screen you will get the highest value possible.

Thus if using either MODE0 or MODE1 on a CPC machine and assuming that colour index 2 has not been redefined using `vs_color` (see above) then

```
vsm_color(handle,2);
```

will cause markers to be drawn in green.

SET TEXT COLOUR

```
int vst_color(handle,color_index)
int handle,color_index;
```

This function sets the `color_index` that text is drawn with when using both the `v_gtext` and `v_curtext` procedures on the CPC machines.

e.g.

```
vst_colour(handle,5);
```

will cause subsequent text to be output in yellow.

Note that you should always reset this to be 1 before you do a `v_clswk` and then return to CP/M because otherwise you will end up with a blue foreground and background.

SET FILL INTERIOR

```
int vsf_interior(handle,style)
int handle,style;
```

This function sets the fill interior style to be one of:

0 - hollow
1 - solid
2 - pattern
3 - hatch

For the pattern and hatch options the fill style index (see below) is also used.

So that for example:

```
vsf_interior(handle,0);
```

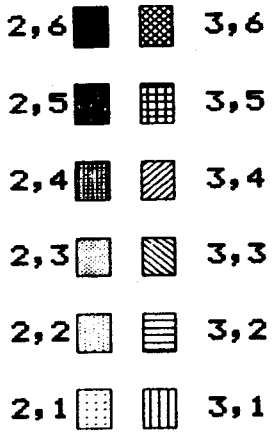
will cause subsequent fills to be done in current background colour.

SET FILL STYLE INDEX

```
int vsf_style(handle,style_index)
int handle,style_index;
```

This changes how areas are filled if pattern (2) or hatch (3) fill interior styles are in use.

The following table shows the patterns generated for different styles and indices. `2,1` indicates an **fill interior style of 2 and a fill style index** of 1.



PICTURE B1

SET FILL COLOUR

```
int vsf_color(handle,color_index)
int handle,color_index;
```

This function sets the colour `index` that areas are filled with. It returns the colour that has actually been set, so that if you try to set a colour index that does not exist for a particular screen you will get the highest value possible.

Thus if using either MODE0 or MODE1 on a CPC machine and assuming that colour index 2 has not been redefined using `vs_color` (see above) then:

```
vsf_color(handle,2);
```

will cause areas to be filled in green.

PLACE GRAPHIC CURSOR AT LOCATION

```
v_dspcur(handle,x,y)
int handle,x,y;
```

Displays the graphics cursor at the graphics location `(x,y)`. The graphics cursor is of the 'cross hair' type. It is drawn using the XOR mode so that in can subsequently be removed.

e.g.

```
v_dspcur(handle,16384,16384);
```

displays the graphics cursor in the middle of the screen.

REMOVE LAST GRAPHIC CURSOR

```
v_rmcur(handle:INTEGER);
```

This removes the last graphics cursor displayed using `v_dspcur`. It is removed by writing the graphics cursor in XOR mode. Thus if it has been cleared already it will in fact be redrawn.

SET INPUT MODE

```
vsin_mode(handle,dev_type,mode)
int handle, dev_type, mode;
```

This function is used to choose between using `vrq_locator` and `vsm_locator`. You must set `dev_type` to be 1 on the Amstrad computers. If `mode` is

1 then `vrq_locator` is to be used.
2 then `vsm_locator` is to be used.

e.g. `vsin_mode(handle,1,2)` is used before calling `vsm_locator`.

GET LOCATOR

```
vrq_locator(handle,x,y,xout,yout,term)
int handle,x,y,*xout,*yout,*term;
```

This routine displays a graphics cursor at graphics position `(x,y)` and then lets the user move it using the cursor keys. Using the cursor keys alone moves the cursor in quite large steps; using the shift keys as well will cause it to move in small steps. When the user presses a non-cursor key this is returned in `term` as the value of the function with `(xout,yout)` containing the final co-ordinates of the graphics cursor which is then removed.

e.g.

```
vrq_locator(handle,16384,16384,&xpos,&ypos,&key);
printf(
  "You moved the cursor to (%d,%d) and pressed %c \n",
                xpos,pos,key);
```

If you have used the samplelocator function (see below) then you must use `vsin_mode(handle,1,1);` before calling this function.

SAMPLE LOCATOR

```
int vsm_locator(handle,x,y,xout,yout,term)
int handle,x,y,*xout,*yout,*term;
```

This function is used to find the current position of the graphics cursor, which must be displayed separately at `(x,y)` using `v_dspcur` (see above) first. The value returned as the result is either:

0	nothing happened.
1	the cursor has been moved; the new cursor position will be in `(xout,yout)`
2	a non-cursor key has been pressed; the character is returned in the parameter `term`.

You *must* call `vsin_mode` (see above) before calling this function.

Here is an extended example of the use of `samplelocator` in the form of an entire program:

```
#data 0x4000

#include "stdio.h"
#include "gsx.h"

main ()
{

FAST st;                    /* status returned from sample locator call */
FAST int i, work_in[11], work_out[57];
FAST int  term;             /* any characters typed */
FAST int handle;
FAST int p[]={10000,10000,10000,10000};
               /*(p[0],p[1]) is the last point and (p[2],p[3])
                                    is the current point */

 work_in[0] = 1;
 for (i=1; i<10; ++i)
      work_in[i] =1;

 v_opnwk(work_in,&handle, work_out);

 vsin_mode(handle,1,2);                 /* set sampling mode */
 v_dspcur(handle,p[0],p[1]);            /* display the cursor initially */

do
{
  switch (st=vsm_locator(
               handle,p[0],p[1],&p[2],&p[3],&term))
{  case 0 : break;                      /* nothing has happened so do nothing */
   case 1 :                             /* the cursor has been moved */
      v_rmcur(handle);                  /* remove the cursor */
      v_pline(handle,2,p);              /* draw a line between the old
                                           and new points */
      v_dspcur(handle,p[2],p[3]);       /*display the new cursor */
          p[0]=p[2]; p[1]=p[3];         /* make the old point the current one */
          break;

  case 2: putchar(term); break;         /* echo any characters typed */

}
}
while (!( (i==2) && (term=='E')));      /* finished if E is typed */

 getchar();
 v_clswk(handle);                       /* clear the screen etc */

}

#include ?gsx.lib?
```

THE TERMINAL 'ESCAPE' FUNCTIONS

These give text, cursor positioning etc in a machine independent way providing most of the facilities of the CP/M Plus terminal emulator on the Amstrad machines.

INQUIRE CHAR CELLS

```
vq_chcells(handle,rows,columns)
int handle,*rows,*columns;
```

This returns the size of the screen in character positions. This varies depending on whether you are using a PCW or a CPC and whether or not the status line is enabled.

Thus given the declaration

```
int r,c;

inqcharcells(handle,&r,&c)
```

will normally return r=31 and c=90 on a PCW with the status line enabled.

EXIT ALPHA MODE

```
v_exit_cur(handle)
int handle;
```

This simply switches the cursor OFF on the Amstrad machines.

ENTER ALPHA MODE

```
v_enter_cur(handle)
int handle;
```

This simply switches the cursor ON on the Amstrad machines.

ALPHA CURSOR UP

```
v_curup(handle)
int handle;
```

This moves the cursor up one line; if it is already on the top line it does not move.

ALPHA CURSOR DOWN

```
v_curdown(handle)
int handle;
```

This moves the cursor down one line; if it is already on the bottom line it does not move.

ALPHA CURSOR RIGHT

```
v_curright(handle)
int handle;
```

This moves the cursor right one character; if it is already on the right most column it does not move.

ALPHA CURSOR LEFT

```
v_curleft(handle)
int handle;
```

This moves the cursor left one character; if it is already on the left most column it does not move.

HOME ALPHA CURSOR

```
v_curhome(handle)
int handle;
```

This moves the cursor to the top left corner of the screen.

ERASE TO END OF ALPHA SCREEN

```
v_eeos(handle)
int handle;
```

Erases the screen from the current alpha position to the end of page. The cursor does not move.

ERASE TO END OF ALPHA TEXT LINE

```
v_eeol(handle)
int handle;
```

Erases the screen from the current cursor position to the end of the line. The cursor does not move.

DIRECT ALPHA CURSOR ADDRESS

```
vs_curaddress(handle, row, column)
int handle, row, column;
```

This function moves the cursor to alpha position `(row,column)` with `(1,1)` as the top left hand corner. If the position is not on the displayable screen it is moved to the nearest value that can be displayed.

OUTPUT CURSOR ADDRESSABLE ALPHA TEXT

```
vs_curtext(handle, string)
int handle, *string;
```

This outputs a text string starting at the current alpha cursor position. `string` is a pointer to the string. The string must be null terminated.

e.g.

```
vs_curaddress(handle,1,10);
vs_curtext(handle,"A HiSoft Powerful Tool");
```

writes `A HiSoft Powerful Tool` on the top line of the screen 10 characters in from the left.

REVERSE VIDEO ON

```
v_rvon(handle)
int handle;
```

This function causes all subsequent alpha text to be displayed in inverse video until `v_rvoff` is called.

REVERSE VIDEO OFF

```
v_rvoff(handle)
int handle;
```

this causes subsequent alpha text to be printed in normal video.

INQUIRE CURRENT ALPHA CURSOR ADDRESS

```
vq_curaddress(handle,row,column)
int handle,*row,*column;
```

This function returns the current alpha cursor position in the variables `(row,column)` with (1,1) as the top left hand corner.

INDEX