HiSoft C Interpreter

for the Atari ST

HiSoft

High Quality Software

HiSoft C

Interpreter for the Atari ST

System Requirements:
Atari ST Computer with a mouse and a disk drive

Program Copyright © Loriciels 1988, 89
English translation of manual Copyright © HiSoft 1989, 90

HiSoft C Version 2 April 1989

Printing History:
1st Edition April 1989 (ISBN 0 948517 16 6)
2nd (revised and corrected Edition February 1990 (ISBN 0 948517 25 5)

Set using an Apple Macintosh™ with Microsoft Word™ & Aldus Pagemaker™

ISBN 0 948517 25 5

Table of Contents

0 Introduction

The C language has always been a *compiled* language. This means that the programmer has to type in his or her program and then compile it to change the program into machine code.

Unfortunately this process normally takes several minutes and only then can you execute your program.

HiSoft C takes a different approach.

It is a C *Interpreter*. This means you type in your program and execute it immediately as with a BASIC interpreter. There's no waiting around for compilers and linkers.

To do this, **HiSoft C** has a powerful GEM-based editor which lets you edit up to eight files at once. You can even create modular programs and then, with a simple click of the mouse, run these modules together.

This gives you an easy-to-use environment where you can spot errors much more quickly than with a compiler. You can even single step and put breakpoints in your code.

HiSoft C gives the C language the accessibility of BASIC without losing the language's power. So **HiSoft C** is aimed at two categories of programmers:

- Beginners will find **HiSoft C** gives them a very easy way to learn the C language.

  C is the language around which the Atari ST is based and is the language of choice for many professional programmers. Many newcomers to the ST decide that they would like to try out this powerful language. So they buy a compiler, but it seems to take forever to get their first program to run because of numerous compiler or linker errors.

  This interpreter lets beginners discover the C language more gently, using an environment that reminds them (or you!) of BASIC.

- Experienced programmers will find **HiSoft C** provides a faster way to develop their programs.

Compiled and assembly language functions can be loaded and used with interpreted modules. **HiSoft C** also supports almost the full C language as described in Kernighan & Ritchie "The C Programming Language" (First Edition).

Developers can use **HiSoft C** to develop their programs with all the advantages this gives and then, when their application is finished, just compile to a finished product.

Porting programs written with the interpreter is easy as long as you respect the standard rules of programming. Nevertheless, each implementation of C (whether a compiler or interpreter) has its own peculiarities and extensions; so some work may be needed. This is the case when moving code between a compiler and the interpreter and between two compilers.

HiSoft C has a library of 460 functions. These are detailed in **Section 4** and include the usual ANSI, UNIX, C, GEM and DOS standard functions. In addition, **HiSoft C** has a toolbox of functions which are both simple and powerful, to make programming with menus, dialog boxes and windows much easier.

So, thanks to **HiSoft C**, you can easily write programs that use all the facilities of your machine, with or without GEM. Finally, if you need faster running code you can use your favourite compiler (Lattice C from HiSoft, we hope!) to speed up your programs.

0.1 Always make a back-up

Before using **HiSoft C** you should make a back-up copy of both the distribution disks and put the original away in a safe place. They are not copy-protected to allow easy back-up and to avoid inconvenience. The disks may be backed-up using the Desktop or any back-up utility. The disks are single-sided but may be used in double-sided drives.

If you have a double-sided disk drive you can copy the contents of both disks on to one double-sided disk. If you have a hard disk, just drag the files to a folder on your hard disk.

Before hiding away your master disks make a note in the box below of the serial number written on them. You will need to quote this if you require technical support.

Serial No:

0.2 Registration Card

Enclosed with this manual is a registration card which you should fill in and return to us after reading the licence statement. Without it you will not be entitled to technical support or upgrades. Be sure to fill in all the details, especially the serial number and version number. The version number is given in the About box when you run **HiSoft C**.

0.3 HiSoft C Disk Contents

The **HiSoft C** disk 1 contains the following files:

`HC.PRG`	the interpreter itself
`F1.IC - F11.IC`	supplementary files the interpreter needs
`README.TXT`	see below for details

The **HiSoft C** disk 2 contains the following folders:

`HEADER`	the standard C header files
`HELP`	the information files used by the editor help command
`EXAMPLES`	example C programs
`SOURCE`	the source code to the HiSoft C toolbox. See **Appendix D.1**
`DUMPFILE.*`	the files for the resource file of the `DUMPFILE.C` example
`RESOURCE.*`	the files for the rsource file of the `RESOURCE.C` example
`CHECKST.PRG`	a program to give details of your machine. Please include this information when writing to us.

0.4 The README File

As with all HiSoft products **HiSoft C** is continually being improved and the latest details that cannot be included in this manual may be found in the `README.TXT` file on the disk. This file should be read at this point, by double-clicking on its icon from the Desktop and then clicking on the Show button. You can print it by clicking on the Print button.

0.5 Using this manual

Different sections of this manual are aimed at different readers.

After this introduction, **Section 1** describes how to use the Editor and Interpreter and so everyone should read this.

Section 2 is an introduction to the C language for those of you who are new to the language.

Section 3 is an introduction for those new to programming in GEM, whether experienced or novice C programmers. This section also provides a description of the **HiSoft C** GEM toolbox for experienced GEM programmers.

Section 4 describes the 460 functions of the **HiSoft C** library. This is mainly designed for reference and includes a summary so you can find which function you need for a particular job.

Appendix A contains the answers to the Exercises set in the tutorial in **Section 2**. You will need to use this as you work through **Section 2**.

Appendix B is the reference language for the C interpreter and is aimed at the experienced programmer.

Appendix C covers the interpreter's error messages and is designed to be used as a reference if you don't understand an error message.

Appendix D covers porting programs to and from **HiSoft C** from and to other implementations.

Appendix E is the Bibliography which recommends some further reading on the C langauge and the ST itself.

Appendix F covers technical support and upgrades.

Finally, there's the index.

1 Using HiSoft C

This section describes how to write, modify and run a program with **HiSoft C**.

1.1 The Keyboard and the cursor

1.1.1 The Editor

As soon as you load **HiSoft C** a menu bar appears and a window covers most of the screen. **HiSoft C** uses GEM and the mouse to make your work easier. The menu lets you select the editor and execution options. The program that you type in will appear in the window.

At the top left of the window you will see a flashing block cursor. This indicates where the characters you type will be inserted.

The editor is a full-screen editor; that is to say you can move the cursor where you like within the window and immediately change the text as you wish.

HiSoft C lets you work with eight files at the same time. If you cannot see the file that you want at a given instant you can very easily switch from one to another.

1.1.2 Moving the cursor

You can move the cursor in all four directions by pressing the cursor keys labelled ← → ↑ and ↓ to the right of the keyboard. You can also use `Control S`, `Control D`, `Control E` and `Control X` respectively.

You can position the cursor by moving the mouse and clicking at the point to which you wish to move.

You can also use the vertical scroll bar and the window's arrow icons to change the cursor position.

The following keys are for rapid cursor movement within the text:

`Shift` ←	moves to the start of line
`Shift` →	moves to the end of line
`Control` ←	moves to the end of the previous word
`Control` →	moves to the start of the next word

1.1.3 The special keys

The following keys modify the text in some way:

`Delete`	deletes the character under the cursor.
`Backspace`	deletes the character to the left of the cursor.
`Return`	inserts a carriage return. If the cursor isn't at the end of the line the line is split at the cursor position and the cursor moves to the beginning of the new line.
`Control Return`	inserts a new line without splitting the current one.
`Tab`	inserts a tab character at the cursor position thus moving the cursor to the next tab position.
`Shift Tab`	moves to the previous tab position.
`Undo`	'Undoes' any changes to the current line.
`Help`	displays some help information on a specified subject.
`Home`	moves the cursor to the start of the C block. A C block is a sequence of instructions surrounded by curly brackets `{` and `}`.
`Shift Home`	moves the cursor to the end of the C block.
`Esc`	See **Section 1.1.6**.
`Alternate 1` to `Alternate 8`	select one of the eight possible files.

There's no need to remember these by heart; they can be displayed by selecting Cursor keys from the Help menu.

1.1.4 Function keys

The function keys when used with the `Shift` key work as ten macro function keys that can be modified by the user. See **Section 1.7.1**.

When the function keys are used without the `Shift` key they have the following meanings:

`F1`	Deletes to the end of the word : under the cursor. A word consists of a sequence of characters separated by blanks or the ends of lines.
`F2`	Insert the last word which has been deleted with `F1` to the right of the cursor. If `F3` was used since `F1` then the line deleted by `F3` is used.
`F3` or `Control Y`	Delete the line containing the cursor.
`F4`	Insert after the current line the last word deleted with `F1` or the last line deleted with `F3`.
`F5`	Insert an empty line after the current line and position the cursor there.
`F6`	Join the next line to the end of the current one by copying.
`F7`	Comment out the current line. If the line was originally commented out the comment is removed.
`F8`	Delete to the end of the current line. This may *not* be recovered with `F2` or `F4`.
`F9` or `Control-R`	Move the cursor 23 lines (a screen full) towards the start of the file.
`F10` or `Control-C`	Move the cursor 23 lines (a screen full) towards the end of text.

The function keys item from the Help menu may also be used to display a summary of these keys.

1.1.5 Menu short cuts

The major commands can be accessed from the keyboard without having to use the mouse to access the menus.

In general these combinations consist of either the `Alternate` or the `Control` keys with a letter or digit.

To the right of the menu entries is a single letter. If this is a capital (upper case) then this item may be selected with `Alternate` and the letter key. If this is in lower case then it can be selected with `Control` and the letter.

Here is a list of the key combinations and the corresponding menu item.

`Alternate A`	Abandon
`Alternate B`	Bottom of file
`Alternate F`	Find
`Alternate G`	Go to line
`Alternate I`	Insert file
`Alternate J`	Load project
`Alternate L`	Load file
`Alternate M`	Module list
`Alternate N`	Repeat find (Next)
`Alternate O`	Save options
`Alternate P`	Program Information
`Alternate Q`	Quit **HiSoft C**
`Alternate R`	Find & replace
`Alternate S`	Save as
`Shift Alternate S`	Save file
`Alternate T`	Top of file
`Alternate V`	Information on the variables
`Alternate X`	Run (eXecute)
`Alternate Z`	Go to last position
`Control 1`	Set mark 1
`Control 2`	Set mark 2
`Control 3`	Go to mark 1
`Control 4`	Go to mark 2
`Control C`	Page down
`Control D`	Cursor right
`Control E`	Cursor up
`Control R`	Page up
`Control S`	Cursor left
`Control X`	Cursor down
`Control Y`	Delete line

These combinations of keys may be modified to configure the keyboard in the way that you want; you can thus make the editor similar to any editor that you are used to.

See **Section 1.8.3** for details.

1.1.6 Keyword completion

This facility can save a lot of typing and looking up the syntax of functions.

You type only the beginning of the name, and the complete function call (including the parameters) is automatically created by the editor.

For example if you type:

```
strni
```

and then press the `Esc` key, the editor will display:

```
strnicmp(str1,str2,n)
```

on the screen, with the cursor one the first argument of thefunction. You are now in `Esc` mode. Look at the top right corner of thescreen. `Esc` is displayed, instead of `Ins`. The editor is waiting for you to type in the arguments of the function. After each parameter, you should press `Return`, and the cursor is positioned on the first character of the nextargument.

The parameter names which are written by the editor are automatically replaced by the new names you type. You don't need to erase the old names with `Backspace` or `Del`; just type the new one and press `Return`. If you don't want tochange the name of a parameter, just type `Return`.

You exit `Esc` mode when you have typed in all the parameters, or when you use cursor keys or select a menu item.

`Esc` mode can also be used with the C language keywords, as follows:

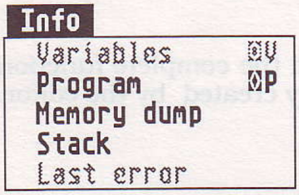
short cut	**statement**
`f`	`for`
`w`	`while`
`s`	`switch`

The list of the functions and parameters used by `Esc` mode is in file `F11.IC`. You can modify this as you wish.

1.1.7 Menu commands

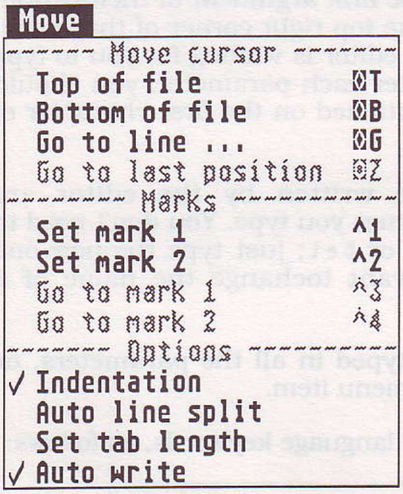
You know how to select a menu item, now we will examine the various menus in detail.

On the far left are the credits! The first entry under the Atari logo, which is equivalent to the Desk menu on many ST programs, is the credit for the author of the program.



In the Info menu on the far right, the Program info command displays a dialog box which gives some statistics for the current program. With this you can see how many lines your program has, the free memory figure, how much memory this program needs and where the cursor is in the text. The other commands on this menu will be discussed later.

The Move menu commands are also useful:



Top of file moves the cursor to the front of your program and Bottom of File moves to the end (oh good - you guessed).

When you click on the Go to line... item a dialog box appears and asks you to enter the line number that you want to move to. You are not allowed to move past the end of text.

There are a set of commands that some people find incredibly useful and other people find a waste of time. As they weren't difficult to program, they have been put in **HiSoft C**.

These are the marks. You set a mark in your program at the cursor position and then after you have moved the cursor you can return to the place that you marked. There are two different marks that you can use.

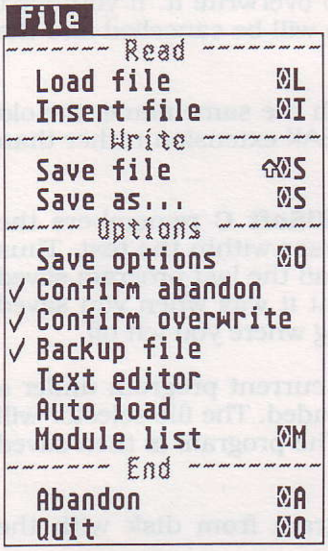
The command Set mark 1 sets the first mark at the cursor position; to return there use the Go to mark 1 command. Naturally Set mark 2 and Go to mark 2 perform the same operations for the second mark.

The Go to last position command is similar to the marks. It is as if a mark was automatically placed at the last cursor position that the text was changed. When you click on this command the cursor moves to the last place that the text was modified.

This is very useful when you are typing in a program. It lets you interrupt your typing, have a look at another part of the program and then continue typing where you were before.

Note that the marks can be used to move between modules. If you go to a mark that is in a different module, the module switching is automatic.

1.2 The File menu



We will now describe most of the items on the File menu.

The Quit command lets you leave **HiSoft C** after asking whether you wish to save any unsaved work. If you realise that you don't want to quit you can cancel this command.

Abandon clears the program in memory and leaves you with a blank screen. If you have modified the program without saving it you will be warned.

Note that, with Abandon, only the current text is deleted. The other seven modules are left alone. This option, as well as the Load file, Insert file, Save File, and Save as commands, only affects the current module.

The Load file command loads a file into memory, surprisingly enough.

The file selector appears and you can choose the file which you wish to load. If there is a file already loaded in this module that you haven't saved then **HiSoft C** will ask for confirmation because the new program overwrites the old one in memory.

The name you specify is displayed in the work window's title bar. This is the name of the current file.

Save file stores the program in memory on disk under the current name. If a name hasn't been given yet (the title bar shows `No name`) then the file selector will appear for you to enter the file name and possibly change the directory (folder) or drive.

Initially the file selector is set up in directory `A:\EXAMPLES`. This is so that you can load the examples off Disk 2 straightaway. If you want to save a file to a newly formatted disk, you will need to change the directory to `A:\*.*`. Otherwise you will try to save your program in a non-existent directory.

If, when you save a file, a file with the same name already exists, **HiSoft C** will ask you whether you wish to overwrite it. If you don't want to, click on No and the save operation will be cancelled and the existing file will be left as it was.

But if you do want to save the program with the same name, the old version of the program will survive with a .BAK extension rather than .C.

Every time you save a program to disk, **HiSoft C** remembers the name of the file and the position of the cursor within the text. Thus the next time you load **HiSoft C** it can re-load the last program saved and re-position the cursor at the place that it was when you saved the program. You can then continue working where you left off.

The Save as... command lets you save the current program under a different name to that of the file that was loaded. The file selector will appear so you can specify the new name. The program is then saved under that new name.

Finally Insert File lets you merge a program from disk with the program in memory.

You can insert a whole file in to the text in memory. The insertion takes place before the line after the current cursor position.

When you select this option a file selector appears and you can specify the file to insert. You can only insert **HiSoft C** coded files rather than ASCII ones. If you need to insert an ASCII file you can convert it first to the **HiSoft C** format. See **Section 1.9**.

For the same reason files may not be inserted when in Text Mode.

HiSoft C will let you suppress some of its dialog boxes if you wish.

For example, you can suppress the alert box that appears when you quit with an unsaved program. You can also control whether a backup is made, and even whether **HiSoft C** automatically loads the last program saved. These commands are described in **Section 1.8**.

1.3 Working with several files

HiSoft C lets you work simultaneously with up to eight modules that contain eight different programs.

However there is only ever one window open at once. The different programs appear one after another in the same window.

1.3.1 Selecting a module

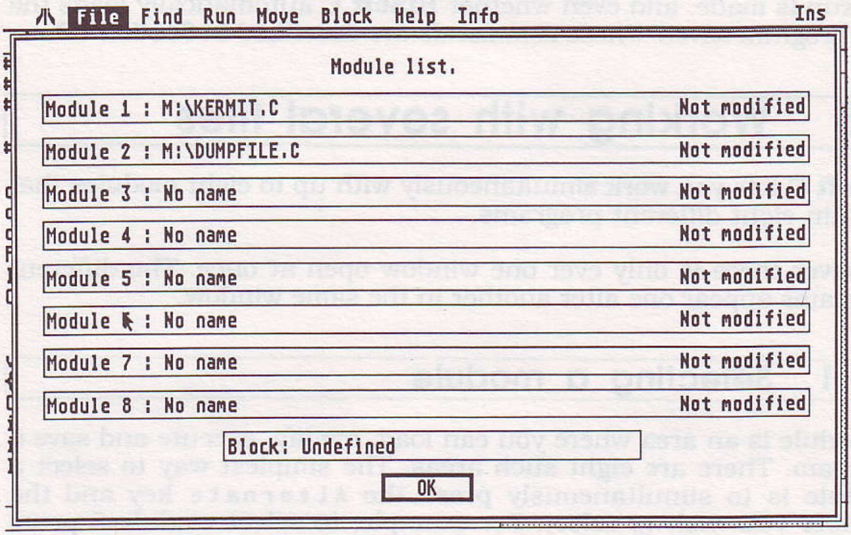
A module is an area where you can load, modify, execute and save a program. There are eight such areas. The simplest way to select a module is to simultaneously press the `Alternate` key and the number you wish to select. For example, to select module 5 press `Alt-5`.

The module's name and the name of the file being edited are shown in the window's title bar. But if you have selected module 5 for the first time there won't be any file loaded. The window title will be `module 5 : No name`

You can move to the next module by clicking on the "Full window" box on the top right of the current window. So, if you are editing module 3 and you click on the full box you will select module 4.

1.3.2 The Module List

The option Module List on the File menu displays a dialog box containing a list of the modules.



```
Module list.

Module 1 : M:\KERMIT.C        Not modified
Module 2 : M:\DUMPFILE.C      Not modified
Module 3 : No name            Not modified
Module 4 : No name            Not modified
Module 5 : No name            Not modified
Module 6 : No name            Not modified
Module 7 : No name            Not modified
Module 8 : No name            Not modified

Block: Undefined

OK
```

For each module the name of the file being edited and the state of the file are shown. The state of the file is either `Modified` or `Not modified`.

This command also lets you select a module easily.

For example, say you wish to edit the file `DUMPFILE`. Thanks to this dialog box, you can see that this program is in module 2. Click on the box corresponding to this module and module 2 and its file `DUMPFILE` will appear on the screen.

If you wish to exit this dialog box without changing modules click on OK.

This dialog box also has another purpose. It tells you if a block has been marked in another module. If one has it enables you to transfer it to another module. See **Section 1.6.3** for how to cut and paste between modules.

1.4 Running a program

1.4.1 How's it done?

You have typed in a fine program or perhaps you have loaded an example. The only thing left is to run it.

To do this, save your program, leave the editor and load the first pass of the compiler with a stack size of 4096 bytes. If there are no errors, load the second pass and specify that you want a GST output file.

There will be errors. And then you will need to include the file stdio.h which is on disk 4. Load the editor again to fix this. And then you link with the wrong libraries so it still doesn't work...

No don't worry it's all lies. That's how it used to be.

To run your program click on Run program from the Run menu. That's it.

Press a key or click on the mouse when the program has finished and you'll be back in the editor.

But we recommend that all programs should be saved before running them. An array index that is a little too big or recursion that uses 20k memory can both crash the machine and you will have to re-load the interpreter.

If the computer crashes instead of displaying bombs (you've coming across bombs no doubt) and returning to the Desktop, **HiSoft C** displays a description of the error and positions the cursor at the program statement that crashed **HiSoft C**.

This considerably reduces the problems caused by program errors but doesn't remove them completely; after such a crash, GEM may become confused.

When you start a program running, the work window and menu bar disappear and are replaced with a blank screen. Then you are free to open your own windows and display your own menu. See **Section 3** for details of functions which let you do this.

You can also deliberately stop a program that is running by pressing two of the `Shift`, `Control` or `Alternate` keys simultaneously. For example, press both `Shift` keys or `Control` and `Shift` together.

An alert box will appear and you are asked to confirm that you wish to halt execution. The work window and editor window will be re-displayed on the screen and the cursor positioned at the point where the program was interrupted.

The function `stop()` can be used within programs to stop the program.

Please note however that it is not possible to interrupt a program whilst a built-in **HiSoft C** routine is running. That is to say when a library function is executing. This is because the library functions are written in assembler rather than interpreted by **HiSoft C** and so the interpreter does not have control. Thus the program may only be interrupted after the function returns. This normally isn't a problem as most functions execute very quickly, but the `event` function may run for several seconds and during that time the program can't be stopped.

1.4.2 Error Messages

If you run a program you have written yourself the chances are that there are either syntax errors or semantic errors.

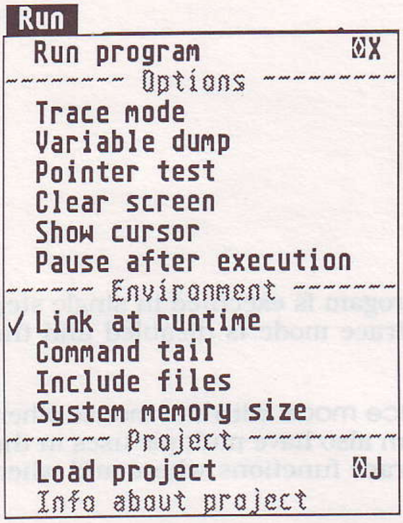
If there is an error whilst running a program, **HiSoft C** displays an error number and message in an alert box. See **Appendix C** for a detailed description of the possible errors.

As soon as you acknowledge the alert box describing your error, the cursor is positioned at the place where the error was found, so you can correct it straight away.

Once your program is displayed you won't be able to see the error message any more. To re-display it click on the Last error item on the Info menu if this isn't disabled and this will display the last error message detected by **HiSoft C**.

We will now describe the other items on the Run menu.

1.4.3 Trace Mode



If you have an error in your program that you can't identify, the Trace Mode will let you visualise the execution of your program. It's a single step mode whereby for each instruction that is executed **HiSoft C** displays the program line that is executing on the screen. You can therefore follow the execution of your program.

To single step a program, all you need to do is to select Trace mode from the menu; a tick mark will be displayed to show that it has been selected. Now, each time you execute a program, it will be in trace mode.

To override this, click on Trace Mode again. The tick will disappear.

Before executing each instruction, **HiSoft C** displays the corresponding program line on the top line of the screen. You must now press a key. Depending on which key you press the program will continue, with or without single step or will be terminated.

If you press `Return`, `Q` or `Control-C` execution continues but not in Trace Mode.

If you press another single key, execution continues in trace mode; the program line is deleted and the old screen re-drawn. Then the instruction is executed and so on.

But if you press two of the `Shift`, `Alternate` and `Control` keys at the same time, execution is interrupted. The instruction that is displayed is *not* executed.

This method of using trace mode is not of much use in some cases. For example if you are interested in the behaviour of the program around line 944 you want to go straight to the place that is causing the problem.

For this reason, there is another way to access single step mode. Two functions enable and disable this mode, `trace_on` and `trace_off`.

For example, in the following program trace mode is activated in the middle of the program and disabled towards the end.

```
main()
{
printf("911");
printf("924");
printf("928");
trace_on();
printf("930");
printf("944");
trace_off();
printf("959");
}
```

After the `trace_on` instruction the progam is executed in single step mode. Then, after the `trace_off`, trace mode is disabled and the program runs normally.

Note that you don't have to select Trace mode with the mouse when using these two functions, and you can also have multiple uses in the same program, so you can use the trace functions where and when you want them.

1.4.4 Following variables

If you don't already know the C language, skip these sections on variables and read them later.

In conjunction with trace mode you can follow the values of one or more variables. You can see the value of one or more variables after each instruction if you wish.

This facility must be used with Trace Mode.

After the display and execution of the current instruction, a dialog box appears which lets you display the values of all the variables you want. Close this box when you have finished and then the next instruction will be executed. See **Section 1.4.6** for a description of this dialog box.

To enable variable dump mode just select the Variable Dump option on the Run menu. As with trace mode there are two functions `var_on()` and `var_off()` which let you enter and exit this mode whilst a progam is running.

In Trace Mode, if you press V (for variable) when about to execute an instruction you will be placed in Variable Dump mode. Powerful, isn't it? If you press M then this will activate the Memory dump (see **Section 1.4.7**) and pressing S will activate the Stack display (see **Section 1.4.8**).

1.4.5 Pointer Tests

You may know that assigning a bad value to a pointer may have catastrophic consequences.

Fortunately, **HiSoft C** tests the values of pointers every time you try to write to an address pointed to by a pointer. If the value of the pointer is junk then an error message (error 33) appears on the screen.

These Pointer Tests are active by default. But if you use pointers a lot in a program that has been debugged you can suppress this option. You'll gain a few tenths of seconds. To do this, click on the Pointer Test item of the Run menu and the tick will disappear.

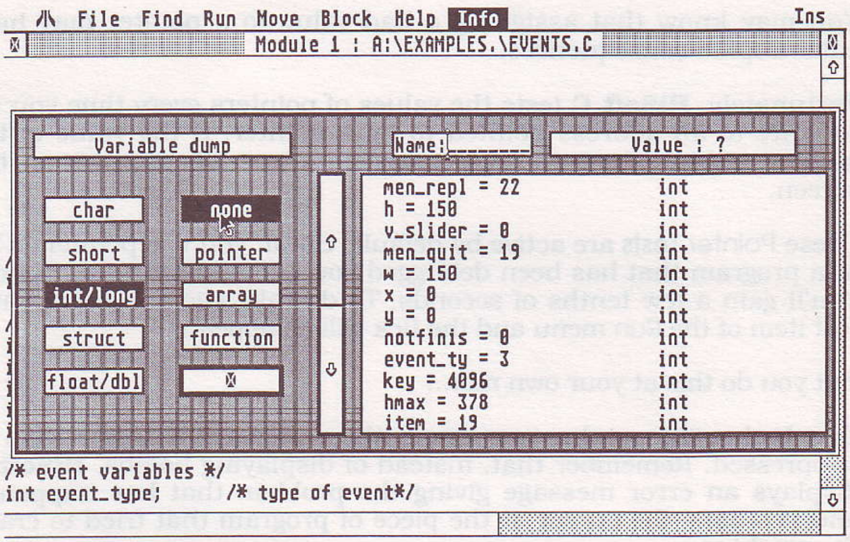
But you do this at your own risk...

One bad pointer value can crash the computer if this option is suppressed. Remember that, instead of displaying bombs, **HiSoft C** displays an error message giving the problem that has happened and positions the cursor at the piece of program that tried to crash the machine.

1.4.6 Variable Dump

The Info menu has five options. We already know about one of them, program Info. We will now cover the Variables option.

This command allows you to display the names and values of variables of a given type.



```
men_repl = 22          int
h = 150                int
v_slider = 0           int
men_quit = 19          int
w = 150                int
x = 0                  int
y = 0                  int
notfinis = 0           int
event_ty = 3           int
key = 4096             int
hmax = 378             int
item = 19              int
```

```
/* event variables */
int event_type;     /* type of event*/
```

When you click on this menu entry a dialog box like that above appears. This is the same as as the box that appears when Variable dump mode is selected. The dialog box is split into two parts, the right hand side is a window where the names and values of variables are displayed.

On the left are ten selectable buttons. One of these lets you leave the box and the other nine let you choose which types of variables you wish to display.

A variable's type is composed of an elemtentary type (char, short, int/long, float/double or struct) and a class (none, pointer, array or function). For example, an ordinary integer variable is a combination of int/long and none. An array of floating point numbers is float/dbl and array. A pointer to a structure: struct and pointer. A function returning a character: char and function.

All types of variables can be split up like this. To display all the variables of a given type, you must select a type and a class by clicking on the appropriate buttons with the mouse.

For example, to display all the pointers, select pointer and all the primary types (char, short, int/long, float/dbl, struct). In this way you will display all the pointers used by the program.

If you click a second time on a button, this button will be de-selected. You can thus successively examine several types of variable.

Finally a little trick we haven't mentioned before. The process above is a bit involved if all you want to do is display one variable. This is what the two small boxes near the top right of the dialog box are for Type the name of a variable in one and then click on the name you have just entered with the mouse and the value of the named variable is displayed in the second box.

The values are displayed in a format that depends on the class of the variable. If it's a simple variable (none) then the value is displayed in decimal. If it's a pointer the hexadecimal address to which it points is displayed. For arrays, the address of the first element is shown and for functions the address of the start of the code.

You may display several types of variables simultaneously.

If there are too many variables to display them all in the window use the up and down buttons to scroll through them.

HiSoft C displays after each value the type of the value in a simplified fashion. The base type is displayed first (`int`, `char`, `double`, `short`...) followed by an indication of the class; if the variable is a pointer a `*` is displayed, an array `[]` is added, or for a function `()` is added. For simple variables nothing is added.

For STATIC variables, the module in which they are declared is also given. For example for a static variable declared in Module 1, **HiSoft C** adds an extra `mod:1`.

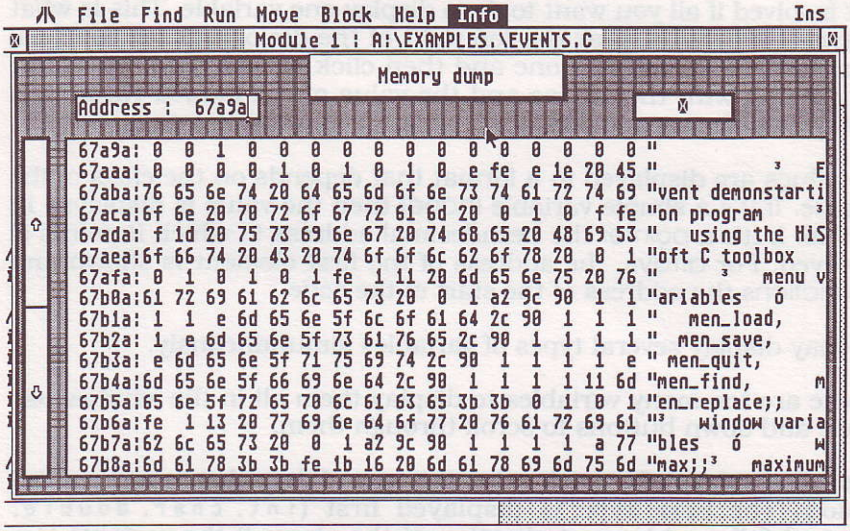
This command can not be selected until the program has been run, nor can the variable dump be consulted if the program is changed. Finally, only global and static variables may be displayed - local variables are invisible.

If memory is very short, **HiSoft C** can not retain the symbol table after execution and so this option is not available.

1.4.7 Memory dump

This command lets you look at part of the memory of your computer. You give it an address, and the memory area concerned is displayed on the screen.

To do this, select Memory dump from the Info menu. A dialog box will appear that nearly fills the screen. An editable field marked `Address` lets you enter the hexadecimal address of the memory area that you wish to examine.


File Find Run Move Block Help Info Ins
Module 1 : A:\EXAMPLES.\EVENTS.C
Memory dump

```
Address :  67a9a

67a9a: 0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0 "
67aaa: 0  0  0  0  1  0  1  0  1  0  e fe  e 1e 20 45 "          ³    E
67aba:76 65 6e 74 20 64 65 6d 6f 6e 73 74 61 72 74 69 "vent demonstarti
67aca:6f 6e 20 70 72 6f 67 72 61 6d 20  0  1  0  f fe "on program      ³
67ada: f 1d 20 75 73 69 6e 67 20 74 68 65 20 48 69 53 "   using the HiS
67aea:6f 66 74 20 43 20 74 6f 6f 6c 62 6f 78 20  0  1 "oft C toolbox
67afa: 0  1  0  1  0  1 fe  1 11 20 6d 65 6e 75 20 76 "      ³   menu v
67b0a:61 72 69 61 62 6c 65 73 20  0  1 a2 9c 90  1  1 "ariables   ó
67b1a: 1  1  e 6d 65 6e 5f 6c 6f 61 64 2c 90  1  1  1 "   men_load,
67b2a: 1  e 6d 65 6e 5f 73 61 76 65 2c 90  1  1  1  1 "  men_save,
67b3a: e 6d 65 6e 5f 71 75 69 74 2c 90  1  1  1  1  e " men_quit,
67b4a:6d 65 6e 5f 66 69 6e 64 2c 90  1  1  1  1 11 6d "men_find,      m
67b5a:65 6e 5f 72 65 70 6c 61 63 65 3b 3b  0  1  0  1 "en_replace;;
67b6a:fe  1 13 20 77 69 6e 64 6f 77 20 76 61 72 69 61 "³   window varia
67b7a:62 6c 65 73 20  0  1 a2 9c 90  1  1  1  1  a 77 "bles   ó        w
67b8a:6d 61 78 3b 3b fe 1b 16 20 6d 61 78 69 6d 75 6d "max;;³   maximum
```

Press `Return` and a display of 256 bytes starting at the address you entered appears. You can click on the two arrows at the side of the display window and these move 256 bytes forwards and backwards. Each byte is displayed in both hexadecimal and ASCII.

This command is most frequently used to examine the contents of structures and arrays having read their address using the Variable dump command. It can be accessed in trace mode by pressing M.

The memory dump can also be activated and de-activated from within your program using the mem_on() and mem_off() functions. These work in the same way as the var_on() and var_off() functions described in **Section 1.4.4**.

1.4.8 Stack display

This command is similar to the previous command except that it lets you examine the area of memory used by the interpreter's stack. It is here that the intrepreter stores your local variables in the order in which they were declared and in order of function calls.

Thus this may be used to check the values of local variables. The local variables of the current function start 8 bytes from the top of the stack. This is the address of the local variable that is declared first, with the second variable next, etc.

The stack display can be accessed in trace mode by pressing S. It can be activated and de-activated from within your program using the stack_on() and stack_off() functions. These work in the same way as the var_on() and var_off() functions described in **Section 1.4.4**.

1.4.9 Link at runtime

When executing your program **HiSoft C** can consider the modules in two ways.

When the Link at runtime option is not selected, it considers the eight modules to be eight different programs which have nothing to do with each other. So when you run the program only the curent module is executed.

On the other hand when Link at runtime is selected, **HiSoft C** considers that all the modules form one single progam. This lets you use the classic C programming style of using several modules. **HiSoft C** executes all the modules together and 'links' between them in the same way as a linker in a separate compilation system.

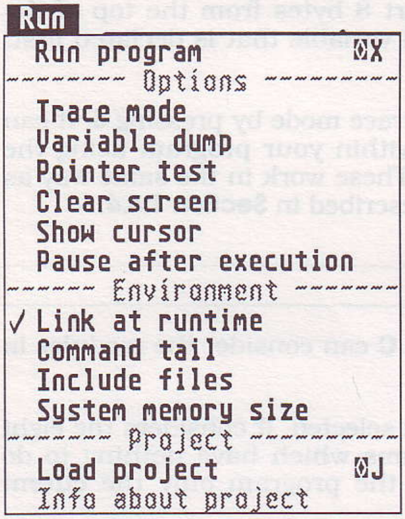
So you can declare a function or variable in one module and use it in another module as if you were using a compiler and linker.

If you have selected Link at runtime, **HiSoft C** effectively links the modules before running them.

1.4.10 Execution Environment

There are three options on the Run menu that affect the execution of a program.

They are selected or de-selected with the mouse. If an option is currently selected there is a tick in front of it.



Clear screen lets you view the execution of your program on a blank white screen rather than a GEM-type screen.

Show cursor starts execution of the program in an environment similar to a `.TOS` program, with a white screen and a solid text cursor.

Pause after execution waits for a key press or a mouse click when the program finishes execution before returning to the editor.

These three options are saved and restored every time you save or load a program with the File menu commands.

In effect, they are dependent on the program. The TOS environment suits some programs but not others. The automatic saving of these options lets the environment change with each program.

1.4.11 Command tail

The Command tail option from the Run menu lets you enter a list of arguments from the keyboard. These are passed to your program when it executes.

This lets you simulate passing parameters to the program with the interpreter in a similar way to those passed to compiled .TTP programs.

The parameters are separated by spaces except when enclosed in double quotes. For example, the following list,

```
        list of "three parameters"
```

consists of three parameters: "list", "of" and "three parameters".

You access the program parameters in the classic C way, that is to say, by having two parameters to the function `main`.

The first (`argc`) is the number of arguments in the command line. The second (`argv`) is a pointer to an array of strings, which contain the parameters, one parameter in each string.

The following program writes the parameters passed to it on the screen.

```
main(argc,argv)
int argc;
char **argv;
{
        while(argc--)
                puts(argv[argc]);
}
```

1.4.12 Include files

The Include files option on the Run menu displays a list of the current resident include files in a dialog box.

When a `#include` instruction is encountered during program execution the corresponding file is loaded into memory. The include files should be stored in a directory called \HEADER on the disk from which **HiSoft C** was loaded, and you should enclose the file name in angle brackets: < and >. If you use quote symbols (") then the interpreter will look in the directory: this will normally be that from which **HiSoft C** was loaded.

The include files remain in memory between executions to avoid loading them repeatedly. This option displays a list of include files that have been loaded by previous runs of programs.

You may have up to eight include files in memory simultaneously. The dialog box can therefore show eight file names.

It is possible to deliberately remove include files from memory by clicking the `Erase resident files` in the dialog box. These files are not deleted from the disk but only from memory. They will automatically be re-loaded if you run a program that needs them. The include files are automatically removed each time you load a program to ensure consistency.

⚠ Do not use the include files from a C compiler with **HiSoft C**. This will cause errors. **HiSoft C** has all the standard include files and only these should be used with **HiSoft C**.

1.4.13 System Memory Size

The system memory size is the amount of free memory that is not allocated to **HiSoft C** but is free for use by the Atari ST's operating system.

This memory is used for memory allocation (the `malloc` function and friends), loading resource files (`rsrc_load`), the file selector and loading compiled procedures.

By default this is fixed at 8K bytes. This is perhaps a little small for many purposes.

If you wish to load a large resource file, or use the memory allocation functions extensively or load many compiled functions then this 8k will be too small.

On the other hand, if you are using none of the facilities above you may be interested in reducing this space thus increasing the memory that **HiSoft C** can use.

Increasing the system memory size has the effect of reducing the space reserved by the interpreter for the storing and running of your programs.

So how do we modify this system memory size?

Select the System memory size item from the Run menu. Enter the number of kilobytes you wish to allocate to system memory and click on `OK`. Save the **HiSoft C** configuration with Save Options from the File menu. Quit **HiSoft C** and re-load it.

The system memory size is now the size you specified. It will be this value each time you load **HiSoft C** as this has been saved on the disk. To change to a new value please repeat the sequence described above.

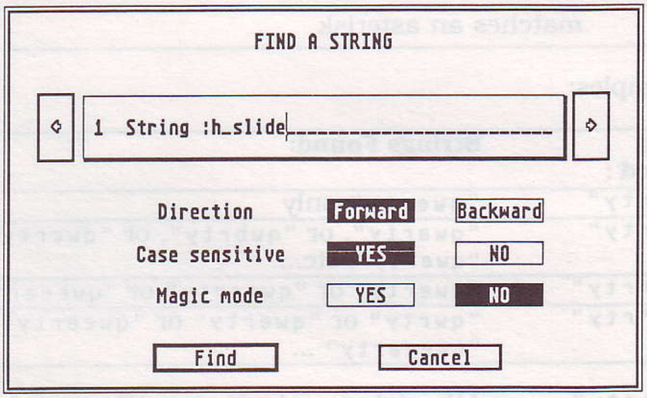
HiSoft C uses this space itself. Always leave at least 6K bytes for the system; it is impossible to reduce this any further.

1.5 Find and replace

1.5.1 Finding

Thanks to the Find... item of the Find menu you can search for a string of characters within a loaded program starting at the cursor position.

When you select this option a dialog box appears on the screen and you can type in the string to search for, thus,



FIND A STRING

1 String :h_slide

Direction [✓ Forward] Backward

Case sensitive [✓ YES] NO

Magic mode YES [✓ NO]

The string that you type in may correspond exactly to the item that you are looking for. For example you could search for `the end` and the editor would search for those exact characters in the text. In this case you should leave Magic Mode set to NO.

But you may have more exotic requirements containing a number of unknowns. For example, you might wish to find all the strings of characters starting with `the end` and finishing with `zz`.

Equally you can search for every time a `*` is separated from a `+` by zero or more blanks. It's not blindingly obvious that this is useful but there are many practical applications of this sort of thing. This is what the Magic Mode is for.

The string that you type will contain ordinary characters which match exactly the same character in the text, but there are also special characters as follows:

`?`	matches any single character
`x+`	match with a sequence of one or more of character `x`. i.e `x`, `xx`, `xxx` or more character `x`'s
`x*`	match with a sequence of zero or more of character `x`. That is nothing, `x`, `xx`, `xxx` etc
`?*`	matches any sequence of characters. It could be no characters at all or a string of characters
`?+`	matches any string of at least one character
`\?`	matches a question mark
`\+`	matches a plus sign
`\*`	matches an asterisk

Some examples:

String Entered :	**Strings Found:**
`"qwerty"`	`"qwerty"` only
`"qw?rty"`	`"qwarty"`, or `"qwbrty"`, or `"qwcrty"`, or `"qwerty"`, etc...
`"qwe+rty"`	`"qwerty"` or `"qweerty"` or `"qweeerty"` ...
`"qwe*rty"`	`"qwrty"` or `"qwerty"` or `"qweerty"` or `"qweeerty"` ...
`"qw?+rty"`	All strings starting with `"qw"`, and finishing with `"rty"`, with any string with at least one character in between
`"qw?*rty"`	All strings starting with `"qw"`, and finishing with `"rty"`, with any string including `"qwerty"`
`"qw\+rty"`	`"qw+rty"` only
`"qw\*rty"`	`"qw*rty"` only
`"qw\?rty"`	`"qw?rty"` only

The string that you are searching for must all be on one line of text. When the characters are found the cursor is automatically moved to that place.

The direction of search, `Forward` or `Backward` may be specified by clicking on the appropriate button.

If you wish the search to distinguish between upper and lower case letters click on `YES` in the `Case Sensitive` field. If you want upper and lower case to be considered the same click on `NO`. If you click on `NO`, the string `qwERT` is considered to the same as `QWERT` and `qwert` during the search.

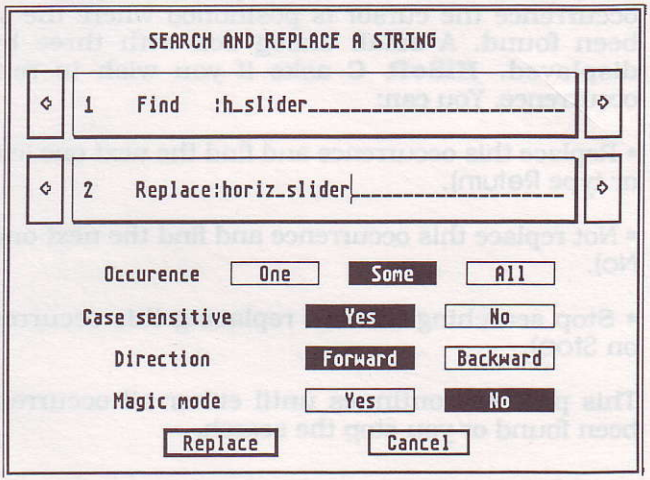
You have eight possible search strings at your disposal. Thus you can build up a little library of search strings that you commonly search for, choosing with the Find command which one you want this time.

Click on the arrow button to select the find string you want; the current number is displayed.

When you have found a string you can use the Repeat Find command to find further occurrences with the same conditions.

1.5.2 Find and replace

You can search for a sequence of characters and replace it with another. To do this choose the Find & replace item from the Find menu.


SEARCH AND REPLACE A STRING

1 Find :h_slider

2 Replace:horiz_slider

Occurence: One [✓ Some] All

Case sensitive: [✓ Yes] No

Direction: [✓ Forward] Backward

Magic mode: Yes [✓ No]

The dialog box has two lines so you can enter two strings. The first is the string to search for and the second is the string that will replace it. You can switch between the two fields using the up and down cursor keys.

You can use all the same facilities that are available with the ordinary Find... command.

You can use wildcards, access 8 find strings, select the search direction and whether the search is to be case sensitive.

There are three modes for using replace. You choose which one by clicking on one of three buttons. You can choose to replace one, several or all occurrences of a string.

• Replace One occurrence

    This replaces only one occurrence. The first one found is replaced.

• Replace All occurrences

    All occurrences found are replaced. This starts from the cursor and in the direction specified.

    During the operation of this command, if you press one of the Shift, Control or Alternate keys the replace operation continues but in "Some Occurrences" mode. See below.

• Replace Some Occurrences

    This is the most commonly used option. For each occurrence the cursor is positioned where the string has been found. A small dialog box with three buttons is displayed. **HiSoft C** asks if you wish to replace this occurrence. You can:

    • Replace this occurrence and find the next one (click on Yes or type Return).

    • Not replace this occurrence and find the next one (click on No).

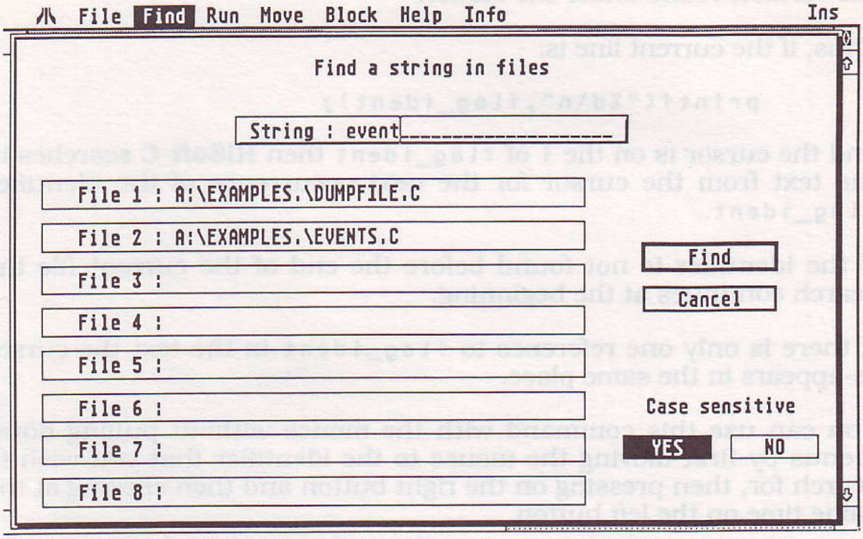
    • Stop searching without replacing this occurrence (click on Stop).

    This process continues until either all occurrences have been found or you stop the search.

Be careful not to do a replace operation that will leave a comment line or a string without its terminator. An error message will appear and the search will stop.

1.5.3 Searching in files

The option Find string in files from the Find menu lets you search for a string within files on disk. A dialog box appears:



Find a string in files

String : event

File 1 : A:\EXAMPLES.\DUMPFILE.C

File 2 : A:\EXAMPLES.\EVENTS.C

File 3 :

File 4 :

File 5 :

File 6 :

File 7 :

File 8 :

The files may be standard ASCII or **HiSoft C** coded files.

First define the files in which the search is going to take place. To do this click on one of the 8 boxes for file names that you can see in the dialog box. You can search up to eight files at once.

The file selector will appear; select a file and it will appear in the box that you clicked on.

To remove one of the eight file names, click on the file name and then select Cancel in the File Selector and the name will disappear.

To start the search click on Find. When the first occurrence is found, **HiSoft C** will display a message indicating in which file, and on what line within the file, the string has been found.

You can stop the search by selecting Cancel, continue the search in the same file (Continue) or continue searching in the next file (Next file).

1.5.4 Find Identifier

This option lets you find an identifier without having to type it. When you use this command **HiSoft C** will search for the next occurrence of the variable name under the cursor.

Thus, if the current line is:

```
        printf("%d\n",flag_ident);
```

and the cursor is on the `i` of `flag_ident` then **HiSoft C** searches in the text from the cursor for the next occurrence of the identifier `flag_ident`.

If the identifier is not found before the end of the current file the search continues at the beginning.

If there is only one reference to `flag_ident` in the text the cursor re-appears in the same place.

You can use this command with the mouse without pulling down menus by first moving the mouse to the identifier that you wish to search for, then pressing on the right button and then clicking at the same time on the left button.

This way you can search for an identifier extremely quickly.

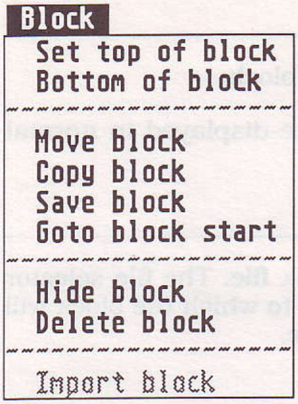
1.6 Block operations

You can define a block of text and carry out various operations on it. For example, delete the text, copy it to another place or save it to disk.

1.6.1 Defining a block

A block is a collection of consecutive whole lines in the text. There are two ways of defining such a block, with the mouse or with the menu.

With the mouse, double click in the first and last line of the block that you want to define.



With the menu, move the cursor to the first line of the block and select the menu item Set top of block from the Block menu. Then move the cursor to the last line of the block and click on the Bottom of block item.

When you have defined a block it is shown highlighted in inverse video (white characters on a black background).

The block stays selected until you choose Hide block or Delete block.

If a block is selected when you change a module the block isn't de-selected but it is treated specially. See **Section 1.6.3** .

You must select a block before using any of the commands described below.

1.6.2 Block operations

There are plenty of choices of operations to perform on blocks:

Move block

This command moves the block to the current cursor position. The block is deleted from its original position and inserted *after* the line containing the cursor.

Note that a block may not be moved if the cursor is currently inside the block.

Copy block

This command copies the block to the cursor position. The contents of the block are duplicated on the line *after* the current cursor position.

Delete block

This command is used to remove a block of lines.

The block is destroyed.

Hide block

All this command does is de-select the current block.

The text that was shown in reverse video is re-displayed in normal video; the text is not changed.

Save Block

The contents of the block are written to a disk file. The file selector appears and you can enter the name of the file to which the block will be written. The text in memory does not change.

Goto block start

This command isn't the most powerful in the editor but it can be useful on occasion.

1.6.3 Copying a block to a another file

HiSoft C lets you select a block in one module and insert its contents in another module.

If you select a block and then change module the position of the block isn't lost; the editor remembers which module the block is in and which lines it contains.

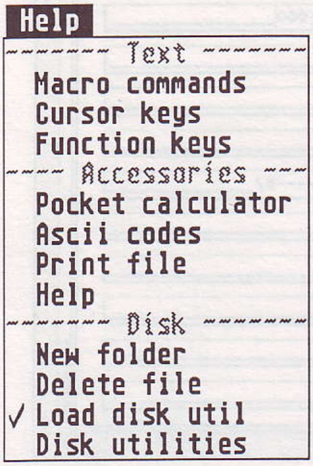
This can be seen using the Module List command from the File menu.

You can insert the block after the current line by selecting Import Block from the Block menu.

This is the only operation you can carry out on the block when you are in another module, apart from re-defining it of course.

1.7 The Help menu

This section describes the various commands under the Help menu.

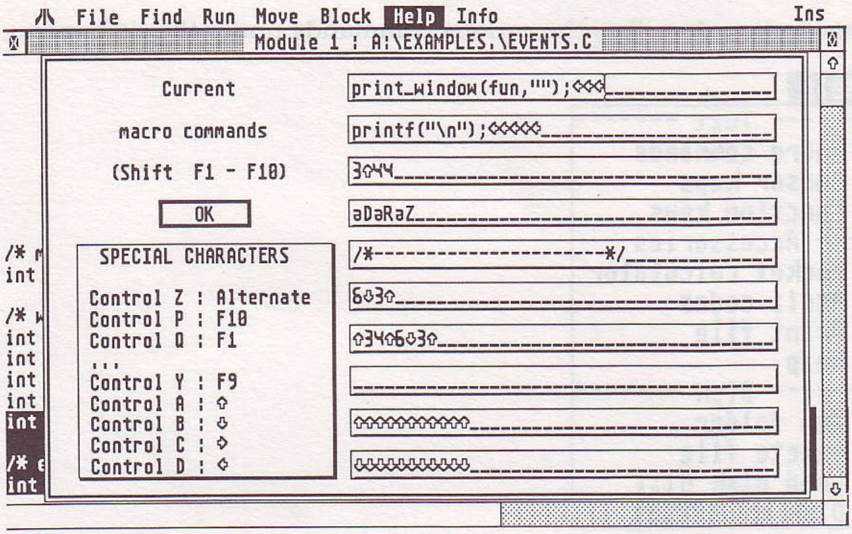


1.7.1 The macro commands

You can set up ten programmable keys `Shift F1` to `Shift F10`. (Press a `Shift` key and the function key together).

When you press one of these keys a sequence of commands is executed as if you had hit the equivalent keys.

When you select the Macro Commands menu item, a dialog box like that below appears:


File Find Run Move Block Help Info
Ins
Module 1 : A:\EXAMPLES.\EVENTS.C
OK

```
Current                  print_window(fun,"");⇦⇦⇦
macro commands           printf("\n");⇦⇦⇦⇦⇦
(Shift  F1 - F10)        3⇧44
                         aDaRaZ
SPECIAL CHARACTERS       /*--------------------*/
                         6⇩3⇧
Control Z : Alternate    ⇧34⇧6⇩3⇧
Control P : F10
Control Q : F1           ⇧⇧⇧⇧⇧⇧⇧⇧⇧⇧⇧
...                      ⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩
Control Y : F9
Control A : ⇧
Control B : ⇩
Control C : ⇨
Control D : ⇦
```

There are ten lines so you can enter ten sets of commands. Each line corresponds to one key.

There are two sorts of characters that can be entered in the command strings: control characters, which are interpreted as commands to execute, and normal alphanumerics and punctuation.

When the macro command is executed, normal characters simply appear on the screen as if they had been typed.

Now for the control keys:

The keys `Control-P` to `Control-Y` correspond to `F10`, `F1` to `F9` respectively. For example if you select the top line of the dialog box (the first macro) and type `Escape` to clear it and then hit `Control-S`, a stylised 3 digit will appear indicating `F3`. Then pressing `Shift F1` is equivalent to pressing `F3`.

if you type `Control-P`, `Control-Q`, `Control-R`, `Control-S`, `Control-T`, `Control-U`, in sequence the command will become equivalent to typing `F10`, `F1`, `F2`, `F3`, `F4`, `F5`.

Be careful not to confuse the non-shifted function keys and the shifted function keys that correspond to macro keys. The unshifted ones are pre-defined and can not be changed. Also the ten macro commands can not call each other although they may call the unshifted function keys as we have seen above.

`Control-Z` followed by a letter is equivalent to pressing `Alternate` and the letter at the same time. This can be used to access some of the menu functions from macro commands.

For example, `Alternate-T` selects the Top of file command. If you you enter `Control-Z` and then `T` in a macro command this will go to the beginning of the file.

Also, `Control-Z` followed by one of the digits `1` to `8` will select the corresponding module.

The only exception to the above is that the `Quit` command can not be incorporated into a macro command.

Cursor movements may be added to macro commands. They are represented by `Control-A` to `Control-D` and are echoed with arrows pointing in the direction of the cursor movement.

In addition the following control codes can be entered:

- `Control-M` is equivalent to the `Return` key.
- `Control-H` is equivalent to the `Backspace` key.

Use of control characters other than those above and attempts to call menu items that do not exist (via a `Control-Z` followed by a letter) will give an error message.

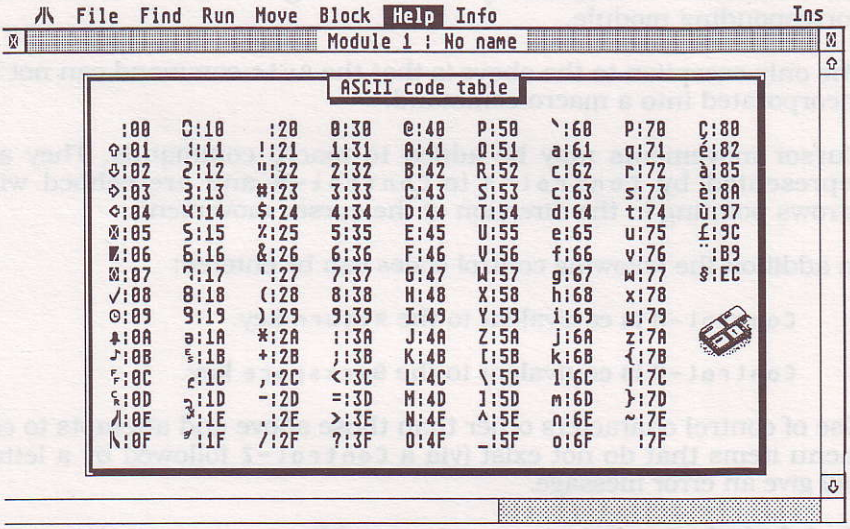
By default the macro keys are set up as follows:

`Shift F1`	Add a call to the function `print_window` to the file.
`Shift F2`	Add a call to the function `printf` to the text.
`Shift F3`	Duplicates the current line.
`Shift F4`	Positions the cursor at the start of the file and calls the Find command.
`Shift F5`	Adds a comment header line.
`Shift F6`	Deletes the new line at the end of this line, thus joining the following line to it.

`Shift F7`	Deletes the new line at the start of this line, thus joining it to the previous line.
`Shift F8`	Spare.
`Shift F9`	Moves the cursor half a screen towards the start of text.
`Shift F10`	Moves the cursor half a screen towards the end of text.

1.7.2 ASCII code table

When you select the ASCII code command, **HiSoft C** displays a dialog box which shows the character codes used by the ST.



```
  :00   ⌧:10    :20   0:30   @:40   P:50   `:60   P:70   Ç:80
 ⇧:01   ¦:11   !:21   1:31   A:41   Q:51   a:61   q:71   é:82
 ⇩:02   2:12   ":22   2:32   B:42   R:52   c:62   r:72   à:85
 ⇨:03   3:13   #:23   3:33   C:43   S:53   c:63   s:73   è:8A
 ⇦:04   4:14   $:24   4:34   D:44   T:54   d:64   t:74   ù:97
 ⌧:05   5:15   %:25   5:35   E:45   U:55   e:65   u:75   £:9C
 ◩:06   6:16   &:26   6:36   F:46   V:56   f:66   v:76   ¨:B9
 ⌧:07   7:17   ':27   7:37   G:47   W:57   g:67   w:77   §:EC
 ✓:08   8:18   (:28   8:38   H:48   X:58   h:68   x:78
 ◷:09   9:19   ):29   9:39   I:49   Y:59   i:69   y:79
 ♠:0A   ə:1A   *:2A   ::3A   J:4A   Z:5A   j:6A   z:7A
 ♪:0B   ᴱₛ:1B   +:2B   ;:3B   K:4B   [:5B   k:6B   {:7B
 ᶠF:0C   ꞔ:1C   ,:2C   <:3C   L:4C   \:5C   l:6C   |:7C
 ᶜR:0D   ʓ:1D   -:2D   =:3D   M:4D   ]:5D   m:6D   }:7D
 ⁄|:0E   ᵡ:1E   .:2E   >:3E   N:4E   ^:5E   n:6E   ~:7E
 |\:0F   ᵍ:1F   /:2F   ?:3F   O:4F   _:5F   o:6F    :7F
```

1.7.3 Print file

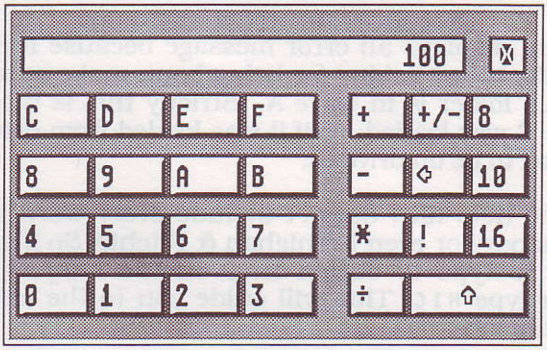
The Print file option lets you print a disk file without monopolising the CPU.

If Print file is selected, **HiSoft C** lets you enter the file to print using the file selector, and the file will be spooled for printing. You keep control of the ST and can, for example, edit or run a program while the file is printed - it's magic!

However, despite the above, you can't use the File menu whilst printing; loading and saving files is not possible, nor is leaving **HiSoft C**. If you select one of these options you will be asked whether you want to cancel printing so that the command you selected can then be executed.

1.7.4 The Pocket Calculator

A calculator can be used in **HiSoft C** by clicking on the Pocket calculator menu entry. It looks like this:



In brief, it uses reverse Polish notation, is integer-only but can calculate in base 8, 10 and 16.

The four standard arithmetic operations are available. To calculate `20E3 - 1FE3` for example, click the digits for `20E3` and the larger up-arrow button (this is the enter key). Then enter `1FE3` in a similar way. Finally press - and the answer appears.

To change base click on the appropriate button on the right hand side of the calculator. The displayed value is immediately converted to the new base.

You can easily see which is the current base because only the appropriate digits are enabled. So when you are in base `8`, the digits `8`, `9`, `A` to `F` are disabled and displayed in grey. In base 10, only `0` to `9` are enabled, and in base 16 you can use all the digits.

The `!` button clears the calculator display.

The left arrow key deletes the last digit entered.

The `+/-` changes the sign of the number. Finally to leave the calculator click on the close box at the top right of the display.

You can paste the value displayed by the calculator into your program by using the `F1` and `F4` keys.

1.7.5 Help

When you select the Help menu item, or press the `Help` key, a small dialog box appears on the screen.

It asks you to enter the word that you would like help about. If you give a valid keyword, help about the appropriate word will be displayed.

If the program displays an error message because it does not know the word that you are asking for help about, make sure that the disk with the `HELP` folder is in drive A. (Strictly this is the current drive when **HiSoft C** was loaded; so if it was loaded from drive C, the `HELP` folder will need to be on drive C).

Obviously the help files doesn't include every word in the Shorter English Dictionary or even Kernighan & Ritchie. So if you don't know which word to type to obtain help on the subject that you are interested in type `HIC`. This will guide you to the information that you are looking for.

The subjects covered by the help files are the library functions, in particular the **HiSoft C** specific ones, the menus, the error messages and a few others.

You can display a list of several library functions with some information about each one. For each function the type returned and the number of parameters is displayed. To do this, enter the name of the function preceded by a dash or minus sign (-). For example, -`objc` will give all the functions starting with `objc`. This facility is available for all 460 library functions.

1.7.6 The Disk Options

You can carry out two disk operations directly from the **HiSoft C** editor.

You can delete a file by selecting Delete file (!) from the Help menu. The file selector will appear and you can enter which file to delete.

You will be asked to confirm that you wish to delete the file before it is destroyed.

You can also create a New folder. Again the file selector is used to enter the name of the new folder that you wish to create.

You can also select Disk utility. This is a program that lets you copy, delete, and rename files and folders, format disks etc. This program is described in **Section 1.11**.

1.8 Editor configuration

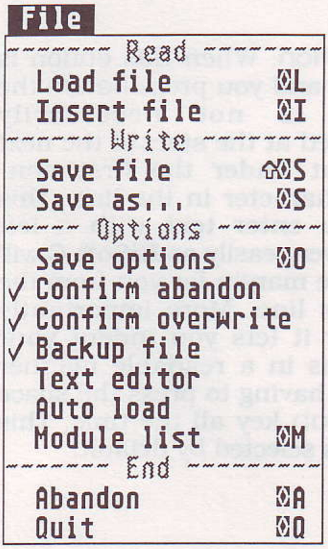
1.8.1 Saving the configuration

The **HiSoft C** editor has many parameters that can be set by the user.

You can save the values of these options on disk so that you do not have to set them up every time you use **HiSoft C**.

The Save options command from the File menu lets you save the editor configuration file to disk. The options are saved in a file which is automatically loaded when you load **HiSoft C**; thus the next time you load the interpreter the configuration will be exactly the same as when you selected Save options. Of course, you must have the disk from which you load **HiSoft C** present when you use this command.

1.8.2 Editor Options



There are ten menu commands that let you configure the editor. They selectively change how the editor works.

For example, if the Confirm abandon item from the File menu is selected then you will be asked to confirm that you wish to abandon a file if you have changed it without saving it to disk. By default this option is selected. But if you wish to suppress this, click on the menu item. The tick mark will disappear and the abandon confirmation box will no longer appear.

There are the following other configuration options on the File menu:

Confirm overwrite. This option informs you if you are about to save on top of a file that already exists. A dialog box appears asking you if you wish to overwrite the old file. This option is selected by default.

Backup file. When you save a program, this option lets you keep the previous version by renaming this file with a `.BAK` extension rather than `.C`. Thus, you always have two versions of the program, the current one and the last one. This is another option that is selected by default.

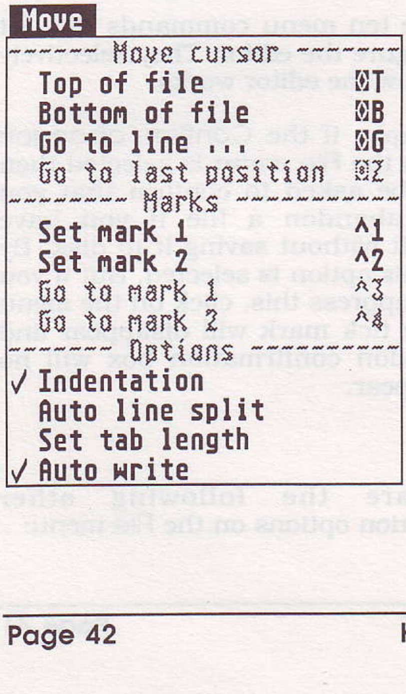
Auto load. If this option is selected, then when **HiSoft C** is loaded it automatically loads the last file that was saved and positions the cursor at its position when the save took place. This option is not selected by default.

Text editor. This option activates a special editor mode that lets you use **HiSoft C** as a mini-word processor. See **Section 1.9** for more details. This option is not set by default.

The Pointer Test option from the `Run` menu is configurable. This enables testing that pointers are valid during program execution. See **Section 1.4.5**. This option is selected by default.

The Link at runtime option is also configurable and is discussed in **Section 1.4.9**. This option is selected by default.

Some cursor control options follow; these are on the Move menu.



Indentation. When this option is selected and you press Return the cursor is not necessarily positioned at the start of the next line, but under the first non-blank character in the line. This lets you enter text with a left margin very easily as **HiSoft C** will enter the margin for you from the previous line. More importantly however it lets you indent your programs in a readable manner without having to press the space bar or Tab key all the time. This option is selected by default.

Auto Wrap. When this option is selected, and you try to type a line that is too long to fit in the work window the editor will automatically enter a new line for you and you can continue to type on the next line. The word that you are currently typing is placed on the next line.

When this option isn't selected and you type a long line the screen scrolls sideways to let you enter lines longer than 80 characters. This option isn't selected by default.

Auto Write. When this option is selected and you type in an if or for statement, the interpreter will insert a matching pair of curly brackets, { and } on separate lines and places you on a line between them: so that you can insert the code for the statements straight away. If you don't like this, switch just it off.

If you do not like one or more of the options above then you can de-select them by clicking on the corresponding menu entry. The tick in front of the entry will disappear and you should then save the configuration using Save options from the File menu.

Set tab length. You can change the size of tabs. By default this is five characters but can be set to any value you like (within reason!). When the editor configuration is saved the size of tabs is also saved.

Macro commands. The current settings of the macro commands are saved as well so that you can set them up in exactly the way you like and they will always be ready when you load **HiSoft C**.

And finally, the last part of the configuration is the path that is used for loading and saving files. The name of the disk and directory that you last used with the file selector is saved when you saved the options.

This option is particularly useful if you have a hard disk as you don't have to select the right folder every time you use **HiSoft C** - you are already there.

1.8.3 Redefining the keyboard

You can modify the keyboard shortcuts that are used for the menu commands.

By default, `Alternate S` is equivalent to selecting Save from the File menu; but you could change this to `Control N` or any other combination you wish.

To do this you will need to know the keyboard scan codes corresponding to the key sequence that you are going to use. For example, for the combination `Control N` the code is 310E.

To find the scan code for a given sequence, run the following program and type the key; the corresponding sequence will then be displayed on the screen.

```
void main()
{
        printf("%x\n",evnt_keybd());
        evnt_keybd();
}
```

Once you know the code put the editor in to Text Mode and load the file `F5.IC` from your backup disk 1.

Each line of this file corresponds to one command. The third line is the line for the Save file command. To change this to `Control N` replace the code for `Alternate S` (1F00) with that for `Control N` (310E).

Now save the `F5.IC` file, quit and reload **HiSoft C**. Type `Control N` and this will be equivalent to saving the file.

Of course this should be performed with your backup copy in case you make a mistake.

1.9 Editor Mode

HiSoft C has an editor mode as opposed to the interpreter mode. The Editor Mode command from the File menu lets you change from one mode to the other.

By default you are in interpreter mode in which **HiSoft C** encodes the lines of your program which you type although you hardly notice this. The program is saved on disk in this non-ASCII format; however this speeds up the execution of the program.

You can remove he encoding thanks to Editor Mode. The memory image is always faithfully what you have typed in and is saved to disk in standard ASCII format. So you can use **HiSoft C** as a normal text editor. Together with the Auto Wrap command you can almost use it like a word processor - you can type away without ever pressing Return.

But in Editor mode, you can run C programs. Clearly, you can't have everything. To run programs you must be in Interpreter mode.

HiSoft C recognises two types of files: its own format and standard ASCII. You are in you can load both type of file in either mode; the editor will convert between them when you load.

In Interpreter mode, you can not load C programs in ASCII that have been written with a compiler in mind; they will be automatically encoded into the internal form. Similarly you can load a program that you have developed with the interpreter when you are in Text mode and then save it out so that you can compile it or even send it to another machine.

1.10 Projects and Compiled Libraries

This section describes one of the more advanced features of **HiSoft C**; beginners are advised to ignore this for now.

1.10.1 What is a project?

As far as **HiSoft C** is concerned a Project is a collection of files that make up a whole program.

All applications can be split into one or more files. The files are of two types:

C source files which are programs that are written in C and interpreted. A C program is composed of several modules; the different modules are loaded together and then interpreted as if you were using a linker with a compiler.

The source files are loaded for execution into one of the eight editor modules. You can therefore split a C program into at most eight modules.

The other type of files contain binary executable code.

These files contain compiled functions that can be called by an interpreted C program.

A project lets you link interpretable C files with machine code files and indicate to **HiSoft C** exactly which files make up the program.

Note that a project can consist of just interpreted C modules. In this case loading the project loads several files in one operation.

1.10.2 Loading a project

A project is described in a file with extension .PRO. When you load a project **HiSoft C** reads the file names of the C files and compiled files and loads them into memory in one single operation.

For example, on the Examples disk there is the project `FILL.PRO`. This describes a project consisting of a C file ad a function written in assembler.

When you click on the Load project item on the Execute menu, the file selector will appear. Select the filename of the project that you want to load.

The loading of C and machine code files happens automatically according to the description in the project file (`FILL.PRO` for example).

All you now have to do is click on Run to execute the whole project.

1.10.3 Executable functions

This section describes how to load machine code functions and link them to an interpreted program.

You can build a function library in assembler, compiled C or another language and then use these procedures with an interpreted C program.

These functions must always be in a special format. You can only have one function per file; the file name is the name of the function.

A library file=one procedure.

For example, if you write an assembler routine that you have called `elephant`, then you must assemble the code of the said routine to a file called `ELEPHANT.COD`.

In practice, **HiSoft C** ignores what is inside the compiled file - it assumes that the name of the file is the name of the procedure.

You can pass parameters to the function and it can return a value.

You can write your procedures in assembler or C although there are rules for writing them because library functions that are loaded by **HiSoft C** have a different format to ordinary executable (`.PRG`) files.

1.10.4 Assembly Language functions

If you aren't an assembly language wizard it is probably best to skip to the next section.

The entry point of your procedure is the first instruction of the program and you must return with an `RTS` instruction.

Do *not* use the GEMDOS `PTERM` to finish the program.

You don't have to set up using the program's base page as **HiSoft C** has already done so. Trying to do this again will cause unexpected results.

The following is a complete example of a function written in assembler. **HiSoft C** passes two parameters on the stack and the function returns the value 1.

This example is purely to show how this is done and is not at all useful...

As we have seen, the entry point of the routine is the first instruction in the program and it terminates with a `RTS`. There is no base page manipulation.

The two parameters that are passed are two pointers to strings of characters. In the C program this function is called as follows: writ(str1,str2).

The `link` instruction makes it easier to access the parameters passed by **HiSoft C**. `8(a6)` is the address of the first parameter, `12(a6)` that of the second. If you have more than two parameters the third will be at `16(a6)` etc.

All parameters are passed as 4 bytes (a long word) whether they are characters, integers or pointers except for floats and doubles which take 8 bytes.

Function number 9 of trap 1 displays a string of characters on the screen.

```
link   a6,#0        ;Initialise A6 for the parameters.

move.l 8(a6),-(sp) ;write the first string
move.w #9,-(sp)
trap   #1

move.l 12(a6),-(sp) ;write the second
move.w #9,-(sp)
```

```
	trap	#1

	unlk	a6	;Restore A6

	move.l #1,d0	;store the value to be returned in D0
	rts	;end of the function;return to HiSoft C
```

The `unlk` instruction restores the A6 and stack registers to their original values.

The value returned by the function must be stored in D0. In our example the value returned is always 1. This value is recovered by the interpreted C program.

If the function returns a double precision 8 byte number then this is taken from D0 and D1.

Finally the `RTS` instruction finishes the function.

To produce an executable program you must assemble it to produce a standard executable file with just the above instructions. You can if you wish link with your assembly libraries (like the GEM ones supplied with DevpacST) but don't forget the following golden rules:

- the first instruction in the executable file must be the first instruction in the function.
- the function must finish with an `RTS`.
- don't mess with the base page.

1.10.5 Compiled C functions

This section describes how to use compiled C functions in your **HiSoft C** program.

As with assembler the C function must be at the very start of the module. The assembler example above is the following in C:

```
#include <osbind.h>

writ(str1,str2)
char *str1, *str2;
{
Cconws(str1);
Cconws(str2);
return (1);
}
```

The procedure `writ` is the first in the module; it does exactly the same thing as the little assembly language program.

To turn this into an executable file that **HiSoft C** can understand you must of course compile and link this function. But you must not include the 'startup' code in you program.

With most C compilers you explicitly link with this startup module. With Lattice C 5 this module is called `C.O`. With Lattice C 3, it was called `STARTUP.BIN`, and with Megamax C it is `INIT.O`. You must not link with any of these files. Remove the startup file from the file list that is used when you link. Your program will now be the first in the list.

With Aztec C, things are slightly easier, just compile with the `+B` option. This will remove the reference to the `.begin` label, so the start up code won't automatically be included.

Although you mustn't link with the usual startup code you can use the normal libraries (C, Gem etc).

So you produce a `.PRG` file, but one that cannot be directly executed by your ST because it does not have the correct initialisation. This is the file that you can load under **HiSoft C**.

1.10.6 Loading executable procedures. Project Files.

A project file consists of commands telling **HiSoft C** which files make up a project and thus which files must be loaded.

These files are of two types: interpreted C files and executable code files.

The project file consists of a series of lines.

If a line starts with a hash character (#), the line is treated as a comment and ignored.

If a line starts with a dollar ($), it is a command.

If not, it is a file name.

The commands are as follows:

`$REP <directory name>`	Specifies the directory in which the files are stored. You may have several of these in the same project file.
`$C`	Specifies that the following files are the C files to be loaded. These files will be treated as interpreted C.
`$ASM`	Indicates that the following filenames are those of executable files to be loaded by **HiSoft C**.
`$32 and $64`	Only relevant for executable files. They give the size of the value returned by the function stored within the executable file.This size is 32 bits by default for integers, characters, pointers etc. This size must be 64 bits for functions that return a double precision real number.

You may have several `$C` and `$ASM` commands in one project file.

Here is an example project file:

```
#==========================================================
#
# Project Description File
#
#==========================================================
# First part (optional) : $REP specifies the directory (le
# repertoire!) to load the files from. This $REP directive
# may occur several times in the file.
#
$REP D:\HC\EXAMPLES
#
#==========================================================
# Second part (optional) :
# C source files to load for HiSoft C to load.
# No more than 8 C files may be loaded
# The list of files must be entered after the $C directive
# one file per line.
$C
fill.c
#
```

```
#==================================================================
# Third stage (optional) :
# Machine code files to load
# Note that their format is special and is not that of
standard
# executable files.
# No more than 200 (!) may be loaded
# The list of files must be entered after the $C directive
# one file per line.
#
# $32 indicates that the functions return a 32 bit value
# (char, short,int or long).
# $64 indicates that the functions return a 64 bit double
# value
#
$ASM
$32
fill.cod
```

This file describes a very simple project containing only one C file fill.c and one executable file fill.cod.

Both files are stored in `D:\HC\EXAMPLES`.

The `fill` function is supposed to return a 32 bit value (the default case). This is also a sensible value to use for all functions that do not return values. If the function returns a double precision real value then you must specify `$64`.

When you select the option Load project from the Run menu all the files mentioned in the project file are loaded into memory.

1.10.7 Calling executable functions

An executable function that you load and call in a C program must be declared as `extern`. For example, the following line must be placed in the interpreted C program in order to use the `fill` function:

```
extern void fill();
```

An executable function may have parameters and may return a value. The definition and call are in the traditional C style.

1.10.8 Project Information

When you click on the Info about project item from the Run menu a list of loaded executable files appears on the screen together with the type returned by the function (32 or 64 bits).

1.11 The Disk Utilities

1.11.1 Loading the utilities

Disk Utilities is a program that can be loaded at the same time as the Interpreter and be co-resident in memory. You can switch between the disk utilities and the interpreter instantly to perform various operations on files and directories (copying, duplicating, deletion) and disks (formatting, information). When you return to **HiSoft C** you will be in exactly the same state as when you left. You can thus work with your files and disk without returning to the Desktop.

However, naturally the Disk Utilities use some memory. If you have 512K memory, you only have a limited amount of memory when using **HiSoft C** and loading the utility will reduce this further.

So you can decide whether to load the Disk Utility depending on how much memory you need.

If you have 512K, **HiSoft C** will not load the Disk Utility by default. This gives a 'decent' amount of usable memory for writing C programs.

If you have more than 512k the utility will automatically be loaded. It is available for use the moment that the interpreter has loaded.

However, you can change the default.

To load the disk utility you should select the Load disk util option from the Help menu. When a tick mark is present, every time you load **HiSoft C**, it will load the utility as well. If you remove the tick by selecting the menu item again then the utility is not loaded and your programs can use the memory that it would otherwise use.

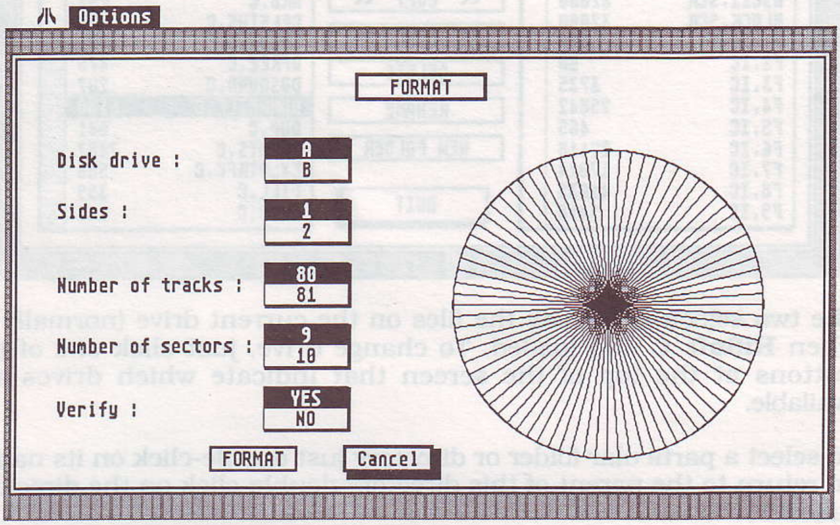
After selecting Load Utilties you should save the options and re-load **HiSoft C**. This is how **HiSoft C** remembers your choice. Then each time you load the interpreter **HiSoft C** will load or not load the utility as you have asked.

1.11.2 Using the Utilities

When you click on Disk utilities, the screen is cleared and a new menu appears. There is only one menu, called Options.

The menu item Return to HiSoft C does just that: it takes you back to the interpreter.

The Format command produces a dialog box like this:



FORMAT

Disk drive : [✓ A] B

Sides : [✓ 1] 2

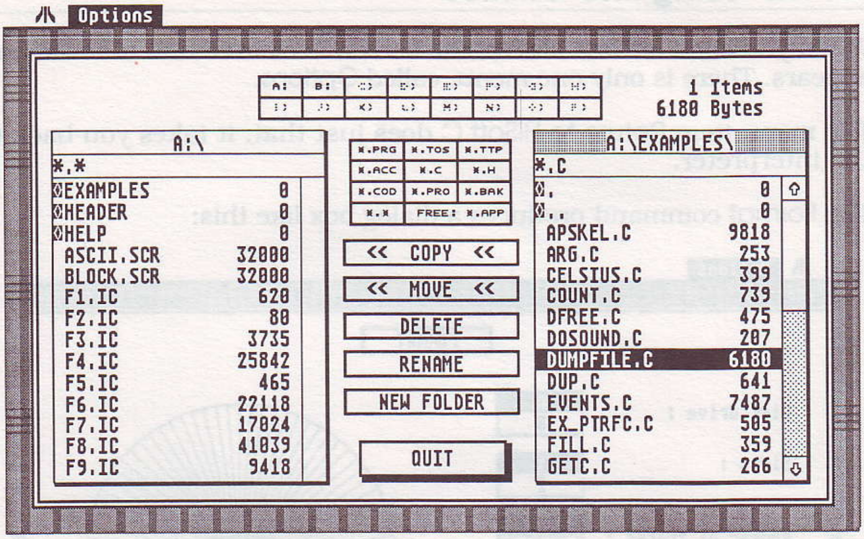
Number of tracks : [✓ 80] 81

Number of sectors : [✓ 9] 10

Verify : [✓ YES] NO

You can use this to format a disk, single- or double-sided, and specify several parameters (the number of sectors per track and the number of tracks per side) which will give you a little extra space on your disks. You may also indicate whether you want to verify that the disk has been formatted successfully. Clicking on Format will bring up a confirmation box and then format your disk.

The File Selector option brings up a dialog box like this:



```
                                                   1 Items
                                                6180 Bytes
```

```
A:\
*.*
EXAMPLES           0
HEADER             0
HELP               0
 ASCII.SCR     32000
 BLOCK.SCR     32000
 F1.IC           620
 F2.IC            80
 F3.IC          3735
 F4.IC         25842
 F5.IC           465
 F6.IC         22118
 F7.IC         17024
 F8.IC         41039
 F9.IC          9418
```

```
A:\EXAMPLES\
*.C
.                  0
..                 0
 APSKEL.C       9818
 ARG.C           253
 CELSIUS.C       399
 COUNT.C         739
 DFREE.C         475
 DOSOUND.C       207
 DUMPFILE.C     6180
 DUP.C           641
 EVENTS.C       7487
 EX_PTRFC.C      505
 FILL.C          359
 GETC.C          266
```

The two selectors display the files on the current drive (normally A) when **HiSoft C** was loaded. To change drive, just click one of the buttons at the top of the screen that indicate which drives are available.

To select a particular folder or directory just double-click on its name. To return to the parent of this directory double-click on the directory called "`..`".

By default, all the files in a directory are displayed; a selection of files can be displayed by choosing one of the 'filters' near the middle of the screen. To display just the `.C` files click on the button `*.C`.

The two windows can display the contents of different files or directories. To switch between windows, click on the appropriate window's title or information bar.

To delete or rename a file select the appropriate file and click on the `DELETE` or `RENAME` button. To copy or move a file, set up the destination directory (where the file will be copied or moved to) in one window and then select the file to copy in the other window. To perform the copy or move click on the appropriate button. `MOVE` is just like `COPY` except that the original file is deleted.

You may copy, move, delete or rename more than one file at once, just hold a `Shift` key down when clicking on a file; any currently selected files will remain selected.

You can find out how much used and free space there is on a particular drive by double-clicking on the appropriate drive button.

Finally, to create a new folder, first select the folder in which you want the new one to appear and then click on the `NEW FOLDER` button. You will then be prompted to enter the name of the new folder.

2 Introduction to the C language

This introduction to the C language is designed for people with some programming experience. We assume that you know and are familiar with the fundamental ideas of programs, statements, variables, conditionals and loops.

Some knowledge of the BASIC language is useful for this section as we will make comparisons with BASIC.

This is designed as a practical tutorial. There are a number of exercises and example problems. **Appendix A** contains answers to the exercises and suggestions for solutions. Note that certain ideas are discussed in the answers to the exercises. If you don't read the answers you will miss some vital information.

This section does not attempt to be a complete C language course but it nevertheless covers the working of a number of simple programs.

2.1 Your first program

2.1.1 The traditional approach

In just about every introduction to a programming language, the first program is one to display `hello` on the screen. So to follow the tradition of many generations of programmers, here is a C program that displays `Hello` and then `How are you?` on the screen and then waits for a key to be pressed.

```
main()
{
printf("Hello\n");
printf("How are you ?\n");
printf("press a key\n");
evnt_keybd();
}
```

That's it. It seems a bit more complicated than in BASIC.

2.1.2 The big moment

Load **HiSoft C** if you haven't already.

It's waiting for you to type a program - go to it!

If you have problems typing the program, see **Section 1**, the section describing the editor.

Always type carefully. In the C language upper and lower case letters are treated differently. The language keywords are always written in lower case. Two identifiers (variables or functions) made up of the same letters, one in capitals and one in lower case are different. Thus the names `language` and `Language` are not the same. You can't use one in place of the other.

Now that you have entered this example, the only thing left is to run it. To do this, click on the Run item on the Run menu. That's it.

Now let's look at the program more closely.

2.1.3 Functions, the fundamental unit in C

A program written in C is made up of functions. They are the bricks from which a program is built.

One of these functions has a special role; it's called `main`.

This function is essential because `main` is where the program starts executing. If there is no `main` function the program can't run.

The whole C program is built around this function. You can have other functions as well as `main` and these are called from the function `main`.

Keeping to this rule, the example above has a `main` function. Obviously it is the only function present in this program. So our first program is very simple in this sense although it may still seem complicated with its funny names and punctuation.

Strictly, a function is a set of statements enclosed in curly brackets and preceded by the function name.

The name of the function is followed by two round brackets (parentheses). This is explained in more detail below.

2.1.4 Statements

A function is made up of statements which must always be terminated by a semi-colon(;). The semi-colon indicates that the statement is over. It is similar to the colon(:) in BASIC, but in C every instruction is terminated by a semi-colon even if it is the only one on the line.

2.1.5 The C library

Our example has four statements. they are:

```
printf("hello\n");
printf("How are you ?\n");
printf("press a key\n");
evnt_keybd();
```

`printf` and `evnt_keybd` are functions from the C library (yes, functions again, well we warned you!). That is to say they have already been written for ease of use. All implementations of C have their own library of functions. They are invisible but very important. **HiSoft C** has over 460 of them.

If 460 functions isn't enough you can, of course, write your own. You have already started to write a function `main`. In fact there's no fundamental difference between library functions and those that you have written. Only that those in the library have been written already (in C of course) and you have to create your own functions yourself.

2.1.6 Calling Functions

Amongst the library functions are `evnt_keybd` which waits for a key to be pressed (short for keyboard event) and `printf` which displays a string of characters on the screen. The string is passed to the `printf` function. The values that are passed to functions are called the parameters or arguments of the function. The list of parameters to pass are given enclosed in round brackets. To conform to this rule the strings in our example are enclosed in parentheses in order to pass them to the function.

A function may have one, several, or no parameters. In our example the `printf` function has a single parameter. The `main` and `evnt_keybd` functions don't have any parameters. This is why the names `main` and `event_keybd` are followed by parentheses. We haven't seen any functions with several parameters yet. In this case the arguments are separated by commas.

2.1.7 Strings

A string is a collection of characters enclosed in double quote signs. What ever it contains a string is a valid element in language. The following are legal strings:

```
        "Hello"
```

and

```
        "How are you ?"
```

In our program we added the two characters `\n` at the end of the string just before the closing quotes. These two characters form one unit and mean together "newline". On the ST this means carriage return and line feed. So because of the `\n` in our example `How are you?` is on the line below `hello`.

Exercise 1

Produce the following display on the screen:

```
he
llo How are
you ?
```

and follow this with a message asking for a key press. There is no need to use any extra `printf` statements.

Exercise 2

Try to run this program after deleting a few characters to see what error messages they produce.

2.2 Variables

2.2.1 One more program

Here's a program that reads a key from the keyboard and displays it on the screen. Then the program waits for another key before it terminates:

```
int ch;
main()
{
        ch = evnt_keybd();
        putchar(ch);
        evnt_keybd();
}
```

Before typing this program, you must first delete the last program that you've already typed. All you need to do is select Abandon from the File menu. This is equivalent to New in BASIC.

Ok, so now type in this program and execute it by clicking on Run. Press a key on the keyboard and the corresponding character will be displayed on the screen. Press a key again and you'll be back in the editor.

2.2.2 Function or Statement ?

In C a function always returns a value. This is sometimes ignored by the programmer. However, the `evnt_keybd()` function can be used either as if it were a statement:

```
        evnt_keybd();
```

or as if it were a function. That is to say it can be treated as a routine that returns a value (the ASCII value of the key pressed in the case of `evnt_keybd()`), like this:

```
        ch = evnt_keybd();
```

2.2.3 Assignment

This last instruction assigns to the variable called `ch`, the value returned by `evnt_keybd`. The variable `ch` then contains the ASCII code of the key that was pressed. The symbol = is the assignment operator as in BASIC. Also, like BASIC

```
        i = 1;
```

gives the variable `i` the value 1. We're on familiar, friendly territory here. But there are mines ahead!

2.2.4 The putchar function

The statement

```
        putchar(ch);
```

writes the character whose ASCII code is stored in the variable `ch`. `putchar` is a C library function like `printf` or `evnt_keybd`. It has one parameter that is the character to display.

2.2.5 Declaring variables

Well, we left the best bit until the end. In C, every variable must be declared before it is used. This is so the user can indicate what type is associated with each variable.

In BASIC, variables are reals by default, integers are followed by `%` and string variables by `$`. Modern BASICs also let you use `!` to indicate single precision and `#` double precision.

This is how BASIC knows the type of each variable. In C it is totally different.

A scheme like that used by BASIC is impossible because there are so many types (in fact, there's theoretically an infinite number). The user must declare his or her variables, i.e. specify the type, before they are used. A variable may be a whole number or a real etc. This procedure seems fussy at first, it requires more thought when defining variables but means that some errors can be detected that otherwise wouldn't be, mis-spellings in particular.

2.2.6 Integers

We now know that all variables must be declared before they are used and that this declaration specifies the type of the variable. In our example, the declaration of the variable `ch` is

```
        int ch;
```

The keyword `int` is short for integer. So the statement above makes `ch` an integer. We say that `ch` is declared as type integer.

In **HiSoft C** integers are signed and use 4 bytes of storage.

An integer variable may therefore contain values between -2147483648 and +2147483647, which is quite enough for most purposes.

The type `int` is probably the most frequently used in C. But there are others.

Whole numbers will often fit into only two bytes. This is the type short. It's the equivalent to integers in most BASICs. The range of values is -32768 to 32767.

We can make our variable `ch` of type `short` because a character code can be between 0 and 255, which is entirely within -32768 to 32767. We do this as follows:

```
        short ch;
```

So this makes `ch` a short integer. This is the only change needed in the program because the ASCII code for the character can be represented by a short integer.

There are also whole numbers that fit into just one byte. This is the amount of storage needed for a character. For this type, the variables may take values between 0 and 255. This type is sensibly enough called `char`. So to indicate that `ch` is of type `char` all we have to do is use

```
        char ch;
```

and the program will work just as well.

We used the type `int` for `ch` to start with because this is more or less right, but it looks more elegant to declare it as type `char`. The difference between these three integer types is only a question of size; they are all integers.

2.2.7 Real numbers

HiSoft C supports real or floating point numbers. These are always double precision. Their values can vary between -1.8 E 308 and 1.8 E 308, and can be as small as 1 E -307 without being zero. They have 16 significant digits and use eight bytes of storage. The type of these numbers is `float` or `double`. As far as **HiSoft C** is concerned they are equivalent. The declarations of real numbers look like this:

```
float x;
double fl;
```

These two lines declare variables `x` and `fl` of real `type. float` and `double` are exactly the same type.

Floats aren't often used in C. Integers are used much more frequently, although engineers tend to use floats a lot!

2.2.8 Conclusion and Exercises

We have seen in this example how to use integer variables in C.

The types discussed above are the primary types. We can also construct arrays, structures and pointers based on these four types.

Exercise 3

Write a program which waits for two keys to be pressed and then writes the two characters on the screen. You should declare two variables.

2.3 Calculations

2.3.1 A little program

Try the following little program:

```
/* Examples of calculations with reals */
double real;
int whole;
short small;
main()
{
        whole = 1+56;
        small = whole/2;
        real = 10.2+2*small+sin(2.);
}
```

This example doesn't include much that is new. It just puts into practice what we have already seen.

2.3.2 Comments

You can add comments to your programs. Comments start with `/*` and finish with `*/`. Between these two pairs of characters anything goes. However a comment must fit on a single line (this is a **HiSoft C** restriction because it is an interpreter).

Comments are normally placed between statements and frequently at the ends of lines, whether they are empty or not.

2.3.3 Arithmetic operators

The C language has the following binary operators:

*	multiplication
/	division
%	modulus
+	addition
-	subtraction

These operators act on the two items to either side of the operator following the usual rules.

The modulus operator is the remainder when the first operand is divided by the second. Both arguments must be integers in this case. For the other four operators the arguments can be integers or reals.

The following priority rules hold:

The operators multiplication (*), division (/) and modulus (%) all have the same priority which is higher than the addition (+) and subtraction operators (-). These rules are the same in most languages, BASIC included.

Note that there are many other operators that we will cover later.

2.3.4 The arithmetic and trig functions

In our example, we used the function `sin` which calculates the sine function of the argument that it is passed. This argument must be expressed in radians. This is only one of the trigonometric functions available. Here's the full list:

```
    cos, sin, tan, atan, log, exp, sqr, asin, acos, floor
```

These function all take one compulsory parameter of type double. They return real values equal to the cosine (`cos`), sine (`sin`) tangent (`tan`), arctangent (`atan`), natural logarithm (`log`), natural anti-log (`exp`), square root (`sqr`), arc-sine (`asin`), arc-cosine (`acos`) or the whole number part (`floor`) of the argument.

If you want to pass a whole number, for example to find the logarithm of 5, you must follow the parameter with a decimal point to indicate that the number is really a floating point number. Otherwise you will be in for a surprise. For example,

```
        log(5.) + sin(2.);
```

2.3.5 Types and assignments

Returning to our example,

```
double real;
int whole;
short small;
```

The first three lines declare the three variables that we need in the program. `real` is a real number, `whole` is an integer and `small` is a short integer.

```
        whole  =  1+56;
        small  =  whole/2;
```

The first statement stores the value 57 in the variable `whole`. Then whole is divided by 2 and this value stored in the variable `small`.

`whole` and `small` aren't of the same type. But there is no problem assigning an `int` variable to a `short` one. More generally it doesn't matter which of the integer types (`int`, `short` or `char`) you assign to another integer variable. The only problem is that the value may have to be truncated. This is the case if you try to store 350 in a variable of type `char` which can't hold a value bigger than 255. No error message will be given but the value stored will be equal to 350 modulo 256.

The division of `whole` by 2 is an integer division because `whole` and 2 are both integers. As `whole` is 57, `whole/2` is 26 and not 26.5.

In general, within expressions where all the terms are integers, the `+`,`-`,`*` and `/` operators are all treated as integer operators and return whole numbers.

If, on the other hand, at least one of the terms in an expression is a real, all the other terms are converted to floating point numbers and `+`,`-`,`*` and `/` are used as real operators.

This is what happens in the third statement:

```
        real  =  10.2+2*small+sin(2.);
```

The terms `10.2` and `sine(2.)` are reals so the other terms in the expression (`2` and `small`) are converted to reals.

Then the calculations are performed and they return a real value which is assigned to the variable real. Thus a floating point expression is assigned to a floating point variable. In fact both sides of an assignment must be of the same type, that is to say, either both integer or both floating point. If you want to break this rule, you will need to use the explicit type conversions (see your favourite book on C).

2.4 Conditionals

2.4.1 Example

In C, like in BASIC, there is a conditional statement that lets you execute a series of instructions if a particular condition is true.

Type the following program:

```
int num;
main()
{
        num = Random ();
        if ((num / 2)*2 == num)
                printf("The number %d is even\n",
num);
        evnt_keybd();
}
```

The example uses the variable called `num`. It is declared in the first line of the program and is of type `int` or integer.

2.4.2 The Random function

`Random` is a function from the BIOS library (see **Chapter 4** for a description). Note that the first letter of this identifier is in upper case and that all the others are in lower case. This is a convention used by Atari for all the BIOS functions, so we use it too.

Right, back to where we were. `Random` returns a random number! It's a 24 bit positive whole number, that is between 0 and 16777215.

In the example, the random number is stored in the variable `num`. So we have an unknown value in `num`.

2.4.3 The if statement

We now suppose that we want to know whether this number is odd or even. The mathematical definition is that a number is even if it is divisible by two. More precisely a number, `num`, is even if `(num/2)*2` is equal to `num` where the division is integer division.

We can write this algorithm in the form:

```
    If (num/2)*2 is equal to num then
        write "the number 'num' is even"
```

In C this becomes:

```
    if ((num / 2)*2 == num)
            printf("The number %d is even\n", num);
```

The `if` statement has the form:

```
if (condition)
        statement;
```

The condition, which must be enclosed in round brackets, is evaluated. If this is true then the given statement is executed. If the condition is false then the statement is not executed and the program continues executing at the following instruction.

One small detail. To test if two expressions are equal we use the two characters ==. In BASIC only one equal sign is used. But two is better than one ...

There are other comparison operators:

!=	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
==	equal

For example, the condition that tests if the two terms `(num/2)*2` and `num` are different is

```
        (num/2)*2 != num
```

There are also logical operators so that you can test several conditions at the same time. These are equivalent to BASIC's AND, OR and NOT keywords.

||	OR	Logical OR between two conditions
&&	AND	Logical AND between two conditions
!	NOT	Negation of a condition

These operators will be discussed further in future examples.

2.4.4 printf... again

Thus if our number is by chance even, then the `printf` statement is executed.

We have already encountered `printf`, but that was in its most simple form. We have only had one parameter. Now there are two and also there's a strange `%d` in the string.

The two characters `%d` in the string `"The number %d is even\n"` indicates where to insert the number num in the output to the screen. For example, if `num` is 256 then the instruction:

```
        printf("The number %d is even\n", num);
```

writes on the screen:

```
The number 256 is even
```

So the value of the variable `num` is sent in place of the `%d`. The `%d` is powerful if cryptic, isn't it?

Thus `%d` replaces an integer value in the string. That's it. Does it seem complicated? Well, its like that regardless!

More seriously, you can write several variables with the `printf` function. For example here's an instruction that writes the values of the three variables `v1,v2` and `v3`:

```
printf("Here are three values : %d, %d and
%d",v1,v2,v3);
```

If these three variables are 1, 2 and 3 respectively then this will display:

```
Here are three values : 1, 2 and 3
```

You will have noticed that there are three lots of `%d` in the string and also three variables to write. The three variables are substituted for the three `%d`'s left to right.

The first variable `(v1)` is substituted for the first `%d` and so on.

Exercise 4

Write a program (complete with variable declarations and so on) which lets you see the three variables above.

2.4.5 if...else...

Even the simplest of machines have BASIC IF...THEN...ELSE... statements nowadays, and C doesn't stop there.

Let's improve our previous program.

In the last example a message was only displayed if the number was even but not if it was odd. Now let's have a message if it was odd as well.

```
int num;
main()
{
        num = Random ();
        if (num%2 == 1)
                printf("The number %d is odd\n", n);
        else
                printf("The number %d is even\n", n);
        evnt_keybd();
}
```

The algorithm for this program is simple:

```
    Make num a random number.
    if num is odd
    then write "The number 'num' is even"
    if not
        write "The number 'num' is even"

    Wait for a key press.
```

The form of the if...else instruction is

```
if (condition)
        statement1;
else
        statement2;
```

The expression, which must be enclosed in parentheses, is evaluated. If the `condition` is true then `statement1` is executed. If the `condition` is false, `statement1` is not executed but `statement2` is executed instead.

Examine the program in detail. There's a new little bit in the condition of the if. We use the modulus or remainder operator which is written as % in C. A number is even if its remainder when divided by two is 0. It's odd if the remainder is 1.

If the condition is true, that is to say the number is odd, then the first of the two `printf` statements is executed. Thus we could get:

```
The number 257 is odd
```

If the condition is false then the second instruction is executed:

```
The number 256 is even
```

2.4.6 Blocks

The examples we have seen so far only let you execute a single statement conditionally with `if`. Thus the syntax is:

```
if (condition)
        statement;
```

and not

```
if (condition)
        statement1;
        statement2;
        statement3;
```

What happens if you want to execute more than one instruction in the body of an `if`?

Answer: There's a structure that is always considered as a single statement but (and this is its secret) actually contains several statements.

This groups several statements into a *block*. A block can be put anywhere that you can put a single statement. Anywhere you see a statement in a syntax description you can put a block instead.

In practice, a block is a set of statements (followed by semi-colons as usual) and surrounded by curly brackets. Here's a block consisting of two printf statements:

```
{
        printf("Hi!  ");
        printf("How are you ?");
}
```

Note that the curly brackets surrounding the block are on lines by themselves and the statements are on separate lines. This is to make it more readable. However, this layout is not compulsory. The block above is equivalent to:

```
        {printf("Hi! ");printf("How are you?");}
```

The first block is more readable but the second is quicker to type.

Now that you know (we hope) what a block is, the syntax for an `if` statement can be written:

```
if (condition)
{
        statement;
        statement;  /* etc... */
}
else
{
        statement;
        statement;  /* etc... */
}
```

This uses the rule that a block can replace any instruction and vice versa.

The following sequence is equally correct.

```
if (condition)
        statement;
else
{
        statement;
        statement;
}
```

Note that a block can contain just one statement.

Exercise 5

Write a program that generates an even random number from some random number. The program should display neatly whether the initial random number is odd or even. If it is already even it should be left as it is.

2.5 Loops

Loops enable you to execute the same sequence of statements several times.

In old-style BASIC there is only one sort of loop. Everyone knows the famous FOR...TO...NEXT statement. There's a similar statement in C called `for`.

In slightly more modern BASIC's there's a WHILE...WEND loop. This executes a particular set of statements while a condition is true. This statement is also called `while` in C.

Finally there's a third statement which is a variant on the `while` statement called `do...while`.

2.5.1 The while statement

Type and run the following program:

```
int i;
main()
{
        i = 1;
        while (i <= 20)
        {
                printf("%d squared is %d\n", i, i*i);
                i = i+1;
        }
        evnt_keybd();
}
```

If you have followed the previous section you can probably guess what this program does.

It finds the squares of all the numbers between 1 and 20.

The algorithm used by this program is:

```
initialise the number to one.
while the number is less than 20 do
    write the square of the number
    add one to the number
```

Let's look at this program in detail.

The variable `i` is declared in the first line as of type integer. The first statement in the program is `i=1;` which stores the value 1 in the variable `i`.

The `while` statement repeatedly executes the block containing the following two statements:

```
        {
                printf("%d squared is %d\n", i, i*i);
                i = i+1;
        }
```

while the condition `i<=20` is true. The two statements are repeated twenty times during the execution of the program to write the twenty squares on the screen.

The syntax of `while` statements is

```
while(condition)
        statement;
```

or

```
while(condition)
{
        statement;
        statement; /* etc...*/
}
```

The one or more statements that form the body of the `while` statement are executed whilst the condition is true.

The two groups of `%d` in the mysterious `printf` statement insert the values of `i` and `i*i` in the string to be output. This follows the rules we have seen in **Section 2.4.4**.

The statement `i = i + 1` adds the value 1 to the variable `i`, just like you would write in BASIC.

The statement `evnt_keybd` waits for a key to be pressed on the keyboard.

Note that if the condition in a `while` statement is false on entry the statements in the loop aren't executed at all, as in the following program:

```
int i;
main()
{
        i = 5;
        while(i<4)
        {
                printf("toto");
                i++;
        }
}
```

Exercise 6

The example above writes the first twenty squares in increasing order. Write a program which writes the squares of whole numbers starting with 20 and finishing with 1.

2.5.2 The for statement

In BASIC the FOR...NEXT statement uses a loop counter variable. When the said variable reaches a certain value the program stops executing the statements between the FOR and NEXT statements. It's nearly the same in C.

The following program writes the squares of the numbers between 1 and twenty that we've already seen using a `for` statement.

```
int i;
main()
{
        for (i = 1; i <= 20; i = i + 1)
                printf("%d squared is %d\n", i, i*i);
        evnt_keybd();
}
```

This example is exactly equivalent to the previous one. To prove it, run it!

The algorithm for this program is as follows:

```
Varying i from 1 to 20 in steps of 1 do
    write the square of i
Wait for a key press
```

In BASIC this corresponds to

```
FOR I=1 TO 20
        PRINT I;"squared is";I*I
NEXT I
A$ = INKEY$
```

if we remember BASIC correctly.

In detail, the C syntax for `for` statements is

```
for (statement1; condition; statement2)
        statement3;
rest of the program;
```

or alternatively

```
for (statement1; condition; statement2)
{
        statement3;
        statement4; /* etc... */
}
rest of program;
```

When **HiSoft C** comes across such a statement, it does the following:

It executes `statement1` first of all (`i=1`). Next it evaluates the `condition` (`i<=20`).

If this is false, **HiSoft C** starts executing the rest of the program. If, however, the condition is true, **HiSoft C** executes `statement3` then `statement4`... which constitutes the body of the `for` statement (`printf`...)

Finally `statement2` is executed (`i=i+1`).

The condition is evaluated once more.

If it is false, **HiSoft C** executes the rest of the program. But if it is still true, **HiSoft C** executes `statement3`, `statement4` and then `statement2` again. The condition is evaluated once more...

But perhaps we are repeating ourselves.

Looking at it another way, `statement1` is executed. Then the body statements followed by `statement2` while the condition is true.

So the statement above is exactly equivalent to:

```
statement1;
while(condition)
        {
        statement3;
        statement4;
        statement2;
}
rest of the program;
```

Note the numbering of the statements.

Exercise 7

This is the same as the last one. Rewrite the program above (with the `for` statement) so that it writes the squares in reverse order using a `for` statement.

Exercise 8

Write a program which finds the sum of the whole numbers between 1 and 100.

Exercise 9

Write a program that displays the ASCII character set. Remember that `putchar(ch)` writes the character whose code is in the variable `ch`.

2.5.3 The do...while statement

This loop structure is used less often than the two previous ones. It is based on the `while` statement and resembles it a lot.

```
char ch;
main()
{
        do
        {
                ch = evnt_keybd();
                putchar(ch);
        }
        while (ch != 13);
}
```

A character is read from the keyboard, stored in the variable `ch`, then written to the screen. This is repeated until the character read is a `Return` (ASCII code 13).

The syntax of the `do...while` statement is as follows:

```
do
        statement;
while  (condition);
```

or

```
do
{
        statement1;
        statement2;
        statement3;  /*  etc...  */
}
while(condition);
```

Note the semi-colon after the `while (condition);`.

The statements are executed while the condition is true.

The difference between the = statement that we saw above and this one is that the body of the statement is executed first and then the condition is evaluated. If it is true, then the program continues to loop. If not, the `do...while` loop finishes.

In the standard `while` loop the condition is tested before the body statements. As a result, with a `do..while` the body statement is always executed at least once, whereas with the `while` statement it may never be executed.

2.6 Switch statements

A switch lets you execute one of several statements depending on a condition. For example, if a variable or expression has one of several values.

The statements that are executed are different depending on the value. This is equivalent to the SELECT CASE statement in **HiSoft BASIC**.

Let's take a simple slot machine as an example.

A number between 0 and 9 is chosen by chance. If you get a number 5 or 7 you win. 7 is the jackpot. If you get a 6 you lose.

The algorithm is:

```
Select a number by chance

Depending on the number
    if it is 6
        write "you have lost"
    if it is 7
        write "Jackpot"
    if it is 5 or 7
        write "you have won"
    other values
        write "you haven't won or lost"
```

Translated into C this becomes:

```
int num;
main()
{
printf("Put a 10p coin in the slot\n");
evnt_keybd();
num = Random() % 10;
switch (num)
{
case 6:
        printf("You have got 6\n");
        printf("You've lost\n");
        break ;
case 7:
        printf("You have got 7\n");
        printf("Jackpot\n");
case 5:
        printf("You have won\n");
        break ;
default :
        printf("You have got %d\n", num);
        printf("and have neither won nor lost\n");
        break ;
}
evnt_keybd();
}
```

Let's examine this program in detail.

The variable `num` is an integer which contains the number chosen by chance.

To obtain this, we use the function `Random` to return a whole random number. We take this value modulo 10 to make it a random number between 0 and 9.

Then we come to the interesting bit of this example.

It's the `switch` statement. `Switch` executes one of several statements depending on an integer value. The expression is evaluated and then compared with a list of constants. When a match is found execution starts with the corresponding statement.

In our example, the said expression is the variable `num`. Depending on the value of this variable we know whether you have won or lost because `number` contains the value that was chosen by chance.

In C, the phrase "if it is 6" translates to `case 6:`. Don't forget the colon.

Thus if the variable has the value 6 then the two statements:

```
        printf("You have got 6\n");
        printf("You've lost\n");
```

which follow `case 6:` are executed.

Just after these two statements is the keyword `break`. This means in this case (if `num=6`) the switch statement is finished.

Thus if the value is 6 the messages are written on the screen and you have lost.

Now, we'll look at the next case. It's the value 7; good news, it's the jackpot and the appropriate message is written in the screen.

```
        printf("You have got 7\n");
        printf("Jackpot\n");
```

Next it's the turn of 5. You've won but not the jackpot. So just the fact that you have won is displayed.

```
        printf("You have won\n");
```

Immediately afterwards there's a `break` statement which tells us that the `switch` statement is over and the rest of the program is executed.

If you have been paying close attention, you will have noticed that there is no `break` statement after `case 7`.

Let's take another look:

```
case 7:
        printf("You have got 7\n");
        printf("Jackpot\n");
case 5:
        printf("You have won\n");
        break ;
```

A 7 has been chosen. **HiSoft C** executes the two `printf` statements after `case 7`. But there's no `break` to tell the interpreter to stop, so it continues and executes the statement `printf("You have won\n");` from case 5. After that, there's a break statement so the switch is finished. The next statement in the program will then be executed (`event_keybd()`).

The last part of the `switch` statement is `default:`. This is a special possible switch value that is executed only if no match has been found. So in our slot machine program the following statements will be executed if our random number isn't 5,6 or 7.

```
        printf("You have got %d\n", num);
        printf("and have neither won nor lost\n");
```

Note that you don't *have* to have a `default` in a `switch` statement. In this case if the `switch` variable doesn't match one of the other cases, no statements are executed as part of the switch.

More rigorously, the switch syntax is as follows:

```
switch(int_expression)
{
case constant-1:
        statement-1;
        break;

case constant-2:
        statement-2;
        statement-3;
        break;

case constant-3:
        statement-4;
case constant-4:
        statement-5;

default:
        statement-6;
        statement-7;
        break;
}
```

So in English, this does the following:

If the `int_expression` is equal to `constant-1`, then `statement-1` is executed.

If the `int_expression` is equal to `constant-2`, then `statement-2` and `statement-3` are executed.

If the `int_expression` is equal to `constant-3`, then `statements 4,5,6` and `7` are executed.

If the `int_expression` is equal to `constant-4`, then `statements 5,6` and `7` are executed.

If the `int_expression` is not equal to any of the `constants 1,2,3` or `4` then both statements `6` and `7` are executed.

Note that the expression following `switch` is in parentheses, a `case` must be followed by a colon and all the case items are enclosed in curly brackets.

Exercise 10

Write a program that reads a character from the keyboard, and then displays it if it is A or B, otherwise it displays an asterisk instead. Pressing the space bar should stop the program. You can use a `switch` statement or several `if` statements.

2.7 Functions

2.7.1 Functions and subroutines

At the start of this introduction to the C language we said that a C program is made up of one or more functions which are called from one another. The moment has come to prove it.

A function is a set of statements that are grouped together. Generally there is something linking these statements, that is to say they implement a particular operation.

There are two sorts of functions: library functions and those that you write yourself.

Library functions are subroutines that are already written and are available for your use. You've already used the library functions called `printf`, `putchar`, and `evnt_keybd`.

The functions that you write yourself are the equivalent of subroutines in BASIC that are called using GOSUB. (Actually they are more like SUB and DEF FNxx definitions in **HiSoft BASIC**).

C functions have names of their own which are used to call them. In addition you can pass information between the calling program and the function.

Let's look at a concrete example. We shall create a function called `wait`.

This will write "press a key" on the screen and wait for a key to be typed on the keyboard. This function is useful when you are displaying messages, as it gives the user plenty of time to read what is on the screen.

```
main()
{
        printf("message 1\n");
        wait();
        printf("message 2\n");
        wait();
}
wait()
{
        printf("Press a key\n");
        evnt_keybd();
}
```

The `wait` function consists of two statements. The first writes `Press a key` and the second waits for a key to be typed.

The `main` function has four statements. When you run the program, these and only these are executed. The statements in the function `wait` are not executed unless they are called by the `main` function.

The program starts by displaying `message 1`. Then it calls the function `wait`, which waits for a key to be pressed. After the user has typed a key, `message 2` is displayed. The function `wait` completes the program and it terminates as soon as the user presses another key.

So far functions seem just like GOSUB subroutines in BASIC. However functions are more powerful than that. For example, they can have parameters.

2.7.2 Parameters

Remember our function `wait`. Suppose we want to `wait` for a particular key, but it isn't always the same one. For example, the first time we want to wait for the a key and the second time the b key. This is a little artificial, but better than nothing.

There's a neat solution using parameters:

```
char ch;
main()
{
        printf("message 1\n");
        wait('a');
        printf("message 2\n");
        wait('b');
}

wait(c)
char c;
{
        printf("Press the %c key \n", c);
        do
                ch = evnt_keybd();
        while (ch != c);
}
```

The only real difference between this and the last program is the `wait` function. Now this function has a parameter which is the character that must be typed on the keyboard.

The `main` function calls the `wait` function with a parameter `'a'` and then with a parameter `'b'`. This indicates the key to wait for. Both of these values are the integers whose values are the ASCII codes for `a` and `b`.

Inside the `wait` function the parameter is called `c`. This is a variable which contains the value that was passed as a parameter. During the two calls it takes the value of first `'a'` and then `'b'`.

Be careful, the variable `c` can only be used in the `wait` function. All attempts to access it outside will be greeted with an error message. We say that `c` is a *local* variable of the function `wait`.

You have learnt to declare all variables before using them, and parameters are no exception to this rule. They must be declared just after the name of the function and before the statements that make up the function. See the example.

In our program, the variable `c` is of type `char`. This declares the variable inside the function; you can not access it outside its function, the `wait` function. Inside the `wait` function, we have a variable containing the character to wait for.

The first statement in this function writes a message to the screen asking the user to press the appropriate key.

Notice the bizarre `%c` inside the `printf` string. In a similar way to `%d` meaning 'write an integer here', `%c` means 'write a character here'. So a character will appear on the screen rather than a whole number.

The ASCII code of this character is in the variable `c` which was passed as a parameter.

Thus, if `c` has the value 97 which is the ASCII code for the character `'a'`, the statement:

```
        printf("Press the %c key \n", c);
```

displays

```
Press the a key
```

on the screen. If you change the `%c` to `%d` then

```
        printf("Press the %d key \n", c);
```

will show

```
Press the 97 key
```

Next, after indicating which is the key to wait for, **HiSoft C** executes the following loop:

```
        do
                ch = evnt_keybd();
        while (ch != c);
```

which reads a character from the keyboard (using `evnt_keybd`) whilst that character is different to the parameter of the function.

2.7.3 Return values

A function may return a value to the calling program. For example, `event_keybd` sends back the code of the key that has been pressed.

Let's look at an example. Suppose we want to create a power function which raises a whole number to the power that it has been given. This function will be called from the main program in a couple of places to calculate 2^{10} and 2^{16}. The algorithm used will be a brute force one.

The two parameters of this function are the number to raise and the power to raise it to. The value returned by this power function is the result of the calculation.

```
int val10;
int val16;
int result;
main()
{
        val10 = power(2, 10);
        val16 = power(2, 16);

        printf("2^10 = %d, 2^16 = %d\n", val10, val16);
        evnt_keybd();
}

power(number, exponent)
int number, exponent;
{
        result = number;
        while (exponent > 1)
        {
                exponent--;
                result *= number;
        }
        return (result);
}
```

The two integer variables `val10` and `val16` contain the result of the calculations of 2^{10} and 2^{16}. The `main` function stores 1024 in `val10` and 65536 in `val16`. The `printf` statement displays these values on the screen, thus,

```
2^10=1024, 2^16=65536
```

The `power` function has two parameters called `number` and `exponent`, and they are both declared as type integer.

The statement:

```
        exponent--;
```

subtracts one from the variable `exponent` and the statement:

```
        result *= number;
```

multiplies `result` by `number`.

The variable `result` will thus contain the result of the calculation when the loop has finished. All it needs to do now is return this value to the calling program. This is done with the very simple statement:

```
        return (result);
```

This statement stops execution of the function and returns control to the calling program. In addition, the value in parentheses is taken as the result of the function.

2.7.4 Summary

The general syntax of a function is as follows:

```
function-name(arg1, arg2, ...,argn)
type arg1;
type arg2;
...
type argn;
{
        statement;
        statement;
        ...
        statement;
        return (expression);
}
```

`function-name` is the name of the function.

`arg1,arg2, ..., argn` are the arguments to the function.

`type` is a type specifier (e.g. `int,char`, etc...)

The characters in *italics* are compulsory, the others are optional.

The arguments are optional. There may be none, one or several. But if there are any, then they must be declared. The parameter declarations must be after the list of arguments and before the opening bracket and statements of the function.

Exercise 11

Write a function which does nothing, with no statements, and no parameters that is called from a main function which doesn't do anything either. An exceptionally useful program...

Exercise 12

Write a program that reads two numbers from the keyboard and displays their sum. The `getchar` function reads a character from the keyboard, echoing it and returns the ASCII code of the character. This is similar to our friend `evnt_keybd` except that `getchar` displays the character on the screen. To build up to this, write a function that reads a number from the keyboard where this may have several digits.

2.8 Arrays

The C language has arrays like BASIC. For simple cases (arrays of `int`, `short`, `char` or `double`) the structures used are almost identical to those in BASIC.

Let's consider the example of determining which numbers are produced most frequently by the random number generator.

To do this we will have an array of twenty integers 0 to 19. We'll generate a random number between 0 and 19. To count the numbers, we will increment the array element whose index is the random number. For example, if it is 6 we will increment element number 6 of the array. When we've finished we will know how many times each number has been chosen.

Increment means 'add one to'.

Those of you who can read French, may be interested in the following from the French version of this manual:

«Ajouter deux, c'est deuxcrémenter, ajouter trois, c'est troiscrémenter, etc... Ne pas confondre deuxcrémenter avec décrémenter, qui signifie retrancher un.»

```
int arr[20];
int i;
main()
{
      for (i = 0; i < 20; i++)
            arr[i] = 0;
      for (i = 0; i<1000; i++)
            arr[Random() % 20]++;
      for (i = 0; i < 20; i++)
            printf("%d : %d\n", i, arr[i]);
      evnt_keybd();
}
```

The first line declares an array of 20 elements of type `int`. It is called `arr`.

In BASIC, this corresponds to DIM ARR(20).

In BASIC, array indices are enclosed in parentheses, in C you use square brackets. It's not our fault!

In the program we'll need an integer variable. In a fit of originality it's called `i`.

The first statement in the program is a `for` loop. It initialises all the elements of an array to zero. The variable `i` varies from 0 to 19 (`i=0` and `i<20`).

When we declare an array of integers with `int arr[20];` the indices vary between 0 and 19. So `arr[0]` is the first element and `arr[19]` is the last.

But be careful, there are no checks for array indices. If you want to read from or, worse, write to element `arr[450]` no-one will stop you. But 9 times out of 10 the machine will crash.

Is this stupid? Yes and No. The language is defined this way and it can be very useful in certain cases. Be *very* careful with your array indices.

HiSoft C has an option to test to see if pointers have certain nasty values and if so give an error. This may happen if you get your array indices wrong or then again it may not.

Let's get back to our program.

The third statement is another `for` loop:

```
        for (i = 0; i<1000; i++)
                arr[Random() % 20]++;
```

This loop is executed 1000 times. That is to say the statement `arr[Random() % 20]++` is executed 1000 times. So the variable `i` is a loop count that varies between 0 and 999 or 1000 times.

This statement that is executed 1000 times works as follows: a random number is chosen using the `Random()` function. To make this between 0 and 19 we find its remainder when divided by twenty using `Random()%20`. So we've got a random number between 0 and 19.

This value is used as the index for the element that we want to increment. For example if 6 is chosen it will increment element number 6 of `arr`. Using this we can determine which numbers occur most frequently.

The element of the array we want is called `arr[Random()%20]` as `Random()%20` is the index.

You can wipe your brow now, we're nearly there!

We have seen in several exercises that to add one to the variable `i`, we can just write `i++`. This is equivalent to `i=i+1`.

So to add 1 to `arr[Random()%20]` we just need

```
        arr[Random() % 20]++;
```

That's it. It doesn't take long to write but it does a lot.

The program repeats this statement 1000 times, picking 1000 random numbers and incrementing the corresponding array element.

When execution of the loop finishes, most of the work of the program is done; all we need to do is display the results on the screen. For each element of the array, we are going to write its index (between 0 and 19) and its value (theoretically between 0 and 19).

We have a loop whose counter goes from 0 to 19 and uses a `printf` statement for the screen output:

```
for (i = 0; i < 20; i++)
        printf("%d : %d\n", i, arr[i]);
```

We've used `i` yet again as the loop variable, this time varying between 0 and 19 (`i=0` to `i<20`).

The `printf` function has three parameters.

The last two numbers represent the two numbers to write: `i` and `arr[i]`.

The first parameter gives the format with which the numbers are to be written. It's a string `"%d : d\n"`. The first `%d` replaces the value of `i` and the second writes the value in `arr[i]`. If, for example, `i` has the value 6 and `arr[i]` is 52, then the statement:

```
        printf("%d : %d\n", i, arr[i]);
```

displays the following :

```
        6 : 52
```

The two characters `\n` cause the cursor to move to the start of the next line so that the display for each element is on its own line.

2.9 Conclusion

We have seen in this section the basics (pardon the pun) of programming in the C language. C is a language that needs more care than BASIC or Pascal but it is also more powerful. We have seen this in the last example.

You can write things very concisely in C. This can make programs more difficult to read but can also mean that they run quicker when compiled.

We haven't seen all the features of C so far; the C that is explained in this section corresponds to a modern BASIC. C has many other facilities. To summarise, structured types, pointers, 42 arithmetic & logical operators, macro pre-processor, modular programs, pointers to functions, recursion etc...

To discover all these, equip yourself with one of the books in the bibliography, your **HiSoft C** disks and a large dose of patience!

3 Introduction to GEM

This section will describe some of the secrets of GEM that you can uncover using the toolbox of functions that will let you open windows, use dialog boxes and menus.

3.1 Programming with GEM

3.1.1 GEM itself

GEM stands for Graphics Environment Manager. GEM lets you "manage your environment graphically" via icons and windows, letting you open windows, pull down menus move icons etc...

For the C programmer (that's you), GEM is available to you via a library of functions.

For example, there's a function to open a window, and another to close it. Another function draws a line between two points, another displays text with formatting attributes, and another waits for a menu selection.

GEM has about 200 of these functions altogether. Suitably used they let you create programs with windows and pull down menus. All the GEM AES and VDI functions are available from **HiSoft C**, but the **HiSoft C** toolbox has been created to make things easier.

3.1.2 The HiSoft C toolbox

GEM lets you create windows and pull down menus. However if you've tried programming with GEM before, you've probably realised that calling it is arduous to say the least.

The documentation doesn't help. The only official documents are based on those for the IBM-PC and designed for professional programmers (but then you have to be a registered developer to get hold of it!) and doesn't include a single proper example. Most of the other books available are shortened versions of the official ones, but are just as difficult for a beginner to use. There's one thing to be thankful for, at least these are written in English (well American anyway)!

Most people find learning to program GEM long and hard work. GEM is powerful but huge. For example, to write a program which opens a window on the screen and closes it immediately takes about forty lines of C. In assembler, it's even worse and you need about 100 instructions.

This is because you need to use lots of functions to program with GEM. For this reason, **HiSoft C** includes a library or toolbox of functions that lets you use most of GEM's facilities but is very much simpler to use. The toolbox means you can use GEM effectively and easily. These function provide a 'software cushion' between you and GEM. You don't need to dive into hundreds of pages of documentation to work out how to do each little thing.

We haven't ignored the purists however. All the standard GEM library functions are available with their standard names. If you have been through the struggle of learning GEM with another language implementation, you can use your hard-earned skills straight away.

3.2 Windows

3.2.1 What is a window?

You already know what windows are, because you have used the GEM Desktop and **HiSoft C**. A window is a part of the screen with a border which you can change using sliders, arrow boxes etc.

As far as GEM is concerned, a window is nothing more and nothing less than a border on the screen. You might think that there is a GEM function that would let you move within a window on a virtual screen of say 1000 lines and 200 columns and update the display... but no. GEM has no concept of virtual screens.

When you open a screen window, GEM draws the border; that's all. It is up to the programmer to generate the display that appears in that window. You have to create your own virtual window and use this to generate the window display.

Now we shall look at the **HiSoft C** toolbox functions for windows.

3.2.2 Opening and closing a window.

The first thing we need to do to open a window is to open it, as you probably guessed. The function to do this is called `open_window`.

When you have finished using a window you must close it using `close_window`.

Type in and run the following short program:

```
int window_no;
main()
{
        window_no = open_window(4095, 20, 20, 400, 150,
                " window ", " press a key ");
        evnt_keybd();
        close_window(window_no);
}
```

In the first line, we declare an integer variable called `window_no`, then we have our usual `main()` and `{`.

The first statement proper in the program calls the `open_window` function. This draws the window on the screen and has so many parameters that they are on two lines.

The `evnt_keybd()` waits for a key press once the window is opened but before the window is closed with `close_window`.

3.2.3 The open_window function

Let's look in detail at the `open_window` function. It has 7 parameters.

The first of these indicates the attributes of the window. For example, whether it has a title, sliders, arrows, etc...

Each attribute has a number associated with it. The following table gives a list of the numbers corresponding to the various attributes.

base 10	**base 16**	**Attribute**
1	0x001	Title bar with name
2	0x002	Close box
4	0x004	Full window
8	0x008	Window can be moved
16	0x010	Information line
32	0x020	Change window size
64	0x040	Up arrow
128	0x080	Down arrow
256	0x100	Vertical slider
512	0x200	Left arrow
1024	0x400	Right arrow
2048	0x800	Horizontal slider

If you want your window to have several attributes, you use the sum of the numbers corresponding to each attribute.

For example, if you want a window with a title, two sliders, and a close box then the parameter value is 1+2+256+2048 = 2307.

With our program, we've used the value 4095. This value is the sum of all the attributes above, so our window will be drawn with all the possible 'gadgets'.

The next four parameters give the co-ordinates and size of the window. They are, in order, the x and y co-ordinates of the top left of the window, the width and the height.

These four values are expressed in pixel co-ordinates. The x co-ordinate may be between 0 and 639 (in medium & high resolution) and the y co-ordinate may be between 0 and 199 (medium) or 0 and 399 (high). The origin is the top left of the screen.

In the example, our window has a width of 400 pixels and a height of 150. Its top left corner has co-ordinates (20,20).

The sixth parameter is the name of the window and it is a string. If you don't want your window to have a title use the empty string `""`.

Our window has the original name `"window"`.

The final parameter is the text that is displayed in the information line just below the title. Again, if you have no information line use a null string `""`.

The `open_window` function returns an integer. This tells us which window we have opened and gives us a *handle* to use when calling the other window routines, so that they know which window to operate on. So for example when we want to display text in our window we use this handle to say which window to write to.

This is a similar idea to the file pointers that are used to access files. When you open a file you are given a file handle to access it subsequently.

If GEM can't open the window then the value returned is 0. Your program should cope with this error condition.

So that's it. When you specify all the parameters, the window appears on the screen as if by a miracle.

3.2.4 The close_window function

An opened window must always be closed, and you mustn't close a window that hasn't been opened. It makes sense. The `close_window` function does this job for us:

```
close_window(window_no);
```

It has one parameter which is the integer returned by the `open_window` function. This indicates which window is to be closed and is used in the functions that use windows; `close_window` is no exception.

The `close_window` function returns a value indicating the success or otherwise of the close operation. If there is a problem, such as an attempt to close a window that isn't open, the value returned is 0; if all went well the value returned is not zero.

The returned value is only provided for information. Normally it will always succeed. This value is often ignored, as in the example above. The calling program simply 'throws away' the value returned.

3.2.5 Writing in a window

We have seen that a window is a screen area with a border, no more no less. So we need some functions to change the display in the window.

The `print_window` function lets us write into a window without bothering about the position of the text. The following program, after opening the same window as the previous example, writes some text inside the window.

```
int window_no;
main()
{
        window_no = open_window(4095, 20, 20, 400, 150,
                " window ", " press a key ");
        print_window(window_no, "line 1");
        print_window(window_no, "this is line 2");
        print_window(window_no, "this is line 3");
        evnt_keybd();
        close_window(window_no);
}
```

This program is identical to the previous one apart from the three calls to `print_window` that have been added. Three messages are written in the window.

Run this program to see the effect of this function.

3.2.6 The print_window function

`print_window` has two arguments.

The first is the integer handle of the window that is returned by `open_window`. It indicates which window is to be written to.

The second is the text to display in the window. It is a string of characters enclosed in quotes.

The `print_window` function returns a value indicating whether there was a problem writing the text in the window. If an error occurs then 0 is returned otherwise a non-zero value is passed back to the calling function.

In practice, this never fails unless the window handle is wrong or if the string is a pointer to an invalid area of memory. Thus this return value is often ignored as in the example above.

This function positions the text itself. The first time you use this function, the text is written on the first line of the window. Subsequent calls write the text on the following lines. Thus a newline is always inserted at the end of each string that is written. If you want to write somewhere else, use the `pos_window` function. `print_window` works in a similar way to PRINT in BASIC.

If the text is too long to display in the window, the end of the message isn't displayed - it's clipped to fit in the window.

If you write too many lines to the window, nothing is shown below the bottom of the window and it will not scroll; any subsequent text will not be displayed but will be ignored.

The display is always within the border of the window.

3.2.7 The pos_window function

This function lets you change the position where the text written with `print_window` is displayed.

The program below writes `"hello"` at line 5 column 5 and then closes the window:

```
int window;
main()
{
        window = open_window(2307, 20, 20, 400, 150,
                "title","");
        pos_window(window, 5, 5);
        print_window(window, "hello");
        evnt_keybd();
        close_window(window);
}
```

The window is opened with a title, both sliders and a close box. The values of these attributes are 1, 2, 256 and 2048. Their sum is 2307, and this is used as the first parameter of `open_window`.

The co-ordinates and size of the window haven't been changed from the previous example. They are (20,20) and (400,150) respectively.

The window name has changed to `"title"` and there is no information line and so the last parameter is an empty string "".

The statement

```
pos_window(window, 5, 5);
```

indicates that the next string is to be displayed at line 5 column 5 within the window that we have opened.

Then the text is displayed at the position fixed by the `pos_window` statement.

Finally, after waiting for a key, we close the window.

Let's take a closer look at the `pos_window` function. It has three parameters.

The first is the integer that is returned by the `open_window` function. It indicates which window is to be written to.

The next two parameters indicate where text is to be displayed. They are first the column and then the line. Both values are specified in characters relative to the top left of the window. Thus they must be between 0 and 79 for the column and 0 and 24 for the line. If the values used are too large, and the cursor is thus positioned outside the window, no text will be displayed by the next `print_window` call. No indication of this error is given.

If an error is detected then this function returns 0, otherwise a non-zero value is returned. This will only happen if the window number is wrong and so this is often ignored.

Note that the `pos_window` function doesn't actually display any text. It must be used in conjunction with `print_window`. Text specified to `print_window` appears at the position given by `pos_window`.

3.2.8 Clearing a window

There's a function to clear out the inside of a window. The following program opens a window, and writes `"hello"` into it and waits for a key press. It then clears the window contents, displays `"Hello, world"` and waits for another key. Finally the window is closed.

```
int window;
main()
{
        window = open_window(2307, 20, 20, 400, 150,
                "title","");
        print_window(window, "hello");
        evnt_keybd();
        clear_window(window);
        print_window(window, "hello, world");
        evnt_keybd();
        close_window(window);
        }
```

The function `clear_window` clears out a window. It has one single parameter. This is the integer that is returned by the `open_window` function. It indicates which window is to be cleared.

The value returned by this function is an error indicator. As usual it is 0 if there is an error and non-zero if the operation is successful. This value is normally ignored.

Exercise 13

Write a program that opens two windows and writes alternately to them both.

3.2.9 The size_window function

This function lets you find out the size of the available work area inside a window.

In practice, this space is the total space used by the window when opened minus the space used by the window's borders.

The borders contain the title bar, sliders etc...

Here's an example of a program that draws a straight line between the opposite corners of its window.

```
int x, y, width, height, win;
main()
{
      win = open_window(2307, 20, 20, 400, 150,
            "title","");
      size_window(win, &x, &y, &width, &height);
      draw(x, y, x+width, y+height);
      evnt_keybd();
      close_window(win);
}
```

The first line of the program declares the integers `x`, `y`, `width` and `height` which will contain the work area of the window.

The integer `win` is used to store the handle of the window that we are using.

The window is opened by the `open_window` function. Then the size of the inside of the window is obtained by calling `size_window`.

The function draws a line between two corners of the window, we wait for a key press and then the window is closed.

Let's examine the `size_window` parameters in detail. It has five parameters.

The first is the window handle as returned by the `open_window` function. This indicates which window to find the size of.

The four other parameters are going to receive the co-ordinates of the area inside the window. These are, in order, the x and y co-ordinates of the top left corner, then the width and the height; these parameters are modified during the call of the function. They are set up to be the work area of the window.

In this program you should have noted the presence of the `&` operator before the names of these parameters in the function call.

This `&` symbol is absolutely essential for the `size_window` function and indicates that the parameters are to be modified by the function (see **Section 3.2.11** for the details of this).

3.2.10 The draw function

The `draw` function is new.

It draws a straight line between two points. It has four parameters.

The first two parameters are the x and y pixel co-ordinates of the first point. Their values must be between 0 and 639 for x, and 0 and 399 or 0 and 199 depending on the resolution for y.

The other two parameters are the graphics co-ordinates of the other end of the line.

If the co-ordinates are outside the limits given above then only that part of the line visible on the screen is drawn.

This function does not return a value.

3.2.11 Ways of passing arguments

A function parameter is only a copy of a variable. Only the value of the variable is passed to the function. Thus, modifying a parameter within a function does not modify the variable that is used when the function is called. We say that the parameter is passed *by value*.

However, there is a method so that you can modify variables that are passed as parameters. All you have to do is precede them with the `&` operator.

Then it is the variable itself that is passed and it can be modified. We say that this is a *variable parameter*.

If you have understood this properly, the hardest part is over,

Be careful, as in C if a parameter is to be used by value it must be called that way. Similarly when using a variable parameter you must use the `&` sign.

Otherwise 99 out of a 100 times the machine will crash.

As a general rule, parameters are passed by value. No problem. Every so often a function must modify a parameter. Such a parameter must always be preceded by a `&`.

Exercise 14

Write a program which opens a window of a random size.

Then, inside this window draw a ray of lines in all directions starting from one corner of the window.

3.2.12 Conclusion

We have seen in this section how to open a window and how to draw and print inside it.

The use of sliders, arrows, and close boxes is described in **Section 3.5** concerning events.

3.3 Dialog boxes

3.3.1 What is a dialog box?

You have already seen dialog boxes. They are a sort of window without sliders or title bars that appear in middle of the screen and ask you for information.

For example, when you select Set Preferences from the Desktop, a dialog box appears and asks if you want confirmation for copies, what screen resolution you want etc.

Such a box consists of an outer box and various objects such as buttons, text, and icons and which can be selected with the mouse or changed with the keyboard.

3.3.2 Creating a dialog box

There are two stages in using a dialog box in your program.

First create your box. That is, define its size and its various objects (buttons, text, icons etc) that you want to put in it. This is the first stage that the user of the program doesn't see.

Then you must display the box on the screen and let the operator use it. This is the second stage.

Note that the box creation stage is totally separate from its display on the screen.

A box is thus created at the start of the program, once and for all, but it can be displayed on the screen as many times as you like without having to redefine it.

The creation of a box itself has several stages. First its size and the number of elements within it must be defined with the `init_box` function.

Then, one by one, we add the different objects that we want to appear inside this box.

This is done with the help of the functions: `text_box` (add a string), `button_box` (add an exit button), `gtext_box` (add graphics text), `edit_box` (add an editable text field) and `color_box` (specify the colours of the text, background, and border of graphics or editable text).

Now that we have created our box, we can display it and let the user modify it. This is achieved with the `draw_box` function.

It only remains to find out what modifications to the state of the dialog box the user has made. We use the `readstr_box` (read a string of text) and `readbut_box` (read the state of a button).

It is important to note that only the `draw_box` function has an affect that the user can actually see. The creation of the box is invisible. It is only when we call this function that anything appears on the screen.

3.3.3 The init_box function

This function creates and initialises a dialog box. It's the function that must be called first and specifies the size of the box and the number of objects that will be placed inside it. Put simply, an empty box is created. This provides the ground work for you to position text and buttons.

This function has three parameters.

The first two specify the size that you want to give the box. They are, in order, the width and height. Both values are expressed in characters. They must be between 1 and 24 for the height and 1 and 80 for the width.

The third parameter indicates the maximum number of elements that you will be allowed to add to this dialog box.

For example, if you give this value as 5, you may put up to 5 elements in the box. You can thus, for example, use 2 buttons and 2 strings. Or equally, 1 button and 4 strings, so long as there aren't more than 5 objects.

This function returns an integer. It is the handle for the box that we have just created. This is its number. It lets us distinguish between different dialog boxes when we want to use several.

For example, let's create a dialog box 20 characters wide by 5 characters high. Inside it we only want a string of text and an exit button, i.e. two objects. The statement to start off the creation of this box is:

```
box = init_box(20, 5, 2);
```

This function doesn't produce any result on the screen. It is only when we have completely defined this box that we can call `draw_box` and it will appear on the screen.

3.3.4 The text_box function

A dialog box contains messages to indicate, for example, how to use it. These strings of text are positioned in the box using the `text_box` function.

You must create an empty box using the `init_box` function before calling this function, like all the functions that add objects to a dialog box.

The function `text_box` has four parameters that let you specify the box to which it is to be added, where to position the text, and finally the text itself.

The first parameter is the integer that is returned by `init_box`. It indicates in which box the text dialog is to be positioned.

The following two parameters let you position the text within the dialog box. They are the co-ordinates of the first text character in the form column and then line. They are expressed in numbers of characters relative to the top left of the dialog box. For example if you give both values as 0 then the text appears in the top left of the box.

The last parameter is the text that is to be placed in the box. It is a string of characters enclosed in quotes.

This function returns an integer. It is the number of the object within the box. This value can be ignored, except when you need to distinguish between objects that you have added to the dialog box. This is generally ignored because text is simply displayed on the screen rather than being used to exchange information with the user. Its state is never tested as it would be if it were a button that the user could select.

This example builds on the example of `init_box`. We have added two `text_box` statements to add two text strings to the box:

```
        box = init_box(20, 5, 2);
        text_box(box, 1, 1, "text 1");
        text_box(box, 1, 3, "text 2");
```

These three statements don't make up a valid program because you must have a button in a dialog box so you can click on it to exit the dialog box.

3.3.5 The button_box function

Buttons are used for two purposes in dialog boxes. They let the user choose between several possibilities using *radio* buttons. For example, when you want to search for a word using the Find command of **HiSoft C** you are presented with a dialog box and you can choose the direction of the search by clicking on the appropriate button.

The second use of buttons is to close the dialog box. These are often the usual Ok and Cancel buttons.

The `button_box` function lets us add one of these buttons to a dialog box. We can specify its position and its type (radio or exit).

It has five parameters. The first four are the same as for the `text_box` function.

The first is the integer that is returned by `init_box`. This handle indicates to which dialog box the button is to be added.

The next two parameters are the co-ordinates to position the button in the box. They are in character co-ordinates (line then column).

The fourth is the text of the button, as a string of characters in quotes.

The final parameter is unique to the button_box function. It indicates the button's type. This value is detailed in the reference section under button_box.

There are two types of button.

Radio (or choice) buttons let the user select one of several possibilities without leaving the dialog box. For these this parameter should be 17.

Exit buttons cause the dialog box to be closed when the user clicks on them. If you want to create such a button use 5 for the final parameter.

This function returns an integer; this is the index of the object within the dialog box. It is used to distinguish this button from other ones.

The following examples show some simple uses of the button_box function:

```
        ok = button_box(box_no,1, 2, "OK", 5);
```

This example creates an exit button (last parameter=5) in the dialog box given by box_no. This is placed at line 2 column 1 from the top left of the dialog box. It contains the text "OK".

Finally the number of this box is stored in the variable called ok.

Now for a second example:

```
        choice1 = button_box(box_no, 4, 3, "Choice 1", 17);
        choice2 = button_box(box_no, 4, 4, "Choice 2", 17);
        choice3 = button_box(box_no, 4, 5, "Choice 3", 17);
```

These three statements create three buttons that will let the user choose between Choice 1, Choice 2 and Choice 3. The value 17 as the last parameter means that these are radio buttons.

The three buttons are inserted in the dialog box given by box_no all in column 4 but one beneath another in lines 3, 4 and 5. The text inside these buttons is Choice 1, Choice 2 and Choice 3 respectively.

These buttons aren't exit buttons (as the last parameter is 17). When you click on one of them it will be selected and appear highlighted, but the dialog box isn't closed.

When the user clicks, for example, on Choice 1 this will be the only button selected. If either of the other two had been selected they will return to normal.

Then, when the choice has been made, we click on OK to confirm the choice and exit the dialog box.

3.3.6 The draw_box function

We have seen that this is the function that actually draws a dialog box on the screen.

In general, a box is first created using the three functions that we have just discussed. Then it is displayed in its full splendour all at once.

The draw_box function provides this complicated display function. The box is drawn on the screen, the mouse cursor appears and the user can select the buttons and edit the text that have been drawn in the box.

Then, when the user clicks on an exit button, the box is removed and control returns to the calling program.

Here's an example so we don't get out of practice.

Type in and run the following program:

```
int box;
main()
{
        box = init_box(18, 8, 2);
        button_box(box, 6, 6, "   OK   ", 7);
        text_box(box, 2, 2, "click on OK");
        draw_box(box);
}
```

Let's examine this example in detail.

The first line declares an integer variable called box. This will be used to store the number of the box that we are building up.

The first proper statement initialises the box. It is to be 18 characters wide (the first parameter) and 8 characters high (the second parameter).

The box will contain a total of 2 items (third parameter).

The statement:

```
        button_box(box, 6, 6, "  OK  ", 7);
```

adds the first of these objects to the box. It is a button with the text `OK` and positioned at column 6 line 6. This is an exit button (last parameter is 7). So when this button is clicked on the dialog box will be closed.

The second object is added using:

```
        text_box(box, 2, 2, "click on OK");
```

This is a string at column 2 line 2 and its text is `click on OK`.

Finally the statement:

```
        draw_box(box);
```

draws the box on the screen and waits for the user to click on `OK`.

The `draw_box` function returns an integer result representing which button caused the box to be exited. This is the number of the box as returned by `button_box` when the button was created. You need to know which exit button was pressed if there is more than one exit button.

This value is ignored in the above example because there is only one exit button; it must be this one that caused the box to be closed, so the program can safely ignore this information.

3.3.7 The readbut_box function and radio buttons

There are two types of buttons.

We have already seen that these are exit buttons, which cause the dialog box to be closed and radio buttons, that let the user choose between several options.

For buttons in the latter category we need to know which button the user has chosen. The `draw_box` function returns which exit button the user clicked on but not the last exit button that was pressed.

To do this there is the `readbut_box` function. It tells us if a button is selected or not.

In the following example a dialog box is displayed on the screen and the user is given a choice. She or he must click on a button corresponding to their age. Finally the user must close the box by clicking on OK.

```
int box;
int age1, age2, age3;
main()
{
        box = init_box(30, 10, 5);
        button_box(box, 12, 8, "  OK  ", 7);
        text_box(box, 4, 2, "Select your age...");
        age1 = button_box(box, 1, 5, " 00-20  ", 17);
        age2 = button_box(box, 11, 5, " 21-40  ", 17);
        age3 = button_box(box, 21, 5, " 41-99  ", 17);

        draw_box(box);
        if (readbut_box(box, age1) == 1)
                printf("Youngster...\n");
        if (readbut_box(box, age2) == 1)
                printf("In the prime of your life\n");
        if (readbut_box(box, age3) == 1)
                printf("Ancient...\n");
        printf("press a key\n");
        evnt_keybd();
}
```

The variable `box` contains the dialog box number. The integers `age1`, `age2` and `age3` are where the numbers of the three buttons containing age ranges are stored.

The first real statement initialises the box to have a width of 30 characters and a height of 10 characters. Inside the box we can insert 5 objects which in this case will be, a string, an exit button and three radio buttons.

In line 8 at column 12 we position the OK exit button, thus,

```
        button_box(box, 12, 8, "  OK  ", 7);
```

The value 7 as the last parameter means that this is a default exit button. This is the button that will be selected if we type `Return`.

Next the string `Select your age` is added at column 4 line 2.

```
        text_box(box, 4, 2, "Select your age...");
```

The statement

```
        age1 = button_box(box, 1, 5, " 00-20  ", 17);
```

positions the first of the three radio buttons in the box. It represents the choice of 0 to 20 years. The last parameter is 17 because this is a radio button.

The index of this button is stored in the variable age1.

The two statements:

```
        age2 = button_box(box, 11, 5, " 21-40  ", 17);
        age3 = button_box(box, 21, 5, " 41-99  ", 17);
```

position the other two radio buttons next to the first at columns 11 and 21 where the first one was in column 1.

The three numbers of the radio buttons are stored in the three variables age1 (0-20), age2 (21-40) and age3 (41-99).

Next the box is drawn on the screen using the call to the draw_box function. The user must select one of the three ages and then press on the OK button.

The program must now work out which of the three buttons has been selected with the help of the readbut_box function. This tells us if a radio button has been selected or not and has two parameters.

The first is the number of the dialog box in which the button is located. This is the integer that was returned when the box was created with init_box.

The second parameter is the number of the radio button whose state we are finding, within the box. This is the value returned by button_box when the radio button was created.

The function returns a value, the state of the box. If this value is 0 then the button is not selected. If however the result is 1 then the button is selected. It's that simple.

Back to our example. The statement:

```
        if (readbut_box(box, age1) == 1)
                printf("Youngster...\n");
```

tests if the button 0-20 whose number is stored in the age1 variable is selected. If it is then the value returned by readbut_box will be 1 and the printf statement is executed.

In the same way the next two statements test the other two radio buttons.

Note that only one radio button is ever selected. If the user tries to select another one, the original one is de-selected.

The `readbut_box` function should only be called after a `draw_box` statement where a dialog box will have been displayed and the user has selected a button. This button should always be a radio button that has been created using `button_box`.

3.3.8 The edit_box function and editable fields

The dialog boxes we have created so far let you click on buttons as much as you like, but they don't let you enter text using the keyboard. For example, when you create a new folder on a disk the Desktop will display a dialog box and ask you to type the name of the folder.

The `edit_box` function lets you create such items so that you can enter text.

For example the following program asks the user to enter the current date :

```
int box;
int edit;
char *date;
main()
{
        box = init_box(33, 7, 3);
        button_box(box, 14, 5, " OK ", 7);
        text_box(box, 1, 1, "Enter to-day's date...");
        edit = edit_box(box,11,3,"250687","__/__/__",
                "999999",3);
        draw_box(box);  /* draw the box */
        date = readstr_box(box, edit); /* read the date */
        puts(date);     /* write the date on the screen */
        evnt_keybd();
}
```

This program starts in a way with which you are familiar. A dialog box is created and an exit button called `OK` and a text string `Enter to-day's date` is added to it.

To change the standard recipe, we have added a new call to add a bit of spice. This is the `edit_box` function and it has seven parameters.

Ok, let's start at the beginning.

The first three parameters we have seen before. They are the dialog box number, the column and line where the text will be displayed.

An editable field (where we can enter text) has three parts.

First there is the *template* (or site) of the field. It is where the text is physically written. In general it uses the underline characters: "________". These characters are replaced by the data that we want to write. This is the fifth parameter to `edit_box`.

In our example the string consists of "__/__/__". This lets us type six characters with slashes between each pair.

Thus, we need to know the maximum size of text in advance.

Next there are the characters that we are allowed to enter, or *validation* string. We can just accept digits or only upper case letters for example.

For each character that we can enter, we must specify which characters are allowed. To do this, each "_" character in the template field has another character associated with it. This is represented by a string of characters indicating which characters are valid.

In this example, this string is `"999999"`, meaning that only digits are allowed. No other character may be typed. There are many other possibilities however. For example, using just the character `"A"`, upper case letters only are allowed.

Here's the full selection:

`9`	Only 0-9
`A`	A-Z and space
`a`	A-Z, a-z, 128-255 and space
`F`	A-Z, a-z, 0-9, 128-255, : ? * _
`f`	A-Z, a-z, 0-9, 128-255, _
`N`	A-Z, 0-9 and space
`n`	A-Z, a-z, 0-9, 128-255 and space
`P`	A-Z, a-z, 0-9, 128-255, \ : ? * . _
`p`	A-Z, a-z, 0-9, 128-255, \ : _
`X`	all characters allowed

In the above A-Z includes non-English capital letters. 128-255 means that all characters with value greater than 128 can be used including lower case non-English letters and the £ sign. Note that the f validation character is not normally documented but is present in all versions of the operating system that we have used.

We could replace the "999999" in our example with "XXXXXX". Then the user could type any character on the keyboard not just digits.

Finally, the third part of an editable text field is the text that will be displayed for the user to modify if he wishes. This is the fourth parameter in the edit_box function call. In the example this will write 250687. This is combined with the template (the fifth parameter) to produce the display 25/06/87. When you run the program, it won't be that date so you will need to delete it (using the Esc or Backspace keys) and enter the new one.

We could replace the "250687" in the function call with "". This would make the field empty when the dialog box is displayed producing "__/__/__", and you wouldn't have to delete the old date before entering the new one.

An editable field may, or may not have a border round it. The last parameter of the function specifies the border. 0 means no border, 1 to 3 mean borders of increasing thickness. If this parameter is not between 0 and 3 no border is drawn.

This function returns the index of the edit field within the dialog box.

So at last, we have finished with the edit_box function.

Next in our example, the draw_box function is called to display the box and let the user enter the date.

3.3.9 The readstr_box function

This function is used to read text that has been entered using an editable field. In our example we want to find the date that has been entered by the user.

To do this we use the readstr_box function which is much simpler than edit_box, it only has two parameters.

The first is the number of the dialog box, where the text that we want to read is placed and the second is the number of the editable field within that box. This is the value returned by edit_box when the item was created.

As if by magic, the value returned is the string of characters that was typed; in our example the date.

For C experts, the type of the value returned is `char*`, that is pointer to character. It's the address where the characters typed were stored.

In our example we have a call to `puts` to display the string containing the date on the screen.

3.3.10 Graphics Text

It is possible to have text with graphics attributes in dialog box.

So far we have just had boring strings, but they may have different colours, border, fill patterns and even different sized characters.

```
int box;
int text;
main()
{
        box = init_box(20, 10, 3);
        button_box(box, 6, 8, "OK", 7);
        text_box(box, 3, 2, "click on OK");
        text = gtext_box(box,1,5,"graphics text",1,2,1);
        color_box(box, text, 0, 2, 3);
        draw_box(box);
}
```

This example displays a string in small characters, with a black border, red characters (on a colour screen!) and on a green background.

You will remember the usual box creation function (20 characters wide, 10 high, containing 3 objects). A button and a string are added to the box with text of `OK` and `click on OK`.

Then a couple of new functions `gtext_box` (to create graphics text) and `color_box` (to change the colour of the text). The box is then displayed using `draw_box` as ever.

3.3.11 The gtext_box function

This function places a graphics text item in a dialog box. This is similar to a `text_box` but the text can have several effects applied to it to relieve the boredom.

This function has seven parameters. The first is the number of the dialog box to which the text will be added. It is always the first parameter in all the object addition functions.

The next two parameters give the position of the text within the box as in the `text_box` function for example.

The fourth parameter, if we've counted right, is the string of characters that are the actual message text of the box.

So far this has been exactly the same as the `text_box` function. But this changes with the next parameter which is the size of the characters. If this is 0 then they are written as normal, otherwise they are written as small text.

The parameter before the last one is for the border of the text box, with 0 meaning no border, 3 a very wide one and 1 and 2 meaning borders in between.

The background on which the text is written can have the 'darkness' of the fill specified. This is a value between 0 (white) and 7 (full black) with values from 1 to 6 being shades of grey.

You may need to use the `color_box` function so you can read your text!

The `gtext_box` returns an integer, which is the index of the object within its box as ever. In general this value is ignored as the text is simply displayed on the screen rather than being used in interaction with the user. However there is one time when you need this value and that is when calling `color_box`.

3.3.12 The color_box function

As we have already said this function is used to set the colours of graphics text. It has five arguments.

The first is the number of the dialog box in which the graphics text has been added. The second is the index of the text within that box. This is the value returned by the `gtext_box` function.

The last three parameters indicate the colours of the border, the characters and the background.

With a monochrome screen only the values 0 and 1 should be used, corresponding to white and black respectively.

If you are using a colour screen in medium resolution 4 values are allowed:

0	white
1	black
2	red
3	green

For example, if you want a black border, red characters and green background, specify these parameters as 1, 2 and 3.

If you don't like these colours they can be changed with the Control Panel or using the GEMVDI `vs_color` function.

Note that if you want a non-white background you will need to specify a non-zero fill pattern in your call to `gtext_box`.

Be careful, you can change the colour of text created with `edit_box` or `gtext_box` only. You can not do this for buttons (`button_box`) or non-graphics text (`text_box`).

3.4 Menus

3.4.1 What is a menu?

You've used pull down menus. There is a bar at the top of the screen with titles which have menus on them which drop down when you move the mouse over them.

Using the **HiSoft C** toolbox, this is done as follows:

```
int title, elem;
main()
{
        init_menu(" Desk ", " About Menu", 2, 5);
        title_menu(" File ");
                item_menu(" Load File");
                item_menu(" Save File");
                item_menu(" Quit");
        title_menu(" Options ");
                 item_menu(" Search");
                 item_menu(" Replace");
        draw_menu();
        event(&title, &elem, 0, 0, 0, 0);
}
```

This program creates a menu with three titles: Desk, File and Options. In the File menu there are three entries and in the Options menu there are two.

The menu is drawn on the screen (using the `draw_menu` function), then the user selects an item from the menu (using the `event` function) before the menu disappears.

The `event` function is detailed in the section on events (See **Section 3.5**). This is a **HiSoft C** toolbox routine; it manages the mouse and lets you pull down menus when the mouse reaches the menu bar. It returns control to the program when a menu item is selected.

It is important to note that the `init_menu`, `title_menu` and `item_menu` items create the menu in a way that is invisible to the user. Nothing changes on the screen. Only during the call to the `draw_menu` function does the menu bar appear. The menus can't be pulled down until the `event` call.

3.4.2 The init_menu function

This lets you initialise and reserve space in memory for a menu that you wish to create. It is the function that must be called first during creation of a menu.

`init_menu` has five parameters.

The first is a string of characters representing the title of the first menu. This is nearly always Desk but you can change it if you like. This isn't the only choice you have...

The second argument is another string of characters. It's the first item on the Desk menu and is normally used to display an 'About' box for the program describing the program, its author etc. This string must not be more than twenty characters long. In the example, we've used `"About Menu"`.

The next parameter is the number of titles of our own that we wish to have without containing the Desk menu.

So in our example this parameter has the value 2.

For the **HiSoft C** editor, with the titles File, Find, Run, Move, Block, Help and Info, there are 7 titles. To create such a menu this parameter must have the value 7.

Note that this value is the maximum number of titles that you wish to include in the menu bar. You can indicate that you want more titles than you actually add to your menu.

Finally, the last argument indicates the maximum number of menu entries, not counting the Desk menu items nor the menu titles.

For example, in the **HiSoft C** menu there are 63 entries. In the program above, there are 5.

The `init_menu` function returns an integer. This is the index of the 'About' box in the Desk menu. This number is used to see whether it was the item that was selected. (See **Section 3.5** on events).

3.4.3 The title_menu function

After intialising a menu with `init_menu`, we can start to add the titles in the menu bar after the Desk menu. This piece of magic is performed with the help of the `title_menu` function.

The new title is added to the right of the Desk menu if there aren't any other titles. If you have already called `title_menu` then the new title will appear to the left of the existing ones.

The title bar may not contain more than 80 characters.

`title_menu` only has one parameter. It is the string of characters that you wish to add to the menu bar. So in our example it is `" File "` or `" Options"`.

A value is returned by this function. It is the integer that identifies the title that we have added to the menu bar. We need this to find out from which menu the user has selected an item. See **Section 3.5 Events**.

3.4.4 The item_menu function

This function creates an entry beneath a menu. This item is added below the last title that was added to the menu bar.

This function must not be called before using `title_menu`. `title_menu` is used to create a title and then `item_menu` is used to add the entries beneath that menu.

The value returned by this function is the integer that identifies the item that we have added to the menu bar. We need this to find out which item the user has selected. See **Section 3.5 Events**.

3.4.5 The draw_menu and delete_menu functions

These two functions display or remove the menu bar.

The function `draw_menu` must be called after the menu has been created (using the `init_menu`, `title_menu` and `item_menu` functions) and before the user can interact with the menu (using the `event` function).

If you forget this function, no menu events can occur and the `event` function can wait for ever for a menu event that will never happen.

Quickly turn to **Section 3.5** on events if the event concept isn't crystal clear.

Neither of these functions have any parameters and both return an integer indicating whether an error occurred. This is an integer with value 1 (or true) if all went well or 0 (or false) if an error occurs.

Generally this value is ignored by the program, assuming (hopefully!) that all went well.

3.4.6 The enable_menu, check_menu, & select_menu functions

We've grouped these three functions as they are all used when changing the state of a menu. We won't give an example here because it would need to build on the other menu functions and so be rather long. There is an extended example in the examples folder of master disk 2 called `event.c`.

`check_menu` makes a tick (or check mark) appear or disappear in front of a menu item. If the item isn't already ticked the tick appears; if it is ticked it will disappear.

`enable_menu` 'greys' an item in a menu. That is, it makes the item appear grey so the user may not select it with the mouse. If the item is already grey it appears as normal and so is enabled once more, hence the name of the function.

Finally, `select_menu` makes a title appear in inverse (black on white) or if it is already selected in this way it returns to normal. The most common use for this is to restore menu titles to normal after an item has been selected.

These three functions all have one parameter and it is used for the same purpose in all of them.

This parameter indicates which item is to be selected, greyed or ticked. This is the value returned by the `item_menu` or `title_menu` function.

All three functions return an integer. This indicates the new state of the item. If the item is now selected, disabled or ticked 1 is returned; if not 0 is returned. If you try to change the state of an item that does not exist then 0 is returned.

The `select_menu` function may only be used with titles; trying this with menu items will always return 0.

`select_menu` has a rather special purpose. When you select a menu with the mouse the title appears in inverse and remains that way until you call `select_menu` with the number of the title as its argument. You can see this in the `paleochrome` program.

Here is a tiny program fragment that ticks a menu entry.

```
int item;
item = item_menu(" menu item");
check_menu(item);
```

3.5 Events

This is the section whose importance we stressed a few pages back.

3.5.1 What is an event ?

Events are the fundamental items in a true GEM program.

They are single user actions as far as GEM is concerned. This might be a key press, a mouse button press, the selection of a menu item or the manipulation of a window (moving it, changing the size, clicking on an arrow or slider etc).

An event is therefore GEM's way of telling you what the user has just done with the mouse or keyboard.

A program running under GEM with pull-down menus and windows is based around this concept. Some mis-guided programmers try to avoid this, but this is a waste of time; it is much better to do things GEM's way rather than fighting it!

3.5.2 The layout of a GEM application

The `event` function is by far the most powerful and important GEM function. GEM applications are built round this function.

It lets us wait for the user to complete one of the actions described below.

So thanks to this function you can ask GEM to wait for a key press and/or mouse click, and/or select a menu, and/or change a window. It's powerful isn't it?

When you make the call you specify which events you want to wait for. The function returns to the main program when one of the specified events has occurred.

For example, you can wait until the user selects a menu or presses a key. You call the `event` function 'saying' that you want to wait for a menu or keyboard event. As if by magic, the function doesn't return until the user has chosen a menu item or pressed a key.

You will be told whether a key was pressed or that a menu item was selected.

Typically, a GEM application uses events in the following way:

```
create the menu
create the dialog boxes
open the windows
While we haven't finished do the following:
    wait for an event
    Depending on the type of the event
        If it is a mouse event:
            deal with the mouse event
        If it is a keyboard event
            deal with the keyboard event
        If it is a menu event
            deal with the menu event
        If it is a window event:
            deal with the window event
Close the window
Remove the menu
```

If we translate this algorithm into C, we get:

```
int menu_title, menu_item; /* menu item selected */
int x, y;                  /* mouse co-ordinates */
int event_type;            /* event type
(keyboard,mouse..)*/
int ch;                    /* character typed on keyboard */
short window[6];           /* window event details*/
int notfinished;           /* indicates if the program */
                           /* has finished or not */


main()
{
        create_menu();
        create_dialog();
        openwindows();
```

```
    while (notfinished)
    {
            event_type = event(&menu_title, &menu_item,
                            window,&ch, &x, &y);
            if (event_type == 2)    /* mouse button */
                    do_mouse(x, y);
            if (event_type == 3)    /* menu selected */
                    do_menu(menu_title, menu_item);
            if (event_type == 1)    /* key pressed */
                    do_key(ch);
            if (event_type == 4)    /* window changed */
                    do_window(window);
    }
    destroy_menu();
    closewindows();
}
```

Let's look at this in detail.

The variables declared at the start of the program are used to store the type of event and details about a particular event. They are modified by the call to the `event` function.

Thus the integers, `menu_title` and `menu_item` contain the indices of the title and item of the selected menu if a menu event occurs.

The `x` and `y` variables contain the position of the mouse if the user clicks on a mouse button.

`ch` will contain the key that was pressed on the keyboard if a keyboard event has occurred.

The array `window` contains extra information about a window event. For example that a slider has been moved or that the window size has been changed.

`event_type` contains the type of event that has just happened. This may be a menu, keyboard, mouse or window event.

The integer `notfinished` is used to indicate when the program has terminated. If, for example, the user clicks on a close box or selects Quit from the menu we would set this variable to 0 (false).

The first three statements in the program initialise the environment by calling the functions which open windows, and create menus and dialog boxes. We won't go into detail about these here.

Then we have a `while` loop which waits for and deals with events until the user decides to exit the program.

Now for the big moment - the `event` function is called.

This waits for a keyboard, mouse or window event. See the detailed description of the function in the next section.

We then test which type of event occurred and, depending on the event type, we call a specific function.

Then when the program terminates we call routines to close the windows and remove the menu.

Obviously this program is only a skeleton; it calls functions that we haven't described but which perform specific actions. However, this example shows the layout of a GEM program that uses the **HiSoft C** GEM toolbox.

On your **HiSoft C** disk, there are two programs showing the handling of events called `paleochr.c` and `event.c`.

The commenting of these is un-even but you are likely to find them very useful in understanding this complex mechanism.

3.5.3 The event function

We will now describe in detail the **HiSoft C** toolbox `event` function.

This function has 6 parameters whose syntax is as follows:

```
int menu_title;
int menu_item;
int key_press;
int clickx;
int clicky;
short window[6];
int event_type;
event_type = event(&menu_title, &menu_item,
                window, &key_press, &clickx, &clicky);
```

Note that the parameters are modified by the call to the function. This is the reason for the `&` character before the parameters. If you omit one of these your program won't work and may even crash.

In general, these parameters let you indicate the type of events that you wish to wait for and, in addition, the details of the event that occurred are returned therein.

This function has one or two parameters for each type of event. If a parameter corresponding to an event is zero, this indicates that you do not wish to wait for this type of event. Otherwise this parameter must be a variable and when the function returns the variable will return the details of that event.

This function returns a whole number which indicates which event has happened. So by examining this variable you can find out if a keyboard event or a menu event has occurred.

Value	Event type
1	key press
2	mouse button click
3	menu selected
4	window manipulated

GEM also has other types of events which are not used as frequently, like timer events, which wait for a certain amount of time to pass and events that wait for the mouse to leave an area.

You can not use the `event` function to access these events; you must use the GEM functions `evnt_multi`, `evnt_timer` or `evnt_mouse` directly rather than a toolbox function.

3.5.4 Menu events

First of all, what is a menu event?

When you decide to wait for a menu event, you call the `event` function. The mouse cursor will appear on the screen and the user may 'pull-down' the menus and select an entry. At the precise moment that the mouse button is released the menu event is considered over. The `event` function returns to the calling program indicating which menu item has been selected.

The first two parameters of the `event` function are concerned with pull down menus. If you don't want to wait for events caused by menu selection then these two values must be zero. In place of `&menu_title` and `&menu_item`, you put the value 0.

Otherwise, the `event` function returns the numbers of the title and item of the menu entry that was selected with the mouse.

Remember that these numbers are identical to the values returned by the `title_menu` and `item_menu` toolbox functions when you create the menu. These two functions give unique numbers specifying the menu item. The same values are returned in the `menu_title` and `menu_item` items to inform you which item has been selected and the title of the sub-menu.

For example, if you select the Quit entry from the File menu the variable `menu_title` will contain the number of the title File and `menu_item` will return the number for Quit. These values are the same as those returned by `title_menu` and `item_menu` when the menu was created.

3.5.5 Window events

This event is the most difficult to handle as there are many different 'flavours' of window event: changing the size, closing the window, moving a slider etc...

The third parameter of the `event` function is concerned with events applied to windows. Remember that window events occur when the user changes the window in any way. For example, clicking on the full box, moving the window, or even clicking on an arrow etc.

When such an event occurs the `event` function needs to return the sort of event (changing the size, dragging a slider) and also extra details of that event, for example, when the window size is changed the new size of the window that the user has asked for.

The place where the details of the event are stored must be big enough. To do this, an array of six short integers is used. The third parameter of `event` is the name of this array.

When the function returns, the array contains the full details of the window event.

If you haven't opened a window, or you don't wish to deal with events that can happen to your window then pass zero instead of the array name.

The full list of GEM window events is as follows:

- Clicking in the close box
- Clicking in the full box
- Clicking on the arrows
- Moving a slider
- Changing the size of a window
- Moving a window without changing the size

- Re-displaying the inside of a window
- Making a window the top one

The first (number 0) element of the array indicates which event has taken place. The significance of the other elements depends on the type of event; all the elements are never used at once.

There now follows a list of all the different GEM window messages together with a description of the elements that are used:

• Clicking in the close box

element 0:	22
element 1:	the number of the window to be closed.

When the user clicks on the close box, the first element of the array has the value 22 and the second element indicates which window has been clicked on.

• Clicking in the full box.

element 0	23
element 1	the number of the window to make full size.

• Clicking on the arrows or in the grey part of the slider.

If the user clicks on a window's arrows this moves the window upwards, downwards, to the right or to the left by one line/character. If you click in the grey area of the slider the window should move by a page in the corresponding direction.

element 0 :	24
element 1 :	window number
element 2 :	action to perform:

0	page up
1	page down
2	row up
3	row down
4	page left
5	page right
6	column left
7	column right

For example, if element 2 contains the value 3, the user has clicked on the down arrow of the window.

In this case the program should scroll the window one line towards the bottom of the file.

• Moving the horizontal slider

GEM returns the new position of the slider as a value between 0 (the leftmost position) and 1000 (the rightmost).

element 0 :	25
element 1 :	window number
element 2 :	slider position between 0 and 1000.

• Moving the vertical slider

GEM returns the new position of the slider as a value between 0 (the top position) and 1000 (the bottom).

element 0 :	26
element 1 :	window number
element 2 :	slider position between 0 and 1000.

• Changing the size of a window

GEM returns the new size of the window.

element 0 :	27
element 1 :	window number
element 2 :	x co-ordinate of the top right of the window (should remain unchanged)
element 3 :	y co-ordinate of the top right of the window (should remain unchanged)
element 4 :	the new width of the window
element 5 :	the new height of the window.

• Moving a window without changing the size

The user has moved the window by dragging on the title bar. GEM returns the new window position that the user wants.

element 0 :	28
element 1 :	window number
element 2 :	new x co-ordinate of the top right of the window
element 3 :	new y co-ordinate of the top right of the window
element 4 :	the width of the window (unchanged)
element 5 :	the new height of the window (unchanged).

• Making a window the top one

The user has clicked on a window to make it the front one.

element 0 :	21 or 29
element 1 :	number of the window to become active.

• Re-displaying the inside of a window

This event occurs when a window is opened, or when it becomes larger or when an object (another window or dialog box) has deleted the interior of the window.

element 0 :	20
element 1 :	number of the window to redraw.

Note : The toolbox function `draw_box` which displays a dialog box on the screen saves the screen before drawing it and restores it afterwards. If there is only one window on the screen and it cannot change size this event can be ignored.

Obviously this list will give you something to think about. "Well, have I got to cope with all these types if I'm going to use windows ?" I am afraid so. However to help you, at least a bit, there's an example on the **HiSoft C** disks called `event.c`. It will get you started.

This concludes our discussion of by far the most complicated event type.

3.5.6 Keyboard events

The fourth parameter of the `event` function is concerned with keyboard events. Such events are simply keystrokes from the keyboard.

When you wait for a keyboard event, you are waiting for the user to press a key but it is a bit more serious than saying "Oi you, I'm waiting for a keyboard event!".

If you aren't interested in what the user types you can use zero as the fourth parameter. So if you put 0 instead of `&key_press`, keyboard events will be ignored.

Otherwise the `event` function will return a code for the key you have pressed in the `key_press` variable. The low order byte of this is the ASCII code of the key; the higher order bytes are special keyboard codes that GEM uses all the time. This lets you detect `Help` keys, `Alt` combinations etc.

If you don't understand how to use the powerful/obscure facility, don't worry; you can use the C expression `key_press % 256` to return the normal ASCII code of the key pressed where `key_press` is the variable that you passed to the `event` function. Obviously in this case you can't check for keys with no ASCII equivalent like `Undo` or the function keys.

3.5.7 Mouse events

A mouse event (also known as a button event in GEM-speak) is a click on a mouse button. With the help of the `event` function, you can wait for the user to click on the mouse.

The fifth and sixth parameters of the `event` function are concerned with mouse events. As usual, if you don't want to wait for this event then both parameters should be 0. Instead of `&clickx` and `&clicky`, put the value 0. Otherwise the `event` function returns the position of the mouse at the moment of the mouse click.

The examples below show how to call `event` for a few very simple cases. The example programs `paleochr.c` and `event.c` show how to handle all the types of event at once.

```
int menu_title;
int menu_item;
int key_press;
int clickx;
int clicky;
short window[6];
int event_type;

main
{
event_type = event(0, 0, 0, &key_press, &clickx, &clicky);
/* wait for a key press or a mouse click */

event_type = event(&menu_title, &menu_item, 0, 0, 0, 0);
/* wait for a menu to be selected */

event_type = event(0, 0, window, &key_press, &clickx,
&clicky);
/* wait for a key press, mouse click or window action */
}
```

3.6 Conclusion

This section has shown how to use GEM via the **HiSoft C** toolbox facilities. Clearly, you can access the standard GEM AES and GEM VDI functions directly. These functions are not described here. See the books in the Bibliography that document these.

The toolbox functions which have simplified life for us, are themselves written in C; we supply their source in the COMPILE folder.

Have fun reading them, and you will probably soon be happy that we have written them for you!

4 HiSoft C Library Functions

This section describes all 460 functions in the **HiSoft C** library.

We start with a one line summary of each function in the libraries to make it easy to find which function you need and then describe each function in alphabetic order.

Some of the functions are specific to the **HiSoft C** interpreter and we supply the source code to these free so that you can compile programs that use them. These functions are marked as **HiSoft C** in the reference section.

The GEMDOS library is used to access the lower levels of the ST operating system.

The GEM AES and GEM VDI libraries are used to access the appropriate higher levels of the ST operating system.

The UNIX library contains the low-level routines that are derived from the standard UNIX libraries as used with most implementations of C. We have had to make some changes to these because of differences in the underlying operating systems. To make them all exactly the same the ST would need to be running UNIX of course.The differences between the **HiSoft C** and the UNIX functions are described under the particular function.

Finally, there is the ANSI library. This contains many of the functions described in the draft ANSI standard for C and which have been implemented under UNIX and in many of the more recent C implementations. In particular it is a good idea to use the ANSI file handling functions rather than the lower-level UNIX ones as the ANSI functions are much more portable.

If you have used Lattice C 3 you will be familar with these functions apart from the **HiSoft C** ones. Of course, Lattice C 5 also includes most of these .

Other compilers may have additional functions, or they may omit some of them.

Each function is described in the same way. Its name and the library that it comes from are given in the heading.

Then the syntax and types of parameters are described followed by the usage of the function.

Finally, there may be some other sections, prefaced with an icon:

This icon is used to indicate a tricky point about a function or a common error that beginners make.

This is used to indicate a particular usage of a function or a trick that may not be obvious.

This icon is used before an actual example of the use of a function.

This icon is used to make references to other functions, other parts of the manual or the name of a keyword used by the Help system.

4.1 Library Summary

4.1.1 The HiSoft C library

Debug control functions

`trace_on`	sets trace mode
`trace_off`	disables trace mode
`var_on`	enables display of variables
`var_off`	disables display of variables
`stop`	stops execution of program

Dialog Box functions

`init_box`	creates a dialog box
`draw_box`	displays a dialog box
`text_box`	adds a string to a dialog box
`button_box`	adds a button to a dialog box
`gtext_box`	adds graphics text to a dialog box
`edit_box`	adds an editable field
`color_box`	sets the colours of a graphics text field
`readstr_box`	returns the text component of an editable field
`readbut_box`	returns the state of a button
`adr_box`	returns the address of a dialog box

Line A graphics functions

`linea0`	initialises the lineA routines
`linea1`	plots a point
`linea2`	returns the colour of a point
`linea3`	draws a line
`linea4`	draws a horizontal line
`linea5`	draws a filled rectangle
`linea6`	draws a filled polygon
`linea7`	'blits' a rectangle
`linea8`	writes a character
`linea9`	shows the mouse
`lineaa`	hides the mouse
`lineab`	modifies the mouse form
`lineac`	defines a sprite
`linead`	displays a sprite
`lineae`	copies raster form
`lineaf`	seed fill

Menu Functions

`init_menu`	creates a menu
`title_menu`	adds a title to a menu
`item_menu`	adds an item to a particular menu title
`draw_menu`	displays the menu bar
`delete_menu`	de-installs the menu bar
`enable_menu`	'greys' a menu item
`check_menu`	places a tick in front of a menu item
`select_menu`	select/de-select a menu title

Rectangle functions

`rect_intersect`	finds the intersection of two rectangles
`rect_union`	finds the union of two rectangles
`rect_point`	returns whether a point is within a rectangle
`rect_init`	initialises a rectangle structure

Resource File functions

`rs_drawalert`	draws an alert
`rs_drawobject`	draws an object or objects from an object tree
`rs_drawdial`	draws a dialog box
`rs_erasedial`	erases a dialog box
`rs_addrdial`	returns the address of a dialog box
`rs_addralert`	returns the address of a dialog box
`rs_addredit`	returns the addess of an editable string
`rs_addrbutton`	gives the address of the text of a string or button
`rs_objxywh`	returns the co-ordinates of an object
`rs_objstate`	returns whether an object is selected or not
`rs_objselect`	selects an object
`rs_objunselect`	de-selects an object

Window handling functions

`open_window`	opens a window
`close_window`	closes a window
`clear_window`	clears the work area of the window
`print_window`	displays a string in a window
`pos_window`	positions the text cursor of the window
`size_window`	returns the size of the work area of the window

Miscellaneous functions

`draw`	draws a line on the screen
`mouse`	returns the mouse position
`event`	waits for an event
`timer_value`	returns the value of the interrupt timer

4.1.2 ANSI file handling routines

Opening and Closing files

`fopen`	open a file
`fclose`	close a file
`fdopen`	open a file that has been opened using open
`freopen`	close and then open a file
`fcloseall`	close all files

Reading

`fread`	read bytes from a file
`fgetc`	read a character from a file
`getc`	read a character from the standard input
`ungetc`	put back a character so that the last one will be read again
`fgetchar`	read a character from a file
`getchar`	read a character from the standard input
`fgets`	read a string of characters from a file
`gets`	read a string of characters from the standard input
`fscanf`	formatted input from a file
`scanf`	formatted input from the standard input

Writing

`fwrite`	write bytes to a file
`fputc`	write a single character to a file
`putc`	write a single character to the standard output
`fputchar`	write a character to a file
`putchar`	write a single character to the standard output
`fputs`	write a string to a file
`puts`	write a string to the standard output
`fprintf`	formatted output to a file
`printf`	formatted output to the standard output

Standard files

`stdin`	Standard input file
`stdout`	Standard output file
`stderr`	Standard error file
`stdaux`	Serial device
`stdprn`	Printer device

Positioning within a file

`fseek`	moves to a particular byte in a file
`ftell`	returns the current position within a file
`rewind`	moves to the beginning of a file
`fflush`	writes any remaining bytes in a file's buffer
`flushall`	perform a `fflush` for all open files

Error handling

`feof`	returns whether at the end of a file
`ferror`	tests whether an error has occurred
`clrerr`	cancels an error condition
`errno`	describes the last error

Various

`setbuf`	changes the i/o buffer for a file
`setnbuf`	suppress i/o buffering for a file
`cprintf`	formatted output to the screen
`cscanf`	formatted input from the keyboard
`remove`	delete a file

4.1.3 Unix functions

Opening and Closing files

`open`	open a file
`creat`	create a file
`close`	close a file
`dup`	duplicate a file handle
`dup2`	force a file handle to refer to another file
`fileno`	return the file handle of a file that has been opened with `fopen`

Reading and Writing

`read`	read bytes from a file
`write`	write bytes to a file
`lseek`	position within a file
`tell`	return the current position of a file

File operations

`access`	read the access attributes of a file
`chmod`	change the access attributes of a file
`rename`	change the name of a file
`unlink`	delete a file

Directory operations

`getcwd`	read the current directory
`chdir`	change the current directory
`mkdir`	create a new directory
`rmdir`	delete a directory

4.1.4 ANSI Mathematical functions

Trigonometric functions

`sin`	sine
`cos`	cosine
`tan`	tangent
`acos`	inverse (arc) cosine
`asin`	inverse (arc) sine
`atan`	inverse (arc) tangent
`atan2`	special form of inverse (arc) tangent

Exponential functions

`exp`	raise to the power of e
`log`	natural logarithm (to base e)
`log10`	logarithm to base 10
`pow`	raise to the power
`sqrt`	square root
`sinh`	hyberbolic sine
`cosh`	hyberbolic cosine
`tanh`	hyberbolic tangent

Floating point manipulation routines

`floor`	nearest whole number (rounds down)
`ceil`	nearest whole number (rounds up)
`fabs`	absolute value
`fmod`	floating point modulo
`modf`	return the whole and fractional part
`ldexp`	create a float given a mantissa and exponent
`frexp`	return the mantissa and exponent of a float
`matherr`	the error that occurred with a maths function
`dqsort`	sort an array of double precision numbers

Integer functions

`abs`	absolute value
`iabs`	absolute value
`labs`	absolute value
`max`	larger of two integers
`min`	smaller of two integers
`rand`	generates a random number
`srand`	intialise the random number generator
`sqsort`	sort an array of short integers
`lqsort`	sort an array of long integers

4.1.5 String functions

Numeric/string conversions

`sscanf`	converts from a string to numeric (integer, float etc)
`sprintf`	converts from numerics to strings
`atof`	converts from ascii to float
`atoi`	converts from float to ascii
`atol`	converts from string to a long integer
`strtol`	converts from string to a long integer
`ecvt`	converts from float to string
`fcvt`	converts from float to string
`gcvt`	converts from float to string

String Copy functions

`strcpy`	copy from one string to another
`strncpy`	copy from one string to another
`strcat`	copy from one string to the end of another
`strncat`	copy from one string to the end of another

Search functions

`strtok`	splits a string into words
`strspn`	counts particular characters in a string
`strcspn`	counts all but particular characters in a string
`strchr`	searches for a character within a string

String comparison functions

`strcmp`	compare two strings
`strncmp`	compare two strings
`stricmp`	compare two strings ignoring upper/lower case
`strnicmp`	compare two strings ignoring upper/lower case

Miscellaneous string functions

`strlen`	return the length of a string
`strlwr`	upper case a string
`strupr`	lower case a string
`strrev`	reverse a string
`strtime`	create a string containing the current time
`strdate`	create a string containing the current time
`strgetfn`	create a file name from its components
`strsplfn`	split a file name into its components
`tqsort`	sort a table of strings

4.1.6 Character functions

Testing characters

isalpha	tests if a character is alphabetic
isalnum	tests if a character is numeric
islower	tests if a character is a lower case letter
isupper	tests if a character is an upper case letter
ispunct	tests if a character is a punctuation symbol
isspace	tests if a character is a space
isdigit	tests if a character is a digit
isxdigit	tests if a character is a hexadecimal digit
iscsym	tests if a character is valid in C identifiers
iscsymf	tests if a character is valid at the front of C identifiers
isprint	tests if a character is printable
isascii	tests if a character is a valid ascii character
isgraph	tests if a character is a graphics character
iscntrl	tests if a character is a control character

Modifying characters

tolower	converts to lower case
toupper	converts to upper case
toascii	converts to ascii

4.1.7 Memory functions

Memory allocation

malloc	allocate a block of memory
realloc	allocate a new block of memory
calloc	allocate a block of memory filled with zeros
free	free a block of memory

Block manipulation

memcmp	compare two blocks of memory
memchr	find a character within a block of memory
memcpy	copy a block of memory
memset	fill an area of memory with a particular value
memccpy	copy a block until a particular character is found
repmem	initialises a buffer to a given value

Program Termnation

abort	terminate the program
exit	terminate the program

4.1.8 GEMDOS functions

`gemdos`	call a GEMDOS function
`bios`	call a BIOS function
`xbios`	call an XBIOS function

Character input/output routines

`Bconin`	wait for a character from a specified device
`Bconout`	output a character to a specified device
`Bconstat`	return the input status of a specified device
`Bcostat`	return the state of an output device

Reading the keyboard

`Cconin`	wait for a character from the keyboard with echo
`Crawcin`	wait for a character from the keyboard without echo
`Crawio`	read a character from the keyboard without waiting
`Cnecin`	wait for a character from the keyboard
`Cconrs`	read a string of characters from the keyboard
`Cconis`	returns whether a character is present in the keyboard buffer
`Kbrate`	set the keyboard repeat rate
`Ikbdws`	send a string of commands to the keyboard controller
`Keytbl`	set the keyboard mapping table
`Bioskeys`	re-initialise the keyboard mapping table
`Kbdvbase`	return the address of the keyboard vector table
`Getshift`	return the state of keyboard shift keys

Screen

`Cconout`	write a character to the screen
`Cconws`	write a string to the screen
`Cursconf`	set the text cursor attributes
`Physbase`	return the memory address of the physical screen
`Logbase`	return the memory address of the logical screen
`Getrez`	return the screen resolution
`Setscreen`	change the address or resolution of the screen
`Setcolor`	set the palette colours
`Stepallette`	change the palette colours
`Vsync`	wait for the next vertical sync pluse

Serial and Midi functions

Rsconf	modify the RS232 configuration
Cauxin	wait for a character from the serial port
Cauxout	write a character on the serial port
Cauxis	return whether a character is ready to be read
Cauxos	return the serial output status
Midiws	write a string to the MIDI port
Iorec	return the address of the serial input buffer

Disk functions

Dsetdrv	set the current (default) drive
Dgetdrv	return the current drive
Dfree	return the free space on the speecified drive
Rwabs	read or write logical disk sectors
Floprd	physical read of a floppy disk
Flopwr	physical write of a floppy disk
Flopfmt	formats a floppy disk track
Flopver	physical verify of a floppy disk
Dcreate	create a new directory
Ddelete	delete a directory
Dsetpath	set the default path for a particular drive
Dsetpath	return the current path for a particular drive
Fsfirst	read the first file name that matches a given filespec
Fsnext	read the next file name that matches a given filespec
Fsetdta	set the dta address (used by Fsfirst and Fsnext)
Fgetdta	read the dta address
Drvmap	return the currently connected disk drives
Mediach	returns whether a floppy disk has been changed
Getbpb	returns information about a disk drive
Protobt	creates a boot sector image

File handling

Fcreate	create a new file
Fopen	open a file
Fclose	close a file
Fread	read from a file
Fwrite	write to a file
Fseek	position within a file
Fdelete	delete a file
Fattrib	return a file's attributes
Frename	rename a file
Fdatime	change a file's time and date stamp
Fdup	duplicate a file handle

`Fforce`	force a file handle to refer to the same file as another handle

Printing

`Setprt`	configure the printer device
`Cprnout`	write a character to the printer
`Cprnos`	return the printer output status
`Prtblk`	print a screen dump

Sound

`Giaccess`	read or write a sound chip control register
`Ongibit`	set one bit of port A of the sound chip
`Offgibit`	set one bit of port A of the sound chip
`Dosound`	execute a set of sound commands

Date and Time

`Tgetdate`	read the date
`Tsetdate`	change the date
`Tgettime`	read the current time
`Tsettime`	set the current time
`Gettime`	read the time and date
`Settime`	change the time and date

Memory allocation and application control

`Malloc`	allocate a block of memory
`Free`	free a block of memory
`Mshrink`	return an area of memory to the operating system
`Pexec`	load and execute a program
`Pterm0`	terminate the application
`Ptermres`	terminate the application without removing it from memory
`Puntaes`	reboot the machine
`Setexc`	set a vector
`Super`	enter supervisor mode
`Supexec`	execute a routine in supervisor mode

Programing the 68901

`Mfpint`	initialise a 68901 interupt
`Jenabint`	activate an interrupt
`Jdisint`	deactivate an interrupt
`Xbtimer`	initialise a timer
`Tickcal`	return the timer calibration value

Various

`Random`	return a random number
`Initmous`	initialise the mouse
`Sversion`	return the GEMDOS version number

4.1.9 GEM AES functions

Application control

appl_init	intialise application
appl_exit	deinitialise application
appl_trecord	start recording events
appl_tplay	execute a string of events
appl_write	send a message
appl_read	read a message
appl_find	find an application's id

Event Control

evnt_dclick	specify the mouse double-click speed
evnt_keybd	wait for a keyboard event
evnt_timer	wait for a timer event
evnt_button	wait for a mouse button event
evnt_mouse	wait for a mouse movement event
evnt_mesag	wait for an AES message
evnt_multi	wait for more than one event at once

Form functions

form_dial	perpare the screen for drawing a dialog box
form_do	let the user 'fill in' a form using the AES
form_alert	display an alert box
form_center	centre a form on the screen
form_error	display an error box

File Selector

fsel_input	make the GEM file selector appear

Graphics routines

graf_dragbox	move a rectangle using the mouse
graf_slidebox	move a rectanglel within another rectangle
graf_rubberbox	size a rectangle using the mouse
graf_growbox	draws an expanding box outline
graf_shrinkbox	draws a shrinking box outline
graf_movebox	draws a moving box outline without finally displaying the box
graf_watchbox	changes a box's state when the mouse enters or exits a box
graf_mouse	changes the appearance of the mouse and can be used to hide and show the mouse
graf_handle	returns the GEM VDI handle and character cell information
graf_mkstate	returns the state of the keyboard shift keys

Menu functions

`menu_bar`	displays the menu bar
`menu_text`	changes the text of a menu item
`menu_tnormal`	displays a menu title in inverse or normal
`menu_icheck`	displays a tick in front of a menu item
`menu_ienable`	a 'greys' a menu item
`menu_register`	adds a desk accessory to the desk menu

Object functions

`objc_add`	add an object to a tree
`objc_delete`	delete an object from a tree
`objc_change`	change an object's state (ob_state field)
`objc_draw`	draw an object and optionally its children
`objc_offset`	calculate the co-ordinates of a given object
`objc_find`	find an object given a co-ordinate
`objc_order`	change the order of objects within a tree
`objc_edit`	edits a character within an editable field

Resource routines

`rsrc_load`	load a resource file
`rsrc_free`	free the memory used by the resource file
`rsrc_gaddr`	finds the address of an item in a loaded resource file
`rsrc_saddr`	fixup an address within a loaded resource file
`rsrc_obfix`	convert the co-ordinates of an object to pixel co-ordinates

Clipboard functions

`scrp_read`	find the clipboard directory
`scrp_write`	set the clipboard directory

Shell routines

`shel_envrn`	read an environment string
`shel_write`	indicate the next application to be loaded
`shel_read`	find the name of the application
`shel_find`	find a given program in the current path

Window functions

`wind_create`	create a window
`wind_open`	open a window
`wind_close`	close a window
`wind_delete`	delete a window
`wind_get`	return information about a window
`wind_set`	modify a window
`wind_find`	find which window is below a given co-ordinate
`wind_update`	forbid screen modifications
`wind_calc`	calculate the size of a window's work area

4.1.10 GEM VDI functions

Workstations

`v_opnvwk`	open virtual workstation
`v_clsvwk`	close virtual workstation
`v_clrwk`	clear workstation
`vs_clip`	set clipping rectangle
`vq_extnd`	return extra information about the workstation

Graphics Attributes

`vs_color`	set the RGB components of a colour
`vq_color`	return the RGB components of a colour
`v_get_pixel`	return the colour of a point
`vq_cellarray`	return the colours of a cell array
`vqin_mode`	select a type of input
`vswr_mode`	select writing mode

Font functions

`vst_load_fonts`	load all the font definitions
`vst_unload_fonts`	remove the font definitions
`vst_font`	select a particular font
`vqt_name`	find the name of a given font
`vqt_fontinfo`	return information about the current font

Graphics text

`v_gtext`	display graphics text
`v_justified`	display justified text
`vst_height`	set the character height
`vst_point`	set the character height in points
`vst_rotation`	set the angle of rotation of characters
`vst_color`	set the colour of graphics text
`vst_effects`	set the text graphic effects
`vst_alignment`	set the vertical text alignment
`vqt_attributes`	return the current text attributes
`vqt_extent`	return the size of a graphics text string
`vqt_width`	return the size of a single character

Area fill functions

`v_fillarea`	draw a filled polygon
`v_contourfill`	seed fill
`vsf_interior`	select the type of fill
`vsf_perimeter`	set whether areas are surrounded by borders
`vsf_style`	select the type of fill
`vsf_color`	select the colour of fills
`vsf_udpat`	set user defined fill pattern
`vqf_attributes`	return the fill attributes

Line drawing functions

`v_pline`	draw one or more lines
`vsl_type`	select the line type
`vsl_udsty`	set a user-defined line type
`vsl_width`	set the line width
`vsl_color`	set the line colour
`vsl_ends`	set how the ends of lines are to be drawn
`vql_attributes`	return the line drawing attributes

Marker functions

`v_pmarker`	draw a set of markers
`vsm_type`	set the type of polymarker
`vsm_height`	set the height of polymarkers
`vsm_color`	set the colour of polymarkers
`vqm_attributes`	return the current polymarker attributes

Rectangle drawing functions

`v_cellarray`	draw an array of rectangles
`vr_recfl`	draw a filled rectangle without a border
`v_bar`	draw a filled rectangle with a border
`v_rbox`	draw an unfilled rectangle with a rounded border
`v_rfbox`	draw a filled rectangle with a rounded border

Circular objects

`v_arc`	draw a circular arc
`v_pieslice`	draw a pie slice
`v_circle`	draw a circle
`v_ellarc`	draw an elliptical arc
`v_ellpie`	draw an elliptical pie slice
`v_ellipse`	draw an ellipse

Alpha mode routines

`v_enter_cur`	enter alpha mode
`v_exit_cur`	exit alpha mode
`v_curtext`	draw alpha text
`v_rvon`	inverse video on
`v_rvoff`	inverse video off
`v_curright`	cursor right
`v_curleft`	cursor left
`v_curup`	cursor up
`v_curdown`	cursor down
`v_curhome`	home the cursor to the top right
`v_eeol`	erase to end of line
`v_eeos`	erase to end of screen
`vq_chcells`	return the size of the screen
`vs_curaddress`	position the cursor
`vq_curaddress`	return the current cursor position

Mouse control functions

`v_show_c`	display the mouse cursor
`v_hide_c`	remove the mouse cursor
`v_dspcur`	change the mouse position
`v_rmcur`	make the mouse disappear
`vsc_form`	define a new mouse form
`vq_key_s`	return the keyboard shift key state
`vq_mouse`	return the mouse position and button state

Screen Raster functions

`vro_cpyfm`	copy a screen block
`vrt_cpyfm`	copy a monochrome screen block
`vr_trnfm`	copy a monochrone screen block (device dependent)

Modifying vectors

`vex_timv`	timer vector
`vex_curv`	mouse cursor vector
`vex_butv`	mouse button vector
`vex_motv`	mouse movement vector

ABORT ANSI

```
abort();
```

This function stops a running program immediately.

exit.

ABS ANSI

```
ret  =  abs(val);
int  ret,val;
```

This function returns the absolute value of the value passed as a parameter. Both values are of type `int`.

iabs, labs, fabs.

ACCESS UNIX

```
ret  =  access(name,mode);
int  ret,mode;
char  *name;
```

Tests if a file can be acessed for read and/or write depending on the value of `mode`.

If `mode = 0`, test the existence of the file

If `mode = 2`, test if write access to the file is possible.

The other UNIX modes are ignored.

The value returned is 0 if access is possible and -1 if not. In the latter case the variable `errno` contains the corresponding error number.

chmod, errno.

ACOS — ANSI

```
ret  =  acos(val);
double  ret,val;
```

Calculates the arc-cosine of the value (between -1 and +1) passed as a parameter. The value returned is between 0 and pi.

Both values (the parameter and the result) are double reals.

If the parameter is not between -1 and +1 the variable `errno` will indicate the error condition.

asin, atan, atan2, errno.

ADR_BOX — HiSoft C

```
#include  <gemlib.h>
p  =  adr_box(box_no);
int  box_no;
OBJECT  *p;
```

This function returns the address of a dialog box. `box_no` is the number of box whose address you wish to find. This is the same as the value that is returned by the function `init_box`.

The value returned is a pointer to the tree structure of type `OBJECT`.

Section 3.3, init_box, Help command (adr_box).

APPL_INIT — GEM AES

```
appl_id=  appl_init();
int  appl_id;
```

Initialise the application and the GEM AES.

The value returned is the application's identifier (or appl_id) that is used in the application library functions.

With **HiSoft C**, it is not necessary to call this function if you wish to use GEM.

If the value returned is -1 this indicates an error.

APPL_EXIT GEM AES

```
ret  =  appl_exit();
int  ret;
```

This function de-initialises the application as far as the AES is concerned.

If `ret = 0` there has been an error.

When using **HiSoft C**, there is no need to call this function.

APPL_FIND GEM AES

```
appl_id  =  appl_init(name);
int  appl_id;
char  *name;
```

Finds the application identifier of the named application (normally a desk accessory). The name must be at least eight characters, padded with blanks if necessary.

APPL_READ GEM AES

```
ret  =  appl_read(appl_id,length,buffer);
int  appl_id,length;
char  *buffer;
```

Reads a message (of the specified `length`) that has been sent by another application. The message is stored in the given `buffer`. The application identifer of the program whose message pipe is being read must be supplied as the `appl_id` parameter.

The value returned is 0 if an error has occurred.

APPL_TPLAY GEM AES

```
appl_tplay(address,n,speed);
int  n,speed;
char  *address;
```

Executes a set of `n` events that have been stored by the `appl_trecord` function at `address` at the `speed` given (100 is normal speed).

APPL_TRECORD GEM AES

```
appl_trecord(address,n);
int  n;
char  *address;
```

Stores a set of `n` events at the address given by the `address` parameter.

APPL_WRITE GEM AES

```
ret  =  appl_write(appl_id,length,buffer);
int  appl_id,length
char  *buffer;
```

Writes a message (of the length specified from `buffer`) so that other applications may read them. The application identifier of the program that is to read the message must be passed as the `appl_id` parameter.

The message can be read using `appl_read`.

0 is returned if there is an error.

appl_read.

ASIN ANSI

```
ret  =  asin(val);
double  ret,val;
```

Calculates the arc-sine of the value (between -1 and +1) passed as a parameter. The value returned is between -pi/2 and +pi/2.

Both values (parameter and return value) are double reals.

If the value passed is not between -1 and +1 then the variable `errno` will indicate that an error has occurred.

acos, atan, atan2, errno.

ATAN ANSI

```
ret  = atan(val);
double  ret,val;
```

Calculates the arc-tangent of the value passed as a parameter. The parameter and the value returned are double reals.

The result (in radians) is between -pi/2 and +pi/2 inclusive.

atan2.

Find the value of pi

```
void  main()
{
      printf("%10.10f",  atan(1.)*4);
}
```

ATAN2 ANSI

```
ret  =  atan2(y,x);
double  x,y,ret;
```

Calculates the arc-tangent of `y` divided by `x`.

The value returned corresponds to the angle (expressed in radians between -pi and +pi) formed by the positive x-axis and the vector from (0,0) to (`x,y`).

If `x` is zero this function returns `+pi/2` or `-pi/2` depending on the value of `y`.

It is an error to call this function with both parameters 0 (`errno` will indicate this).

atan, asin, acos, errno.

ATOF ANSI

```
x  =  atof(string);
double  x;
char  *string;
```

This function converts a string of characters to a double precision floating point number.

The string may contain white space, a plus or minus sign followed by a standard scientific format number.

The conversion stops at the first inappropriate character.

atoi, atol, sscanf.

ATOI ANSI

```
i  =  atoi(string);
int  i;
char  *string;
```

This function converts a string of characters to an integer number.

The string may contain white space, a + or - sign followed by a string of digits.

The conversion stops at the first inappropriate character.

atof, atol, sscanf.

ATOL ANSI

```
i = atol(string);
long i;
char *string;
```

This function converts a string of characters to a long integer and it works in the same way as `atoi`.

⚠ atof, sscanf.

BCONIN GEMDOS

```
character_code = Bconin(peripheral);
int peripheral, character_code;
```

Waits for a character to be input from the specified peripheral:

0	printer (!)
1	serial port
2	screen/keyboard
3	MIDI
4	keyboard processor
5	screen

The character code returned consists of two bytes.

The low byte is the ASCII code and the high byte the keyboard scan code.

BCONOUT GEMDOS

```
Bconout(peripheral, character);
int peripheral, character;
```

Writes a character to the given peripheral (see the `Bconin` function).

BCONSTAT GEMDOS

```
status = Bconstat(peripheral);
int status,peripheral;
```

Returns the input status of the peripheral given as a parameter (see the `Bconin` function).

The returned value is -1 if the device is ready, 0 otherwise.

BCOSTAT GEMDOS

```
status  =  Bcostat(peripheral);
int  status,peripheral;
```

Returns the output status of the peripheral given as a parameter (see the `Bconin` function).

The returned value is -1 if the device is ready, 0 otherwise.

BIOS GEMDOS

```
ret  =  bios(no,  arg1,  arg2...);
int  no;
long  ret,arg1,arg2,...
```

Executes a BIOS function using TRAP #13.

`no` is the number of the function.

`ret` is the value returned by the function.

`arg1`, `arg2`... are the parameters for the particular function.

xbios, gemdos.

BIOSKEYS GEMDOS

```
Bioskeys();
```

Re-initialises the standard BIOS key table.

Keytbl.

BUTTON_BOX HiSoft C

```
button_no = button_box(box_no, x, y, button_name, state);
int  button_no,box_no,x,y,state;
char  *button_name;
```

Adds a button to a dialog box that has been initialised by `init_box`.

`box_no` is the number of the dialog box to which the button is to be added.

`x` and `y` are the co-ordinates (in characters) of the position of the button within the box.

`button_name` is the string of characters giving the text of the button.

`state` gives the state of the button (the `ob_state` value).

The value returned is the index of the button within the box.

⚠ init_box, **Section 3.3.5**, Help command (button_b).

CALLOC ANSI

```
adr = calloc(no_elements, element_size);
char *adr;
long  no_elements, element_size;
```

Allocates a block of memory of size `no_elements * element_size`.

The block is initialised to zero.

The value returned is a pointer to the block of memory or 0 if the block cannot be allocated.

The memory is allocated from the system memory whose size is fixed when the interpreter is loaded. It is possible to change this value if you have enough memory.

⚠ **Section 1.4.13**, Malloc, Mfree, malloc, free, realloc.

CAUXIN GEMDOS

```
character = Cauxin();
int  character;
```

Reads a character from the serial port.

CAUXIS GEMDOS

```
status = Cauxis();
int  status;
```

Returns whether a character has been received from the serial port (`status=-1`) or not (`status=0`).

CAUXOS GEMDOS

```
status  =  Cauxos();
int  status;
```

Returns whether the serial port is ready for output (`status=-1`) or not (`status=0`).

CAUXOUT GEMDOS

```
Cauxout(character);
int  character;
```

Write a character to the serial port.

CCONIN GEMDOS

```
character  =  Cconin();
int  character;
```

Wait for a key-press and echo it to the screen.

Returns the character code read in the same form as that used by `Bconin`.

CCONIS GEMDOS

```
status  =  Cconis();
int  status;
```

Returns whether a character has been typed on the keyboard (`status=-1`) or not (`status=0`).

CCONOUT GEMDOS

```
Cconout(character);
int  character;
```

Write a character to the screen.

CCONRS GEMDOS

```
Cconrs(string);
char  *string;
```

Read a string of characters from the keyboard.

On entry, the first byte in the string must contain the maximum number of characters to read. On exit, the second byte contains the number of characters actually read. The actual characters are stored starting at the third byte.

CCONWS GEMDOS

```
Cconws(string);
char  *string;
```

Writes a string of characters to the screen.

CEIL ANSI

```
ret  =  ceil(val);
double  ret,val;
```

Returns the whole number larger or equal to the value of the argument.

The value returned is a double real with a zero fractional part; not an integer.

floor

CHDIR UNIX

```
ret  =  chdir(path);
int  ret;
char  *path;
```

Changes the current directory to the given `path`.

This function returns zero if there was no error; otherwise the type of error can be found in the variable `errno`.

Dsetpath, mkdir, rmdir, getcwd, errno.

CHECK_MENU HiSoft C

```
state  =  check_menu(entry);
int  state,entry;
```

Makes a tick in front of a menu item appear or disappear. The menu must have been created with the function `init_menu`.

`entry` is the number of the menu entry as returned by `item_menu`.

The value returned is the new state of the menu entry (1 if ticked, 0 if not).

init_menu, item_menu, **Section 3.4.6**, Help command (check_me).

CHMOD UNIX

```
#include  <fcntl.h>
ret  =  chmod(name,mode);
int  ret,mode;
char  *name;
```

Changes the protection status of the file called `name`.

Unlike UNIX, only the write status can be changed.

If `mode = S_IREAD`, the file may only be read.

If `mode = S_IWRITE`, the file may only be written. These two values are defined in the file fcntl.h.

On return, `ret=0` if the operation was successful; otherwise the type of error can be found in the `errno` variable.

access, errno.

CLRERR ANSI

```
#include  <stdio.h>
clrerr(fp);
FILE  *fp;
```

Resets the error condition on the given stream.

This function is called `clearerr` under UNIX system V so be careful when porting programs.

fopen, feof, ferror.

CLEAR_WINDOW HiSoft C

```
clear_window(window_no);
int window_no;
```

Clears the interior of a window opened with the **HiSoft C** function `open_window`. The number of the window to clear must be passed as a parameter.

open_window, **Section 3.2.8**, Help command (clear_wi).

CLOSE UNIX

```
ret = close(fhandle);
int fhandle, ret;
```

Close a file opened using `open`.

`fhandle` is the handle of the file to close as returned by the `open` function.

The value returned by `close` is zero if successful otherwise `errno` indicates which error has occurred.

open, fopen, fclose, errno.

CLOSE_WINDOW HiSoft C

```
close_window(window_no);
int window_no;
```

Closes a window that was opened by the **HiSoft C** function `open_window`.

`window_no` is the window number as returned by the `open_window` function.

open_window, **Section 3.2.4**, Help command (close_wi).

CNECIN GEMDOS

```
character = Cnecin();
int character;
```

Read a character from the keyboard.

COLOR_BOX HiSoft C

```
color_box(box_no,  object_no,  bord,text,back);
```

This function enables you to change the colour of graphics text of a dialog box created by the **HiSoft C** function `init_box`.

`box_no` is the number of the dialog box created by `init_box`.

`object_no` is the object number as returned by `gtext_box`.

`bord`, `text`, and `back` are the colours of the border, the text and the background respectively.

⚠ init_box, gtext_box, **Section 3.3.12**, Help command (color_bo).

COS ANSI

```
ret  =  cos(val);
double  ret,val;
```

Calculates the cosine of the angle (in radians) that is passed as a parameter. The parameter and the result are both double reals.

⚠ sin, tan.

COSH ANSI

```
ret  =  cosh(val);
double  ret,val;
```

Calculates the hyperbolic cosine of its parameter. The parameter and the result are both double reals.

If the argument is too big for the result to be in range the `errno` variable will indicate this error.

⚠ sinh, tanh, errno.

CPRINTF ANSI

```
length = cprintf(format, arg1, arg2,...);
int length;
char *format;
???? arg1,arg2,...
```

This function writes formatted text. ???? indicates that the parameters may be of different types.

The characters are sent direct to the screen, unlike printf where the characters are sent to the file stdout. This function avoids the filing system as used by printf.

Apart from this, this function behaves exactly like printf.

See the printf function for the description of the parameters.

CPRNOS GEMDOS

```
status = Cprnos();
int status;
```

Returns whether the printer is ready (status=-1) or busy (status=0).

CPRNOUT GEMDOS

```
status = Cprnout(character);
int status, character;
```

Writes a character to the printer. The return value indicates that the character has been printed (status=-1) or that the printer is busy (status=0).

CRAWCIN GEMDOS

```
character = Crawcin();
int character;
```

Waits for a character from the keyboard. The full scan code is returned (See Bconin).

CRAWIO GEMDOS

```
character = Crawio(parameter);
int character, parameter;
```

Writes a character to the screen with the ASCII code passed as `parameter`.

Or... reads a character from the keyboard if `parameter` is `-1`. In this case the value returned is the scan code (see `Bconin`) or 0 if there has been no key pressed. The function returns immediately if no character has been typed.

CREAT UNIX

```
#include <fcntl.h>
fhandle = creat(name,mode);
int fhandle,mode;
char *name;
```

This function creates and opens for write a new file with the given name. If the file already exists it is deleted.

The file is always opened for write.

The access privileges for the file are fixed by the value of `mode`:

`S_IREAD`	the file is read only
`S_IREAD|S_IWRITE` or 0	both read and write is possible

`fhandle` is the file handle associated with the created file and is the return value of the function.

If the value returned is `-1`, the file could not be created. The `errno` variable will indicate the type of error.

Fcreate, fopen, chmod.

CSCANF ANSI

```
n = cscanf(format, arg1, arg2, ...);
int n;
char *format;
???? arg1,arg2;
```

This function is equivalent to `scanf` except that the characters are read directly from the keyboard rather than via the file `stdin`.

`????` indicates that the parameters may be of different types.

See `scanf` for a description of the parameters.

CURSCONF GEMDOS

```
ret  =  Cursconf(period,  attribute);
int  ret,  period,  attribute;
```

Sets the text cursor attributes depending on the value of attribute:

0	the cursor is invisible
1	the cursor is displayed
2	flashing cursor
3	non-flashing cursor
4	set the flash period
5	the function returns the current flash period

DCREAE GEMDOS

```
ret  =  Dcreate(directory_name);
int  ret;
char  *directory_name;
```

Creates a directory whose name is passed as a parameter.

The value returned is zero if the operation was successful.

DDELETE GEMDOS

```
ret  =  Ddelete(directory_name);
int  ret;
char  *directory_name;
```

Deletes the directory whose name is passed as a parameter.

The value returned is zero if the operation was successful.

DELETE_MENU HiSoft C

```
ret  = delete_menu();
int  ret;
```

Deletes a menu that has been created with `init_menu` and drawn with `draw_menu`.

The value returned by this function is 1 if the operation was successful and 0 if not.

init_menu, draw_menu, **Section 3.4.5**, Help command (delete_m).

DFREE GEMDOS

```
ret = Dfree(buffer, disk_no);
int ret, disk_no;
long buffer[4];
```

This function returns information about the disk given by `disk_no`. The numbers used are

0	the current drive
1	drive A
2	drive B
3	drive C

etc.

The function stores the information in the array `buffer` as follows

`buffer[0]`	the number of free clusters on the disk
`buffer[1]`	the total number of clusters on the disk (351 or 711 for floppies)
`buffer[2]`	the sector size in bytes (normally 512)
`buffer[3]`	the number of sectors per cluster (normally 2)

The value returned by the function is 0 if the operation was successful.

DGETDRV GEMDOS

```
drive_no = Dgetdrv();
int drive_no;
```

Returns the number of the current disk drive (0=A, 1=B, 2=C, etc.)

DGETPATH GEMDOS

```
ret = Dgetpath(path_name, drive_no);
int drive_no, ret;
char *path_name;
```

Returns the default path name (in `path_name`) for the specified drive according to the value of `drive_no` (0=current disk, 1=A, 2=B, 3=C etc.).

The value returned is zero if the operation was successful.

DOSOUND GEMDOS

```
Dosound(command_string);
char  *command_string;
```

Executes a set of sound commands passes as a string of characters to the function as a parameter.

A command consists of a command byte followed by optional parameters that depend on the command.

Commands 0 to 15 have a parameter that is to be written to one of the 16 sound chip registers (command 0 for register 0, command 1 for register 1 etc.).

Command 128 has a byte argument that is written to a temporary register.

Command 129 is a form of loop statement and has three byte arguments. The first is the register to use (the temporary register is initially assigned to this register). The second argument is the increment and the third the termination value. The increment is added to the appropriate register until the termination value is reached. How frequently these assignments are executed is set by the commands below.

Commands 130 to 255 have one argument. If this is zero then the sound is terminated; otherwise the argument is taken as how frequently sound commands are to be executed in fiftieths of a second.

DQSORT UNIX

```
dqsort(arr,n);
double  *arr;
int  n;
```

Sorts an array of n double precision floating point numbers into ascending order.

⚠ lqsort, sqsort, tqsort.

DRAW HiSoft C

```
draw(x1,y1,x2,y2);
int  x1,y1,x2,y2;
```

Draws a line between the two points (x1, y1) and (x2, y2).

This function does not return a value.

⚠ v_pline, **Section 3.2.10**, Help command (draw).

DRAW_BOX HiSoft C

```
object_no  =  draw_box(box_no);
int  object_no,  box_no;
```

Draws a dialog box created by `init_box`.

⚠ **Section 3.3.6**, Help command (draw_box).

DRAW_MENU HiSoft C

```
ret  =  draw_menu();
int  ret;
```

Makes a menu bar created using `init_menu` appear. It is then possible to click on menu items.

The value returned by this function is 1 if all went well and 0 otherwise.

⚠ delete_menu, init_menu, **Section 3.4.5**, Help command (draw_men).

DRVMAP GEMDOS

```
active_drives  =  drvmap();
int  active_drives;
```

Returns which disk drives are present in `active_drives`.

The value returned has bit 0 set if drive A exists, bit 1 set if drive B exists, etc.

DSETDRV GEMDOS

```
active_drives  =  Dsetdrv(drive_no);
int  drive_no,  active_drives;
```

Set the default disk drive (`drive_no`= 0 for drive A, 1 for drive B etc.).

The value returned indicates which drives are active (see `drvmap`).

DSETPATH GEMDOS

```
ret  =  Dsetpath(path_name);
int  ret;
char  *path_name;
```

Sets the default directory for the default drive to be `path_name`.

The value returned is zero if the operation was successful.

DUP UNIX

```
new_handle  =  dup(handle);
int  new_handle,handle;
```

Duplicates a file handle. The new file handle (returned by the function) is associated with the same file as the original one.

If the duplication isn't possible -1 is returned and `errno` contains the reason for the error.

Fdup, Fforce, fdopen, errno.

See std.

DUP2 UNIX

```
ret  =  dup2(new_handle,handle);
int  new_handle,handle;
```

Forces the file `handle` to point to the same file as `new_handle`.

If an error occurs -1 is returned and `errno` indicates which error occurred.

Fdup, Fforce, fdopen, errno.

See std.

ECVT UNIX

```
p  =  ecvt(a,prec,decpt,sign);
int  prec,*sign,*decpt;
double  a;
char  *p;
```

Converts the floating point number `a` to a string of characters consisting only of digits.

`prec` is the number of significant digits desired.

On return, `decpt` indicates where the decimal point would appear from the start of the string. `sign` contains zero if the number `a` was positive or zero. `p` points to the string of characters.

EDIT_BOX HiSoft C

```
object_no  =  edit_box(box_no,  x,y,  text,  template,
legal_chars);
int  object_no,  box_no,  x,  y;
char  *text,  *template,  *legal_char;
```

Adds an editable text field to a dialog box created by `init_box`.

`box_no` indicates the dialog box to which the editable text is added.

`x` and `y` are the character co-ordinates of the text relative to the top left corner of the box.

`text` indicates the text to be displayed. `template` describes the format of the field. `legal_chars` describes which characters may be entered. `object_no` is the index of the object within the box.

init_box, **Section 3.3.8**, Help command (edit_box).

ENABLE_MENU HiSoft C

```
status = enable_menu(entry_no);
int status, entry_no;
```

Makes a menu appear in grey or makes it appear as normal if it was already grey.

The menu must be created using the `init_menu` function.

`entry_no` is the index of the menu entry to change as returned by `item_menu`.

The returned value is the new state of the menu: 1 if grey 0 if not.

init_menu, item_menu, **Section 3.4.6**, Help command (enable_m).

ERRNO UNIX

```
#include <error.h>
errno;
```

`errno` is an integer variable defined by **HiSoft C**. It is initialised to zero; it is set to a non-zero value following an error.

When a system function fails (generally returning -1), the reason for the error is stored in `errno`. This variable is not reset to zero when a function terminates correctly, thus it should only be checked when an error has occurred.

HiSoft C does not use all the UNIX error codes. In the following list of all the codes, those that are not used by **HiSoft C** are prefixed with an asterisk. The names of the errors (following the UNIX standard) are defined in the file error.h.

1	`*EPERM`	The user doesn't have permission to access this file.
2	`NOENT`	The file name specified does not exist.
3	`*ESRCH`	Process does not exist.
4	`EINTR`	The function has been interrupted by an event.
5	`EIO`	I/O error during a read or write. This error is not detected until the following call to the function.
6	`ENXIO`	Non-existent peripheral or not working (no disk in drive).
7	`E2BIG`	An argument to the function is too large.
8	`*ENOEXEC`	Error in the format of an executable file.
9	`EBADF`	Invalid file handle. This error is given if the file is not open, or a read is attempted on a write-only file or vice versa.

10	*ECHILD	No child process.
11	*EAGAIN	No more processes available.
12	ENOMEM	The program is asking for more memory than is available.
13	EACCES	Attempt to access a file in a way that does not correspond to the access privileges of the file. For example, trying to open a read-only file for write.
14	EFAULT	Memory access error during a system call (access to an illegal address) whilst trying to access its parameters.
15	ENOTBLK	File name used instead of a device identifier.
16	EBUSY	Device busy.
17	EEXIST	Attempt to create a file which already exists.
18	*EXDEV	Attempt to mount a volume that is already present on another device.
19	NODEV	Trying to execute a system function that is inappropriate for this device (reading from a write-only device).
20	ENOTDIR	Using an invalid name when a directory is required. For example, the path name required by the `chdir` command.
21	EISDIR	Trying to write to a directory.
22	EINVAL	Invalid function argument. For example, trying to read or write a file after `lseek` has returned a negative value, or a bad argument to a mathematical function.
23	ENFILE	The table of open files is full. No more files may be opened.
24	EMFILE	A process may not open more than 20 files simultaneously.
25	*ENOTTY	Not a terminal.
26	ETXTBUSY	Opening a file that is already opened.
27	*EFBIG	File too long.
28	ENOSPC	Device full (no disk space).
29	ESPIPE	Call to a seek function for a device that only supports sequential access.
30	EROFS	Attempt to write or modify for a device which only supports read access.
31	*EMLINK	Too many links to a file.
32	*EPIPE	Writing to a pipe without a process to read it.
33	EDOM	The argument of a mathematical function is outside its defined domain.
34	ERANGE	The value calculated by a function is too big to be represented by the machine.

EVENT — HiSoft C

```
event_type = event(&menu_title, &menu_entry,
        window, &key, &xclick, &yclick);
int  menu_title, menu_entry;
int  key, xclick, yclick;
int  event_type;
short  window[6];
```

This function waits for an event as described in **Section 3.5.3**.

⚠ evnt_multi, **Section 3.5**, Help command (event).

EVNT_BUTTON — GEM AES

```
num  =  evnt_button(no_clicks, mask, state_required,
     &x, &y, &button_state, &keyboard_state);
int  num,no_clicks,mask,state_required;
short  x, y, button_state, keyboard_state;
```

Wait for a given mouse button state.

`no_clicks` is the number of clicks to wait for.

`mask` indicates which buttons to test (1=left, 2=right, 3=both).

`state_required` indicates if the function should return when the button is pressed (0) or when it is released (1).

`num` is the number of clicks that actually occurred.

`x` and `y` are the mouse co-ordinates when the mouse was clicked,

`button_state` indicates which mouse buttons were pressed when the button was pressed or released.

`keyboard_state` indicates the state of the right shift (bit 0), left shift (bit 1), Control (bit 2) and Alternate (bit 3) keys. The corresponding bit is set if the given key is down.

EVNT_DCLICK GEM AES

```
interval = evnt_dclick(new_interval, set);
int interval, new_interval, set;
```

Sets or returns the double-click speed of the mouse.

If `set` is 1 then `new_interval` is used to set the double-click speed between 0 (450 ms) and 4 (165 ms).

If `set` is 0 then the current speed is being requested.

In either case, `interval` is returned as the new double-click speed.

EVNT_KEYBD GEM AES

```
key_code = evnt_keybd();
int key_code;
```

This function waits for a key to pressed on the keyboard. It returns a 16 bit code. The bottom 8 bits are the ASCII code for the key.

EVNT_MESAG GEM AES

```
evnt_mesag(buffer);
short buffer[8];
```

Wait for a window or menu event.

The description of the message is contained in the array `buffer`.

EVNT_MOUSE GEM AES

```
evnt_mouse(sort, rx,ry,rw,rh,
    &x, &y, &button_state, &keyboard_state);
int sort,rx,ry,rw,rh
short x, y, button_state, keyboard_state;
```

Wait for the mouse to enter or leave the specified rectangle.

`sort` indicates whether the function should return when the mouse enters (0) or leaves (1) the rectangle.

`rx,ry,rw,rh` give the co-ordinates of the rectangle.

`button_state` indicates which mouse buttons were pressed when the mouse moved across the rectangle border.

`keyboard_state` indicates the state of the right shift (bit 0), left shift (bit 1), Control (bit 2) and Alternate (bit 3) keys. The corresponding bit is set if the given key is down.

EVNT_MULTI — GEM AES

```
which_events = evnt_multi(event_types,
      no_clicks, mask, state_required,     /* evnt_button */
      sort1,rx1,ry1,rw1,rh1,               /* evnt_mouse */
      sort2,rx2,ry2,rw2,rh2,               /* evnt_mouse */
      buffer,                              /* evnt_mesag */
      time1,time2,                         /* evnt_timer */
      &x,&y,
      &button_state, &keyboard_state,
      &key_code,
      &num);
```

Wait for one or more events.

`event_types` indicates the types of event to wait for: 1=keybd, 2=button, 4=mouse1, 8=mouse2, 16=mesag, 32=timer. To wait for more than one event, or (|) them together.

The other parameters are the same as described under the functions `evnt_keybd`, `evnt_mouse`, `evnt_timer`, `evnt_mesag` and `evnt_button`.

The value returned is the event or events that occurred.

event.

EVNT_TIMER — GEM AES

```
evnt_timer(time1,time2);
unsigned int time1, time2;
```

Wait for a time interval. The interval is given by `time1+65536*time2` milliseconds. `temp1` and `temp2` must be less than 65536.

EXIT — ANSI

```
exit(return_code);
int return_code;
```

This function terminates the current program.

The parameter `return_code` is the code returned to the calling program; this is ignored by **HiSoft C**.

abort, stop.

EXP ANSI

```
ret  =  exp(val);
double  ret, val;
```

This function returns e to the power `val`.

The parameter and the result are both double reals.

If the parameter is too large, the variable `errno` will indicate the reason for the error.

log, log10, pow, errno.

FATTRIB GEMDOS

```
ret  =  Fattrib(file_name,  set,  attrs);
```

Reads (`set=0`) or sets (`set=1`) the attributes of a file.

The possible attributes (which may be combined) are

0	the current drive
2	file is hidden
4	file is system
8	user file
16	file is a directory
32	file has been written and closed

FABS ANSI

```
ret  =  fabs(val);
double  ret,val;
```

This function returns the absolute value of the function.

The parameter and the result are both double reals.

iabs, labs, abs.

FCLOSE ANSI

```
#include  <stdio.h>
ret  =  fclose(fp);
int  ret;
FILE  *fp;
```

Close a file that has been opened with `fopen`.

The file to close is given via the file pointer `fp`.

If the value returned is not zero, then an error has occurred during the close and the `errno` variable may be inspected to find out why.

fcloseall, fopen, open, close, errno.

FCLOSE GEMDOS

```
ret  =  Fclose(fhandle);
int ret,  fhandle;
```

Close a file opened with `Fopen`.

The value returned is zero if the file was closed correctly.

FCLOSEALL ANSI

```
#include  <stdio.h>
no  =  fcloseall(fp);
int  no;
FILE  *fp;
```

Close all the files that have been opened with `fopen`.

The returned value indicates how many files have been closed. If this value is -1 then an error has occurred whilst closing a file and `errno` may be inspected to find out why.

All the files are closed even `stdin`, `stdout` etc. After a call to this function all the standard file input/output routines won't work unless you explicitly re-open them.

fclose, fopen, open, close, errno.

FCREATE GEMDOS

```
file_handle = Fcreate(file_name, attributes);
int file_handle, attributes;
char *file_name;
```

Creates and opens a file called `file_name` with the given attributes. See `Fattrib`.

If the file already exists then it is deleted.

The function returns the file handle for use when writing and closing the file.

FCVT UNIX

```
p = fcvt(a, n, &decpt, &sign);
int n,sign,decpt;
double a;
char *p;
```

This function is identical to `ecvt` except that `n` indicates the required number of digits *after the decimal point.*

ecvt.

FDATIME GEMDOS

```
Fdatime(date_time, file_handle, set);
int file_handle, set;
char *date_time;
```

Sets (`set`=0) or returns (`set`=1) the file's creation date and time.

The date and time are returned in the buffer pointed to by `date_time` (see `Tgetdate` for the format).

FDELETE GEMDOS

```
ret = Fdelete(file_name);
int ret;
char *file_name;
```

Deletes the file whose name is passed as a parameter.

The value returned is 0 if the operation was successful.

FDOPEN ANSI

```
#include  <stdio.h>
fp  =  fdopen(file_handle,  mode);
int  file_handle;
FILE  *fp;
char  *mode;
```

This function enables you to use the ANSI file functions (`ferror`, `fprintf`, `fgets`, etc) with a file that has already been opened with the UNIX `open` function.

It is therefore a transformation from a UNIX file handle to a ANSI file pointer.

`file_handle` is the file handle of a file that has been opened using `open`.

`mode` gives the type of access for the file. Its value is the same as that described under `fopen`.

`fp` is the returned value; this is the new file pointer for the same file.

If an error occurs the value returned is 0. See `errno` for which error.

open, fopen, fileno.

see std.

FDUP GEMDOS

```
new_handle  =  Fdup(file_handle);
int  new_handle,file_handle;
```

Duplicates a file handle. `file_handle` and `new_handle` will both use the same file.

This function is very useful for re-directing the standard input/output files.

dup.

FEOF ANSI

```
#include  <stdio.h>
ret  =  feof(fp);
int  ret;
FILE  *fp;
```

Tests whether the a file given by the file pointer `fp` is at the end of the file.

This function returns the value 0 if the end of the file has not been reached.

The file concerned must be opened using `fopen`.

ferror.

FERROR ANSI

```
#include  <stdio.h>
ret  =  ferror(fp);
int  ret;
FILE  *fp;
```

Tests whether an error has occurred during i/o to the file `fp`.

This function returns 0 if an error hasn't occurred on this file.

The file must have been opened using `fopen`.

feof.

FFLUSH ANSI

```
#include  <stdio.h>
ret =  fflush(fp);
int  ret;
FILE  *fp;
```

Empties the output buffer for the file given by `fp`. If the buffer isn't already empty the information in it is written to disk.

If the operation is successful 0 is returned. Otherwise the value `EOF` is returned; for example, this will happen if the file is not open for write. In this case `errno` will indicate which error has occurred.

fflushall, fclose, errno.

FFLUSHALL — ANSI

```
#include <stdio.h>
no = fflushall();
int no;
```

Flushes the output buffer for all files opened with `fopen`.

If the operation was successful the number of files closed is returned.

Otherwise the value `EOF` is returned. This will occur if one of the files is not open for write. The variable `errno` can be examined to find out the error that has occurred.

⚠ fflush, fcloseall, errno.

FFORCE — GEMDOS

```
ret = Fforce(new_handle,handle);
int new_handle,handle;
```

Forces the given file handle to point to the same file as `new_handle`.

If an error occurs the value returned is negative, otherwise it is 0.

FGETC — ANSI

```
#include <stdio.h>
c = getc(fp);
c = fgetc(fp);
int c;
FILE *fp;
```

Reads a character for the file `fp` that has been opened using `fopen`.

The value `EOF` is returned if the end of the file is reached or an error occurs. If an error has occurred the variable `errno` indicates which error.

Be careful not to assign the result of this function to a variable of type `char`. If you do this end of file detection will be impossible.

The two functions `getc` and `fgetc` are equivalent.

⚠ fgetchar.

The program below reads a file byte by byte and displays it on the screen byte by byte. An equivalent but much faster version is given under the function `fgets`.

```
#include  <stdio.h>
void  main()
{
FILE  *fp;
int  i;
        fp =  fopen("test.txt",  "r");
        if  (!fp)
        {
            printf("Fatal  error  ...");
            exit(0);
        }

        while  ((i  =  getc(fp))  !=  EOF)
            putchar(i);
        fclose(fp);

}
```

FGETCHAR ANSI

```
c  =  getchar();
c  =  fgetchar();
int  c;
```

Reads a character from the file `stdin` (the keyboard by default).

The two functions `getchar` and `fgetchar` are equivalent.

fgetc.

FGETDTA GEMDOS

```
buffer_adr  =  Fgetdta();
char *buffer_adr;
```

This function returns the address of the buffer used by the GEMDOS directory search functions.

FGETS ANSI

```
#include  <stdio.h>
p  =  fgets(buffer,  len,  fp);
char  *p,  *buffer;
int  len;
FILE  *fp;
```

Reads a string of characters terminated by a new line (`'\n'`) from the file given by `fp`. This file must have been opened for read.

A maximum of `len` characters are read. Reading stops if either `len` characters have been read or a `'\n'` character is read. A null character `'\0'` is stored instead of the new line or after the last character read.

The value returned is the same as the buffer unless an error occurs. In this case 0 is returned and `errno` indicates the source of the error.

gets, fread, errno.

The program below reads a file line by line and displays it on the screen a line at a time. Compare this with the `fgetc` example.

```
#include  <stdio.h>
void  main()
{
FILE  *fp;
char  *p,  buf[82];
        fp  =  fopen("test.txt",  "r");
        if  (!fp)
        {
                printf("fatal  error");
                exit(0);
        }
        while  (p  =  fgets(buf,  80,  fp))
                printf("%s",  buf);
        fclose(fp);
}
```

FILENO ANSI

```
#include  <stdio.h>
fd  =  fileno(fp);
int  fd;
FILE  *fp;
```

Returns the file descriptor associated with the file pointer `fp` that has been opened using `fopen`.

fdopen.

see std.

FLOOR ANSI

```
ret = floor(var);
double ret,var;
```

Returns the whole number less than or equal to the value of the argument.

Although the result has a zero fractional part it is a `double` not an `int`.

ceil.

FLOPFMT GEMDOS

```
ret = Flopfmt(buffer, 0, drive_no, sectors_per_track, track_no,
      side, reserved_sectors, 0x87654321, virgin);
int ret, drive_no, sectors_per_track, track_no, side;
int reserved_sectors, virgin;
char buffer[8192];
```

Formats a track (`track_no`) on a floppy (`drive_no`: 0=A, 1=B) on a given side (`side`). The number of sectors per track (`sectors_per_track`) and reserved sectors must be specified as parameters. `virgin` gives the word fill value for new sectors.

The value returned is 0 if the operation succeeded.

FLOPRD GEMDOS

```
ret = Floprd(buffer, 0, drive_no, sector, track,
      side, no_sectors);
int ret, drive_no, sector, track, side, no_sectors;
char *buffer;
```

Reads sectors from a given floppy disk and stores the bytes at address `buffer`. This function reads `no_sectors` starting at the given `sector`, `track` and `side` from the drive `drive_no` (A=0, B=1).

The value returned is zero if the operation was successful.

FLOPVER GEMDOS

```
ret = Flopver(buffer, 0, drive_no, sector, track,
      side, no_sectors);
int ret, drive_no, sector, track, side, no_sectors;
char *buffer;
```

Verifies floppy disk sectors and stores the numbers of those that fail in `buffer`. The other parameters are the same as for `Floprd`.

FLOPWR GEMDOS

```
ret = Flopwr(buffer, 0, drive_no, sector, track,
     side, no_sectors);
int ret, drive_no, sector, track, side, no_sectors;
char *buffer;
```

Writes one or more sectors to a floppy disk from `buffer`. The other parameters are the same as for `Floprd`.

The value returned is 0 if the write was successful.

FMOD ANSI

```
a = fmod(b, c);
double a, b, c;
```

Real number modulus.

This function returns `b mod c`.

`a`, `b` and `c` satisfy the relation : `b = k*c + a`.

C % operator.

FOPEN ANSI

```
#include <stdio.h>
fp = fopen(name,mode);
FILE *fp;
char *name,*mode;
```

This function opens a file for the types of i/o given by `mode`.

`name` is the file to access including an optional path specifier.

`mode` is a string of one to three characters.

The first character must be present and should be one of:

r	open for read.
w	create and open for write.
a	open for write having moved to the end of the file (append). If the file does not exist it is created.

The string may contain the character +. This indicates that the file is opened for update (i.e. both read and write). If `"r+"` is used i/o starts at the beginning of the file; if `"w+"` is used the file is created (any existing file is deleted) and if `"a+"` is used then i/o starts at the end of the file.

The `mode` string may also contain the character `"a"` and if so the file is treated as an ASCII file. In this case when reading CR/LF pairs (0x0d,0x0a) are replaced by a single 0x0a. When writing, 0x0a is expanded to CR/LF. In addition Ctrl-Z (0x1a) is considered to indicate the end of file. Alternatively, you may include a `"b"` and the file will be treated as a binary file, so that no conversions will be down.

Examples :

`"r"`	open the file for read
`"r+"`	open a file for read/write
`"ra+"`	open an ASCII file for read/write.
`"wb"`	open a new binary file for write

The value returned by the function is a file pointer for accessing the file that has been opened. If this value is 0, the file could not be opened and `errno` will indicate why.

open, fdopen, freopen, fclose.

FOPEN GEMDOS

```
file_handle = Fopen(file_name, open_mode);
int file_handle, open_mode;
char *file_name;
```

Opens the file with name `file_name` for read and/or write depending on the value of `open_mode`:

0	open for read
1	open for write
2	open for read/write

The value returned is either a positive file handle or a negative GEMDOS error number following an error.

FORM_ALERT GEM AES

```
exit_button = form_alert(default_button, string);
int exit_button, default_button;
char *string;
```

Displays an alert dialog box on the screen and waits for user input.

`default_button` is the button number that will be returned if the user types `Return`.

`string` is a string of characters describing the alert message.

The returned value is the button selected by the user.

FORM_CENTER GEM AES

```
#include  <gemlib.h>
form_center(box_adr,  &x,  &y,  &w,  &h);
OBJECT  *box_adr;
short  x,y,w,h;
```

Centres a dialog box in the screen.

`box_adr` is the address of the tree to be centred.

The function returns `x,y,w,h` as the co-ordinates of the box.

FORM_DIAL GEM AES

```
ret  = form_dial(type,  x1,y1,w1,h1,  x2,y2,w2,h2);
int  type,  ret;
int  x1,y1,w1,h1;
int  x2,y2,w2,h2;
```

Initialise or finish the display of a dialog box.

FORM_DO GEM AES

```
#include  <gemlib.h>
ret  =  form_do(dialog,  field_no);
int  ret,  field_no;
OBJECT  *dialog;
```

Lets the user interact with a dialog box.

FORM_ERROR GEM AES

```
ret  =  form_error(error_no);
int  ret,error_no;
```

Displays a GEMDOS error message given by `error_no`.

FPRINTF ANSI

```
#include  <stdio.h>
len  =  fprintf(fp,format,arg1,arg2,...);
int  len;
FILE  *fp;
char  *format;
????  arg1,arg2,...
```

Write formatted text to the file given by the file pointer `fp`. This file must have been opened using `fopen`.

`????` indicates that the parameters may be of different types.

This function is the same as `printf` except that the text is written to a file rather than to the standard output (the screen by default).

FPUTC ANSI

```
#include  <stdio.h>
ret  =  fputc(c,fp);
ret  =  putc(c,fp);
int  c,  ret;
FILE  *fp;
```

Writes a character `c` to a file given by `fp`. `fputc` and `putc` are identical.

If an error occurs the function returns -1 and `errno` indicates the reason for the failure.

fputchar, putchar, errno.

FPUTCHAR ANSI

```
c  =  fgetchar();
int  c;
```

Reads a file from the standard input (the keyboard by default). The ASCII code of the character is returned by the function.

putchar, fputc.

FPUTS ANSI

```
#include  <stdio.h>
ret  =  fputs(s,fp);
int  ret;
char  *s;
FILE  *fp;
```

Writes a string of characters to the file given by `fp`. The file must be opened for write.

If an error occurs this function returns -1 and `errno` will indicate which error.

⚠ fwrite, puts, errno.

FREAD ANSI

```
#include  <stdio.h>
num  =  fread(buffer,  item_size,  n,  fp);
int  num,n,item_size;
char  *buffer;
FILE  *fp;
```

Reads `n` items of size (in bytes) `item_size` from the file given by `fp`. This file must have been opened with `fopen`.

The items are read so long as end of file is not reached or an error does not occur.

`buffer` is the address where the items are stored.

The function returns the number of whole blocks read. If this is less than `n` then an error has occurred or the end-of-file has been reached. These two cases can be distinguished by using the `ferror` and `feof` functions.

⚠ fgets, fgetc, feof, ferror, errno.

FREAD GEMDOS

```
bytes_read  =  Fread(fhandle,  n,  buffer);
int  fhandle,  bytes_read,  n;
char  *,  buffer;
```

Reads `n` bytes from the file given by `fhandle` (as returned from `Fopen`). The number of bytes to read is given by `n` and the bytes read are stored in `buffer`.

The function's result is the number of bytes actually read. This value will be less than `n` if an error occurs or the end-of-file is reached.

FREE ANSI

```
ret = free(mem);
char *mem;
int ret;
```

This function frees a block of memory that has been allocated with `malloc`, `calloc` or `realloc`.

The value passed as a parameter must have been returned by one of the allocation functions and must not have already been freed.

The value returned is 0 if successful ; -1 if an error occurred.

⚠ malloc, calloc, realloc, Malloc, Mfree.

FRENAME GEMDOS

```
ret = Frename(0, old_name, new_name);
int ret;
char *old_name, new_name;
```

Renames an existing file `old_name` to have the name `new_name`.

The value returned is 0 if all is well.

FREOPEN ANSI

```
#include <stdio.h>
fp = freopen(file_name, mode, fp)
FILE *fp,*fp;
char *file_name,*mode;
```

Close a file and open another one using the same file pointer.

The file `fp` is closed and is immediately re-opened to refer to the file `file_name` in the given access mode. The `mode` and `file_name` parameters are the same as for the `fopen` function.

The value returned is the file pointer and is the same as that passed as a parameter.

This function is generally used to redirect the standard input/output files. For example to divert screen output (via `stdout`) to a file.

⚠ fopen, fdopen, dup.

FREXP ANSI

```
a = frexp(b, exp);
double a, b;
int *exp;
```

Split a floating point number `b` into its mantissa `a` (0.5<=a<1.0) and its exponent `exp` (-1024<exp<1024).

The following relation is satisfied:

```
b = a * (2 ^ exp)
```

FSCANF ANSI

```
#include <stdio.h>
n = fscanf(fp, format, arg1, arg2,...);
int n;
FILE *fp;
???? *arg1,*arg2,...
```

This function is equivalent to `scanf` except that input is taken from the file `fp` rather from the standard input (`stdin`).

`????` indicates that the parameters may be of different types.

Note that

```
  fscanf(stdin,format,arg1,...)
```

is equivalent to

```
scanf(format,arg1,...);
```

scanf, sscanf, cscanf.

FSEEK ANSI

```
#include <stdio.h>
ret = fseek(fp, pos, mode);
int ret, pos, mode;
FILE *fp;
```

This function changes where in a file the next read or write will occur. The position depends on the `mode`:

0	`pos` gives the number of bytes from the start of the file.
1	`pos` gives the number of bytes from the current position.
2	`pos` gives the number of bytes from the end of the file (this number must always be 0 or negative).

The value returned by the function is 0 if the operation was successful or non-zero if an error occurred in which case `errno` indicates the type of error.

When `mode`=2, the number of bytes must be negative to move to *before* the end of the file. It is not possible to position *past* the end of file.

The `ftell` function returns the current position within the file.

lseek, ftell, rewind.

FSEL_INPUT GEM AES

```
ret = fsel_input(directory,file_name,&ok);
int ret;
short ok;
char *directory, *file_name;
```

This function makes the file selector appear on the screen, lets the operator use it and returns the value selected.

`path` is a string containing the drive and directory. This is updated when the function returns.

On input, `file_name` is the name of the default file, on exit it is the value that the user has entered; it should be at least 13 bytes long.

`ok` is returned as 1 if the user clicks on OK or 0 if on Cancel.

`ret` is 0 if an error occurred and 1 if there was no error.

This function displays the file selector and stores in `whole_name` the entire name of the file selected (the path + the name) and returns 1 if the name is valid and 0 otherwise.

The path and file names are, by default, preserved between successive calls to the function, so that if the user changes drive or folder, the file selector will display those that the user has previously selected.

```
char path_name[70] = "A:\*.*", file_name[12] = "";
char whole_name[80];
int ask_for_name()
{
        short ok_clicked;
        char *p;
        int ok;
        ok = 0;
        if (fsel_input(path_name,file_name,&ok_clicked) && ok_clicked)
        {
                strcpy(whole_name, path_name);
                if (p = strrchr(whole_name, '\\'))
                {
                        strcpy(p+1, file_name);
                        ok = 1;
                }
        }
        return(ok); /* 1 if name ok, 0 otherwise */
}
```

FSETDTA GEMDOS

```
Fgetdta(buffer);
char  *buffer;
```

Set the GEMDOS data transfer address as used by the file search functions.

FSFIRST GEMDOS

```
ret  =  Fsfirst(file_spec,  attributes);
int  ret,  attributes;
char  *file_spec;
```

Searches for the first file matching the `file_spec` with the corresponding attributes (see `Fattrib`). The `file_spec` may contain the wildcards `*` and `?`.

If a file is found, the function returns 0 and the GEMDOS data transfer area (found using `dta=Fgetdta()`) contains the following information:

`dta[21]`	file attributes
`dta[22]` and `dta[23]`	time stamp
`dta[24]` and `dta[25]`	date stamp
`dta[26]` to `dta[29]`	file size
`dta[30]` onwards	file name

FSNEXT GEMDOS

```
ret = Fsnext();
int  ret;
```

Searches for the next file satisfying the conditions given by `Fsfirst`. This function must not be called without calling `Fsfirst`.

This function returns 0 if a file is found and the information found is stored in the GEMDOS `dta`. See `Fsfirst`.

FTELL ANSI

```
#include  <stdio.h>
pos  =  ftell(fp)
int  pos;
FILE  *fp;
```

Returns the position in the file `fp` of the next byte that will be read or written.

⚠ This position may be modified by using `fseek`.

FWRITE ANSI

```
#include  <stdio.h>
items_written  =  fwrite(buffer,  item_size,  n,  fp);
int  items_written,  n,  item_size;
char  *buffer;
FILE  *fp;
```

Writes `n` items of size `item_size` bytes to the file given by `fp`. The file must have been opened for write using `fopen`.

The items are written until `n` items have been written or an error occurs.

`buffer` is the buffer where the items to write are read from.

The function returns the number of entire items written as its result. If this number is less than `n` then an error has occurred.

fputs, fputc, ferror, errno.

FWRITE GEMDOS

```
bytes_written  =  Fwrite(fhandle,  no_bytes,  addr);
int  bytes_written,  no_bytes,  fhandle;
char  *addr;
```

Writes `no_bytes` from the buffer at `addr` to the file given by `fhandle`.

This function returns the number of bytes actually written. After an error, this will be less than `no_bytes`.

GEMDOS GEMDOS

```
ret  =  gemdos(no,  arg1,  arg2...);
int  no,result;
long  arg1,arg2,arg2...
```

Call a GEMDOS function.

`no` is the function to call.

`arg1,arg2...` are the parameters for this particular function.

⚠ GEMDOS functions may be called directly using their names. For example `Cconin()` is equivalent to `gemdos(1)`.

⚠ bios, xbios.

GETBPB GEMDOS

```
bpb  =  Getbpb(drive_no);
short  *bpb;
int  drive_no;
```

Returns a pointer to the Bios Parameter Block (bpb) for the disk drive given by `drive_no` (0=A,1=B,...).

GETC UNIX

Equivalent to `fgetc`.

GETCHAR UNIX

Equivalent to `fgetchar`.

GETCWD UNIX

```
path = getcwd(p, length);
char *path, *p;
int length;
```

This function returns the current directory for the current drive.

If the pointer `p` is not zero then the name of the directory is placed in the buffer pointed to by `p` and `p` is returned as the result of the function.

If `p` is zero, a block of memory of `length` bytes is allocated using `malloc` and the name is stored there. The value returned by the function is a pointer to this block.

The value returned is therefore always a pointer to the current path or is 0 if an error has occurred. In this case the variable `errno` will specify the error.

⚠ If the parameter `p` is zero the program should ensure that it returns the memory allocated by the implicit call to `malloc` by using the `free` function.

Dgetpath, errno.

GETMPB GEMDOS

```
Getmpb(buffer);
char *buffer;
int bytes;
```

Returns a pointer to the Memory Parameter Block used by the system.

GETS ANSI

```
p = gets(buffer);
char *p,*buffer;
```

Reads a string of characters from the standard input file, `stdin`, which is the keyboard by default.

Characters are read into `buffer` until a `Return` character is found. This isn't placed in the buffer but a null character `\0` is placed instead.

The value returned is equal to `buffer`.

⚠ This function doesn't make any checks that the buffer hasn't overflowed. We therefore recommend that you use `fgets(buffer, max, stdin)` which will stop if you enter `max` characters before a carriage return.

fgets, fgetchar.

GETSHIFT GEMDOS

```
ret = Getshift(keys);
int ret, keys;
```

Activate some control keys if `keys>=0` or read which keys are pressed if `keys=-1`. The value returned is always the new current state of the keys.

The keys are given as a bit map as follows:

0	Right Shift
1	Left Shift
2	Control Key
3	Alternate Key
4	Caps lock
5	Clr/Home
6	Insert

GETREZ GEMDOS

```
resolution = Getrez();
int resolution;
```

Returns the current screen resolution 2=high, 1=medium, 0=low.

GETTIME GEMDOS

```
date_time = Gettime();
long date_time;
```

Returns the date and time as a long integer in the same format as for the functions `Tgetdate` and `Tgettime`.

GIACCESS GEMDOS

```
val  =  Giaccess(value,  register_no);
int  val,  value,  register_no;
```

Reads or writes a value from/to a given sound chip register.

For reading pass the register number in `register_no`; to write the register number plus 128.

The value returned is the value read or written.

GRAF_DRAGBOX GEM AES

```
ret  =  graf_dragbox(width,  height,
     x_init,  y_init,
     x_limit,  y_limit,w_limit,  h_limit,
     &x_end,  &y_end);
int   width,height,x_init,y_init;
int   x_limit,y_limit,w_limit,h_limit;
short  x_end,  y_end;
```

Displays an outlined rectangle (with initial co-ordinates `x_init, y_init`) of the specified `width` and `height`. The user may then move the rectangle with the mouse until the mouse button state changes. The box is forced to remain inside the rectangle given by `x_limit, y_limit , w_limit, h_limit`.

The final position of the rectangle is returned in `x_end` and `y_end`.

GRAF_GROWBOX GEM AES

```
ret   =  graf_growbox(x1,y1,w1,h1,  x2,y2,w2,h2);
int   x1,y1,w1,h1;
int   x2,y2,w2,h2;
```

Draws an expanding rectangle in the same way as when a window is opened using the Desktop. `x1,y1,w1,h1` specify the smaller rectangle and `x2,y2,w2,h2` the larger one.

The result of the function is 0 if an error occurs.

GRAF_HANDLE GEM AES

```
vdi_handle  =  graf_handle(&char_width,  &char_height,
      &box_width,  &box_height);
int  vdi_handle;
short  char_width,char_height;
short  box_width,box_height;
```

Returns the VDI virtual workstation that the AES is using and also the size of a character cell and the size of a boxed character.

GRAF_MKSTATE GEM AES

```
graf_mkstate(&x,&y,&button,&keyboard);
short  x,y,button,keyboard;
```

Returns the current position of the mouse (x,y), the state of the mouse buttons (button) and the state of the shift keys (keyboard).

evnt_button.

GRAF_MOUSE GEM AES

```
ret  =  graf_mouse(mouse_form,  adr);
int  ret,mouse_form;
char  *adr;
```

Changes the mouse form.

GRAF_MOVEBOX GEM AES

```
ret  =  graf_movebox(width,  height,
     x_source,  y_source.  x_dest,  y_dest);
int  ret,width,height;
int  x_source,y_source,  x_dest,  y_dest;
```

Displays a box of size (width,height) moving between the two points (x_source,y_source) and (x_dest,y_dest).

GRAF_RUBBERBOX GEM AES

```
ret = graf_rubberbox(x,y, min_width,  min_height,
     &width,  &height);
int  ret,  x,  y;
int  min_width,  min_height;
short  width,  height;
```

Displays an outline box from (x,y) to the current mouse position and lets the user change the size of the box without letting it become smaller than min_width, min_height. When the user releases the mouse the current width and height are returned in the variables width and height.

GRAF_SHRINKBOX GEM AES

```
ret  =  graf_shrinkbox(x1,y1,w1,h1,  x2,y2,w2,h2);
int  x1,y1,w1,h1;
int  x2,y2,w2,h2;
```

Draws a rectangle shrinking in the same way as when you close one of the Desktop's windows. `x1,y1,w1,h1` specifies the smaller rectangle and `x2,y2,w2,h2` the larger one.

`ret` is zero unless there was an error.

GRAF_SLIDEBOX GEM AES

```
#include  <gemlib.h>
position  =  graf_slidebox(tree_adr,  parent,  child,
vertical);
int  position,  parent,  child,  vertical;
OBJECT  *tree_adr;
```

Lets a child object slide within its parent.

GRAF_WATCHBOX GEM AES

```
#include  <gemlib.h>
position  =  graf_watchbox(tree_adr,  object_no,
    in_state,  out_state);
int  position,  object_no,  in_state,  out_state;
OBJECT  *tree_adr;
```

Changes an object's state as the mouse moves inside or outside of the object until the mouse button is released. This should only be called when the mouse button is down.

The value returned (`position`) indicates if the mouse is within the object (1) or outside it (0) when the mouse button is released.

GTEXT_BOX HiSoft C

```
object_no  =  gtext_box(box_no,  x,y,  text,
     char_size,  border,  fill);
int  object_no,  box_no,  x,  y,  char_size,  border,  fill;
char  *text;
```

Adds a graphics text item to a dialog box.

⚠ **Section 3.3.10**, Help command (gtext_bo).

IABS ANSI

```
ret  =  iabs(val);
int  ret,val;
```

This function returns the absolute value of its parameter. Both values are of type integer.

⚠ abs, labs, fabs.

IKBDWS GEMDOS

```
Ikbdws(len,  byte_string);
int  len;
char  *byte_string;
```

Writes a string of `len` bytes from `byte_string` to the keyboard processor.

INIT_BOX HiSoft C

```
box_no  =  init_box(width,  height,  objects);
int  box_no,  width,  height,  objects;
```

Creates and initialises a dialog box.

⚠ **Section 3.3.3**, Help command (init_box).

INIT_MENU HiSoft C

```
entry_info  =  init_menu(desk_name,  file_name, titles,
elements);
int  entry_info,  titles,elements;
char  *desk_name,  *name_info;
```

Initialises a menu.

⚠ **Section 3.4.2**, Help command (init_box).

INITMOUS GEMDOS

```
ret = Initmous(type, param, 0);
int ret, type;
char *param;
```

Initialise the mouse depending on the type of the parameter:

0	disable the mouse
1	enable mouse in relative mode
2	enable mouse in absolute mode
4	enable mode in keyboard mode

`param` is a pointer to a parameter block:

`param[0]`	gives the origin of y co-ordintates: 1=top 0=bottom
`param[1]`	is a parameter to the keyboard's set mouse button command
`param[2]`	is the horizontal mouse scale
`param[3]`	vertical mouse scale
`param[4 and 5]`	maximum horizontal mouse position
`param[6 and 7]`	maximum vertical mouse position
`param[8 and 9]`	initial horizontal mouse position
`param[10 and 11]`	initial vertical mouse position

IOREC GEMDOS

```
buffer_adr = Iorec(device);
int device;
char *buffer_adr;
```

Returns the address of the i/o buffer for a device (0=RS232, 1=keyboard, 2=MIDI).

IS ANSI

```
ret  =  isalnum(c);      /*  alphanumeric character */
ret  =  isalpha(c);      /*  alphabetic character */
ret  =  isascii(c);      /*  ASCII character (<128) */
ret  =  iscntrl(c);      /*  Control character (<32) */
ret  = iscsym(c);        /*  C identifier character */
ret  =  iscsymf(c);      /*  initial C identifier character */
ret  =  isdigit(c);      /*  decimal digit */
ret  =  isgraph(c);      /*  graphics character */
ret  =  islower(c);      /*  lower case letter */
ret  =  isprint(c);      /*  printable character */
ret  =  ispunct(c);      /*  punctuation character */
ret  =  isspace(c);      /*  space, tab or \n */
ret  =  isupper(c);      /*  upper case letter */
ret  =  isxdigit(c);     /*  hexadecimal digit */
int ret, c;
```

These functions test the value of a character `c` and return either (0= FALSE or 1=TRUE) if the character satisfies a certain condition.

For example `islower('e')` returns 1 because `e` is a lower case letter and `isupper('e')` returns 0.

ITEM_MENU HiSoft C

```
entry_no  =  item_menu(title);
int  entry_no;
char  *title;
```

Adds an entry to a menu.

JDISINT GEMDOS

```
ret  =  Jdisint(int_no);
int  ret,  int_no;
```

Disables interrupt `int_no` of the 68901.

JENABINT GEMDOS

```
ret  =  Jenabint(int_no);
int  ret,int_no;
```

Enables the 68901 interrupt number `int_no`.

KBDVBASE GEMDOS

```
descriptor_adr = Kbdvbase();
char *descriptor_adr;
```

Returns the address of the keyboard vector table.

KBRATE GEMDOS

```
period = Kbrate(delay1, delay2);
long period;
int delay1, delay2;
```

Sets or returns the keyboard repeat rate. `delay1` is the delay before key repeat and `delay2` after.

If both parameters are -1 then the current settings are not changed.

The function returns a long composed of the new values of `delay1` and `delay2`.

KEYTBL GEMDOS

```
old_address = Keytbl(unshift, shift, capslock);
char *old_address, unshift, shift, capslock.
```

Sets the BIOS keyboard translation tables.

LABS ANSI

```
ret = labs(val);
long ret,val;
```

This function returns the absolute value of its parameter. Both values are of type `long`.

abs, iabs, fabs.

LDEXP ANSI

```
a=ldexp(b,exp);
double  a,b;
int  exp;
```

Produces a floating point number from a mantissa `b` (`0.5<=b<1`) and exponent `exp`. This is the inverse operation to `frexp`.

The following relation is true:

```
a = b * (2 ^ exp)
```

⚠ frexp.

LINEA HiSoft C

```
#include <linea.h>
param_adr  =  lineaO(); /* return address of LineA vars */
linea1();               /* draw a point */
linea2();               /* read colour of a point */
linea3();               /* draw a line */
linea4();               /* draw a horizontal line */
linea5();               /* draw a filled rectangle */
linea6();               /* draw part of a polygon */
linea7(blk);            /* bit blk operations */
linea8();               /* write a character */
linea9();               /* show mouse */
lineaa();               /* hide mouse */
lineab();               /* change mouse form */
lineac(save);           /* undraw sprite */
linead(x, y, sprite, save);  /* draw sprite */
lineae();               /* copy raster form */
lineaf();               /* seed fill */

LA_INIT  *param_adr;    /* Pointer to LineA variables */
LA_SPRITE  *sprite, *save;/* Pointers to Sprites */
int  x, y;              /* Sprite co-ordinates */
char  *blk;             /* Pointer to a bitblk structure */
```

The following three structures are used when calling the LineA routines. They are defined in the linea.h file.

```
typedef  struct  la_variables

{
    short la_planes;        /*number of planes n */
    short la_width;         /* number of bytes/line */
    short *la_contrl;       /* pointer to the contrl array */
    short *la_intin;        /* pointer to the intin array */
    short *la_ptsin;        /* pointer to the ptsin array */
    short *la_intout;       /* pointer to the intout array */
    short *la_ptsout;       /* pointer to the contrl array */
    short la_col0bit;       /* colour for plane 0 */
    short la_col1bit;       /* colour for plane 1 */
    short la_col2bit;       /* colour for plane 2 */
    short la_col3bit;       /* colour for plane 3 */
    short la_lstlin;        /* whether last line point is plotted*/
    short la_lnmask;        /* polyline type */
    short la_wmode;         /* display mode */
    short la_x1;            /* x1 coord */
    short la_y1;            /* y1 coord */
    short la_x2;            /* x2 coord */
    short la_y2;            /* y2 coord */
    short *la_patptr;       /* pointer to current fill pattern */
    short la_patmsk;        /* fill pattern mask */
    short la_mfill;         /* multi-plane fill flag */
    short la_clip;          /* clipping flag */
    short la_xmincl;        /* minimum x clipping value */
    short la_ymincl;        /* minimum y clipping value */
    short la_xmaxcl;        /* maximum x clipping value */
    short la_ymaxcl;        /* maximum y clipping value*/
    short la_xdda;          /* accumulator for linea8 */
    short la_ddainc;        /* fractional amount to scale up/down */
    short la_scaldir;       /* scale direction flag (0=down,1=up) */
    short la_mono;          /* 1 if mono-spaced,0=proprtional */

    short la_xsource;       /* x co-ord of char in font form */
    short la_ysource;       /* y co-ord of char in font form */
    short la_dstx;          /* x co-ord of character on screen*/
    short la_dsty;          /* y coord y of character on screen */
    short la_delx;          /* width of character */
    short la_dely;          /* height of character */
    short *la_fbase;        /* pointer to font base */
    short la_fwidth;        /* width of font form */
    short la_style;         /* textblt style */
    short la_litemask;      /* mask used to "grey" text*/
    short la_skewmask;      /* mask used to skew text */
    short la_wieght;        /* width to thicken text */
    short la_roff;          /* offset above char when skewing */
    short la_loff;          /* offset below char when skewing*/
    short la_scale;         /* scaling flag */
    short la_chup;          /* character rotation vector */
    short la_textfg;        /* text foregroung colour */
    short *la_scrtchp;      /* pointer to special effects buffers */
    short la_scrpt2         /* offset of 2nd buffer */
    short la_textbg;        /* text background colour */
    short la_copytran;      /* copy raster form type */
}LA_VARIABLES;
```

```
typedef struct la_init
{
    struct la_variables *la_a0;
                                /* pointer to the LA_VARIABLES area */
    long la_a1;                 /* pointer to system font header list */
    long la_a2;                 /* pointer to LineA routines */
}LA_INIT;

typedef struct la_sprite
{
    short la_xhot;              /* x offset of sprite hot spot */
    short la_yhot;              /* y offset of sprite hot spot */
    short la_format;            /* 1 for VDI, -1 for XOR */
    short la_col1;              /* background colour */
    short la_col2;              /* foreground colour */
    short la_image[32];         /* sprite image */
                                /*(words of mask & data alternate) */
}LA_SPRITE;
```

```
#include <linea.h>
LA_INIT *ptr;
LA_VARIABLES *p;
void main()
{
     ptr = linea0();

     /* make p contain the address of the lineA variables*/
     p = ptr->la_a0;

     /* draw a point */
     (p->la_ptsin)[0] = 100;
     (p->la_ptsin)[1] = 50;
     (p->la_intin)[0] = 1;
     linea1();

     /* draw a line */
     p->la_x1 = 105;
     p->la_y1 = 55;
     p->la_x2 = 305;
     p->la_y2 = 105;
     p->la_col0bit = 1;
     linea3();

     /* draw a rectangle */
     p->la_x1 = 300;
     p->la_y1 = 60;
     p->la_x2 = 400;
     p->la_y2 = 120;
     linea5();
}
```

LOG ANSI

```
a  =  log(b);
double  a,  b;
```

Returns the natural log (to base e) of the argument. Both values are double reals.

LOG10 ANSI

```
a = log10(b);
double a, b;
```

Returns the log to base 10 of the argument. Both values are double reals.

LOGBASE GEMDOS

```
scr_addr = Logbase();
char *scr_addr;
```

Returns the address of the logical screen (that which software routines modify).

LQSORT ANSI

```
lqsort(arr,n);
long *arr;
int n;
```

This function sorts an array `arr` of long integers into ascending order.

⚠ dqsort, sqsort, tqsort.

LSEEK UNIX

```
ret = lseek(fd, pos, mode);
int ret, fd, pos, mode;
```

Moves the input/output position on the UNIX file `fd` depending on the value of `mode`:

0	`pos` gives the number of bytes from the start of the file
1	`pos` gives the number of bytes from the current position
2	`pos` gives the number of bytes from the end of the file (this number must always be 0 or negative)

The value returned by the function is 0 if the operation was successful or non-zero if an error occurred in which case `errno` indicates the type of error.

When `mode=2`, the number of bytes must be negative to move to *before* the end of the file. It is not possible to position *past* the end of file.

This function returns the new position from the beginning of the file.

This function may be used to find the length of a file, as follows:

```
length = lseek(fd, 0, 2);
```

fseek.

MALLOC ANSI

```
adr  =  malloc(size);
char  *adr;
unsigned long size;
```

Allocates a block of memory of the size that is passed as a parameter.

The value returned is a pointer to the allocated memory or zero if the allocation was not possible.

The memory is allocated from GEMDOS system memory whose size is fixed when the interpreter is loaded. This can be changed if you have sufficient memory.

Section 1.4.13, Malloc, Mfree, calloc, free, realloc.

MALLOC GEMDOS

```
adr  =  Malloc(size);
char  *adr;
unsigned long size;
```

Allocates a block of memory whose size is passed as a parameter.

The value returned is a pointer to the allocated memory or 0 if the allocation was not possible.

This function returns the number of free bytes if the value passed is -1.

malloc

MATHERR UNIX

```
#include <math.h>

ret = matherr(mathstr);
int ret;
struct exception *mathstr;
```

This function is called when an error occurs during the execution of a mathematical function. The details of the error are stored in the `mathstr` structure which is modified by the function.

MAX ANSI

```
larger = max(a, b);
int a, b, larger;
```

This function returns the larger of the two integers passed as parameters.

MEDIACH GEMDOS

```
test = Mediach(drive_no);
int test, drive_no;
```

Tests if the floppy disk has been changed in drive A (`drive_no=0`) or drive B (`drive_no=1`).

The value returned is:

0	the disk has not changed
1	the disk may have changed
2	the disk has changed

MEMCCPY UNIX

```
p = memccpy(dest, source, ch, bytes);
int bytes, h;
char *dest, *source, *p;
```

This function copies a block of memory from `source` to `dest`.

`bytes` gives the number of bytes to copy.

The copy stops when the number of bytes has been copied or the character `ch` is found.

The value returned is a pointer and has the same value as `dest`.

memcpy, strcpy, strncpy.

MEMCHR UNIX

```
pos  =  memchr(block,  ch,  bytes);
int  ch,  bytes;
char  *block,  *pos;
```

This function returns a pointer to the first character `ch` found in the block starting at `block` and of `length` bytes.

If the character cannot be found 0 is returned.

⚠ strchr, strrchr.

MEMCMP UNIX

```
comp  =  memcmp(block1,  block2,  bytes);
char  *block1,  *block2;
int  comp,  bytes;
```

This function compares two blocks, `block1` and `block2`, of `length` bytes.

If the value returned is 0 then the two blocks are identical.

If not, the comparison stops as soon as two bytes differ and the value returned is positive or negative depending on whether the character from the first block is larger or smaller than the second respectively.

⚠ strncmp.

MEMSET UNIX

```
p  =  memset(buf,  ch,  len);
char  *buf,  *p;
int  len,  ch;
```

This function fills a block of memory pointed to by `buf` and `len` bytes long. The value returned is a pointer to the same place as `buf`.

MEMCPY UNIX

```
dest  =  memcpy(dest,  source,  len);
int  len;
char  *dest,  *source;
```

This function copies a block of `len` bytes from `source` to `dest`. The value returned is `dest`.

The areas to be copied may partially overlap.

MENU_BAR GEM AES

```
#include  <gemlib.h>
ret  =  menu_bar(menu_adr,  display);
int  ret,  display;
OBJECT  *menu_adr;
```

Displays (display=1) or deletes (display=0) the menu bar given by menu_adr.

The value returned is 0 following an error.

MENU_ICHECK GEM AES

```
#include  <gemlib.h>

ret  =  menu_icheck(menu_adr,  entry_no,  check);
int  ret,  entry_no,  check;
OBJECT  *menu_adr;
```

Displays (check=1) or removes (check=0) a tick in front of menu entry entry_no within the tree menu_adr.

The value returned is 0 following an error.

MENU_IENABLE GEM AES

```
#include  <gemlib.h>

ret  =  menu_ienable(menu_adr,  entry_no,  enable);
int  ret,  entry_no,  active;
OBJECT  *menu_adr;
```

Enables (enable=1) or disables (enable=0) entry number entry_no from menu tree menu_adr. Disabled menus are displayed in grey.

The value returned is 0 following an error.

MENU_REGISTER GEM AES

```
entry_no  =  menu_register(appl_id,  entry_name);
int  appl_id,  entry_no;
char  *entry_name;
```

Adds the name of a desk accessory within name entry_name and application id appl_id to the desk menu. The appl_id is returned by appl_init.

The value returned is the entry number within the desk menu.

MENU_TEXT GEM AES

```
#include <gemlib.h>
ret = menu_text(menu_adr, entry_no, string);
int ret, entry_no;
char *string;
OBJECT *menu_adr;
```

Changes the text for the menu entry `entry_no` of tree `menu_adr` to be `string`.

The value returned is zero if an error occurred.

MENU_TNORMAL GEM AES

```
#include <gemlib.h>

ret = menu_tnormal(menu_adr, title_no, inverse);
int ret, title_no, inverse;
OBJECT *menu_adr;
```

Displays a menu title in inverse video (`inverse=1`) or normal (`inverse=0`). The menu title is given by `title_no` and the tree address `menu_adr`.

The value returned is 0 following an error.

MFPINT GEMDOS

```
Mfpint(vector_no, adr);
int vector_no;
char *adr;
```

Sets interrupt vector `vector_no` to be `adr`.

MFREE GEMDOS

```
ret = Mfree(pointer);
int ret;
char *pointer;
```

Frees the memory array given by `pointer` which has been allocated by `Malloc`.

This function returns zero if successful.

MIDIWS GEMDOS

```
Midiws(len, str);
int len;
char *str;
```

Sends a string of `len` characters given by `str` to the MIDI port.

MIN ANSI

```
smaller = min(a, b);
int a, b, smaller;
```

Returns the smaller of two whole numbers `a` and `b`.

MKDIR UNIX

```
ret = mkdir(directory_name);
int ret;
char *directory_name;
```

Creates a new directory called `directory_name`.

The value returned is 0 if successful; if the operation fails `-1` is returned and `errno` will indicate the source of the error.

⚠ Dcreate, errno.

MODF ANSI

```
frac = modf(dbl, &whole);
double whole, dbl, frac;
```

Splits a double floating point number into its whole (returned in `whole`) and fractional part (returned as the result of the function).

Of course,

```
dbl = whole + frac;
```

⚠ fmod, frexp, ldexp.

MOUSE ANSI

```
status = mouse(&x, &y, button_no);
int x,y,status,button_no;
```

Reads the position of the mouse into (x,y) after the user has clicked a button.

If `button_no=0` then the function doesn't wait for a click but returns the mouse position immediately.

If `button_no=1` then `mouse` waits for a click on the left button, 2 the right button, 3 both buttons at once.

The value returned gives the state of the button in the same form as for the parameter `button_no`.

Help command, mouse.

MSHRINK GEMDOS

```
ret = Mshrink(base_page, bytes);
int ret, bytes;
char *base_page;
```

Free memory to system during initialisation.

`base_page` is the program's base page; `bytes` is the number of bytes to return.

The value returned by the function is zero if the operation was successful.

OBJC_ADD GEM AES

```
#include <gemlib.h>
ret = objc_add(tree_adr, parent_obj, add_obj);
int ret, parent_obj, add_obj;
OBJECT *tree_adr;
```

Adds an object (of index `add_obj`) to have parent `parent_obj` in the tree `tree_adr`.

The value returned is 0 if an error occurs.

OBJC_CHANGE GEM AES

```
#include  <gemlib.h>
ret  =  objc_change(tree_adr,  object_no,  0,  x,  y,  w,  h,
      object_state,  draw_object);
int  ret,object_no;
int  x,  y,  w,  h;
int  object_state,  draw_object;
OBJECT   *tree_adr;
```

Changes the state of the object `object_no` in the tree `tree_adr` to be `object_state`. If `draw_object =1` then re-draw the object with the rectangle (x,y,w,h).

The value returned is 0 if an error occurs.

OBJC_DELETE GEM AES

```
#include  <gemlib.h>
ret  =  objc_delete(tree_adr,  object_no);
int  ret,  object_no;
OBJECT   *tree_adr;
```

Removes the object `object_no` from the tree `tree_adr`.

OBJC_DRAW GEM AES

```
#include  <gemlib.h>
ret  =  objc_draw(tree_adr,  object_no,  level,  x,  y,  w,  h);
int ret,  object_no,  level;
int x,  y,  w,  h;
OBJECT  *  tree_adr;
```

Draws an object (`object_no`) in a tree (`tree_adr`) within the clipping rectangle (x,y,w,h).

If `level=0` then the function draws just the object itself.

If `level=1` then the function draws the object and its children.

If `level=2` then the object, its children and grand children are drawn.

The maximum `level` is level 10.

Zero is returned if there was an error.

OBJC_EDIT GEM AES

```
#include  <gemlib.h>

ret  =  objc_edit(tree_adr,  object_no,  character,  position,
     kind,  &ret_position);
int  ret,  object_no,  character,  position,  kind;
short  ret_position;
OBJECT  *  tree_adr;
```

Display a character in a G_FTEXT object (given as `object_no` of the tree `tree_adr`).

`position` gives the index where the character is to be added. The position after the addition is given as `ret_position`.

The operation performed depends on the value of `kind`:

0	`ED_START`	reserved
1	`ED_INIT`	display the string and turn the text cursor on
2	`ED_CHAR`	validate the character and re-display string
3	`ED_END`	turn text cursor off

The value returned is zero if an error occurred.

OBJC_FIND GEM AES

```
#include  <gemlib.h>
object_found  =  objc_find(tree_adr,  object_no,  level,  x,  y);
int  object_no,  level,  object_found  ;
int  x,  y;
OBJECT  *  tree_adr;
```

Finds which object is under co-ordinates `(x, y)`.

The search is within tree `tree_adr` starting at object `object_no` (normally 0). The search descends to a level given by `level` (see `objc_draw`).

The value returned is -1 if the object is not found; otherwise it is the index of the object that has been found.

OBJC_OFFSET GEM AES

```
#include  <gemlib.h>

ret  =  objc_offset(tree_adr,  object_no,  &x,  &y);
int  ret,  object_no;
short  x,  y;
OBJECT  *  tree_adr;
```

Returns the co-ordinates of `object_no` of tree `tree_adr` in `(x, y)`.

The value return is zero if an error occurs.

OBJC_ORDER GEM AES

```
#include  <gemlib.h>
ret  =  objc_offset(tree_adr,  object_no,  new_position);
int  ret,  object_no,  new_position;
OBJECT  *  tree_adr;
```

Moves an object (`object_no`) from tree `tree_adr` within the tree structure. The value `new_position` (0,1,2...) indicates which child (first, second, third) of its parent the new object is to be.

The value return is zero if an error occurs.

OFFGIBIT GEMDOS

```
Offgibit(bit_no);
int  bit_no;
```

Resets (to 0) a bit of port A of the sound chip. Which bit to zero is passed as a parameter.

ONGIBIT GEMDOS

```
Ongibit(bit_no);
int  bit_no;
```

Sets (to 1) a bit of port A of the sound chip. Which bit to set is passed as a parameter.

OPEN UNIX

```
#include  <fcntl.h>
nf  =  open(file_name,  mode,  access);
char  *file_name;
int nf,  mode,  access;
```

This function opens a file `file_name` and returns its file handle. `mode` indicates how the file is to be opened as given in the following table. These constants are defined in the file fcntl.h):

`O_RDONLY`	read only
`O_WRONLY`	write only
`O_RDWR`	read and write
`O_APPEND`	write starting at the end of the file
`O_TRUNC`	if the file exists, delete it
`O_CREAT`	if the file doesn't exist create it
`O_EXCL`	(used only with `O_CREAT`) if the file exists *don't* open it

The values above can be combined using the OR operator e.g. `O_CREAT|O_EXCL`.

The `access` parameter specifies how the file may be accessed and takes one of the following values:

`access = S_IREAD` : file is read only.

`access = S_IREAD|S_IWRITE` or `mode = 0` : file is open for read/write.

The value returned by the function is the file number which must be used for all input-output operations on the file.

If an error occurs, this function returns -1. The variable `errno` indicates which error has occurred.

When you open a file with this function, you can only use the Unix input/output functions (`read` and `write` for example) and not the ANSI ones (e.g. `fread`, `fwrite`, `fprintf`).

creat, chmod, close, fopen, errno.

OPEN_WINDOW HiSoft C

```
window_no = open_window(attributes, x, y, w, h, title,
comment);
int window_no, attributes, x, y, w, h;
char *title, *comment;
```

Opens a window.

Section 3.2.3, Help command (open_win).

PEXEC GEMDOS

```
base_page = Pexec(execute, program_name, arg_list,
environment);
char *base_page;
int execute;
char *program_name, *arg_list, *environment;
```

Load and/or execute a program in memory.

If `execute` = 0, the program is loaded but not executed. If `execute` = 3, the program is loaded and executed.

`program_name` is the name of the program to load. `arg_list` gives the arguments that you wish to pass onto the program. `environment` gives the environment variables to be passed.

The function returns the address of the base page of the program that has been loaded.

PHYSBASE GEMDOS

```
screen_adr = Physbase();
char *screen_adr;
```

Returns the address of the physical screen (that which is actually displayed on the monitor).

POS_WINDOW HiSoft C

```
pos_window(window_no, column, row);
int window_no, column, row;
```

Positions the text cursor within a window that has been opened by `open_window`.

⚠ **Section 3.2.7**, Help command (pos_wind).

POW ANSI

```
result = pow(number, exponent);
double result, number, exponent;
```

Raise the parameter `number` to the power of `exponent`.

The two parameters and the value returned are both double reals.

The three values satisfy:

```
result= number ^ exponent.
```

⚠ log, log10, exp.

PRINTF ANSI

```
length = printf(format, arg1, arg2,...);
char *format;
int length;
???? arg1, arg2, ...
```

This function performs formatted output. The output is to the standard file `stdout` (the screen, by default).

The `printf` function builds up its output based on a control string and then sends it to the screen by default. The value returned by this function is the number of characters output by `printf`.

The parameter `format` contains both ordinary characters to be simply copied to the screen and format conversion characters with which to write the other arguments to the `printf` function.

Each one of these conversion sequences outputs a string of characters that is not explicitly contained in the format string.

`????` indicates that the parameters may be of different types.

You must have an exact correspondence between the control string and the parameters `arg1`, `arg2`, ... Each conversion specification is associated with one parameter that describes how that parameter is to be written to the screen.

The type of the variable indicated in the specification must correspond exactly with the type of the variable that is passed as a parameter. Otherwise, the results are not predictable...

The conversion specifications are of the form:

```
%[attributes][minimum][.precision][l]type
```

A specification always starts with the character `%`. There then follows various elements in the following order.

1. An optional attributes field which can contain 0 or more of the following characters:

-	A minus sign indicates that the value converted is left justified within the output field specified. Right justification is used by default. This attribute does not have a visible effect on the display unless you specify a minimum width field.
0	The digit 0 is used only in the conversion of numeric values (integer or floats) and when you a use a minimum width field. It indicates that the number is to be preceded by 0 characters rather than spaces.
+	The plus sign can only be used with signed values. It indicates that the value is to be preceded by a plus sign if it is positive. Negative values are always preceded by a minus sign, this cannot be disabled.

Space	This attribute is similar to plus. It can only be used for signed values; it indicates that the value is to be preceded by a space if it is positive. Negative values are always preceded by a minus sign, this cannot be disabled.
#	This attribute may only be used with the numeric conversions `g`, `G`, `f`, `F`, `o`, `x` and `X`. Its effects are described under the appropriate conversion.

2. The optional `minimum` field is a whole number constant. This field can also be replaced with an asterisk (`*`). In this case the next value from `arg1,arg2,...` is used as the value of this field, which must be an integer.

This field specifies the minimum width for this conversion. If the converted value isn't large enough to fill the minimum length then characters (space or `0`) are added to fill the space so that it is always at least minimum size. If the the value converted is too big then the field grows to be large enough.

3. The optional `precision` field is preceded by a decimal point and must be followed by an integer constant. If the decimal point is not followed by a digit it is taken as zero, which is *not* the same as the absence of this precision field.

This field can be an asterisk (`*`) in which case the next argument from `arg1,arg2,...` is used as the value of the field. The argument must be a positive integer.

For the `f` format, fixed point numbers, this value represents the number of digits after the decimal point. For the `e` format, exponential floating point format, it is the number of significant digits. By default this value is 6. This represents the maximum number of digits that will be generated.

When used with the `s` format this is the maximum number of characters to write; any further characters in the string are ignored.

4. An optional letter `l` which when used with the `d`, `o` , `u` and `x` formats indicates that the argument is of type `long`.

5. A compulsory `type` indicator. This is one of the characters: `c`, `d`, `e`, `E`, `f`, `F`, `g`, `G`, `o`, `s`, `u`, `x`, `X`.

d	writes a signed whole decimal integer of type `int` or `long`. The result of the conversion is a sequence of digits preceded by a sign if the argument is negative or if the `+` attribute is used.
u	performs the conversion for an unsigned decimal number. This parameter must be of type `int` or `long`. The result is a string of decimal digits.
o	writes a number in octal. The argument must be `unsigned`. The value is converted into a sequence of digits. If the `#` attribute is used the value converted is preceded by a zero.

`x, X`	writes a number in base 16. The argument is supposed to be a string of type `unsigned`. The result of the conversion is a string of hexadecimal digits. The letters `a-f` are used with `x` and `A-F` are used with `X`. If the attribute `#` is used then the characters are preceded by `0x` or `0X` (with `X`).
`c`	the argument is used as a character. A single character is converted. The type of the parameter can be `char`, `short`, `int` or `long`. In the last three cases, only the least significant byte is used.
`s`	the argument is written as a string of characters. It must be a pointer to a string of characters terminated by a null character. If the `precision` format was specified then this indicates the maximum number of characters to write from the string. The string will be truncated if this value is less than the length of the string.
`f`	write a floating point number (of type `float` or `double`) without using exponential notation. A sequence of digits preceded by an appropriate sign and containing a decimal point is produced. The sign is present if the number is negative or the `+` attribute is used. The number of digits after the decimal point is fixed by the precision asked for (6 by default). If the precision is 0 or the value is a whole number then the decimal point is omitted unless the `#` attribute is used.
`e, E`	write a floating point number (of type `float` or `double`) using exponential notation. The result of the conversion is number of the form `-x.xxxxxxe-xxx` (with `e`) or `-x.xxxxxxE-xxx` (with `E`), `x` can be any digit. The sign before the number is present if the number is negative or if the `+` attribute is used. There is always exactly one digit before the decimal point. The number of significant digits is set by the precision attribute (6 by default). If the precision is 0 or the value has only one significant digit then the decimal point is omitted unless the `#` attribute is used.
`g, G`	write a floating point number (of type `float` or `double`) using fixed point or floating point notation depending on which would require the least characters. If the exponent is greater than -4 and is less than the precision asked for, the fixed point (`f`) notation is used, otherwise the exponential form is used. In this case if `G` is used then `E` format is used, if `g` is used then `e` is used. Non significant zeros and decimal points are suppressed unless the `#` attribute is used.

⚠ `%%` in the format string outputs a single `%` character.

```
printf("To-day is the %d/%02d/%02d\n",day,month,year);
```

would display, for example:

```
To-day is the 19/07/89
```

If x=1.34 then

```
printf("The value is %#*.*g",14,8);
```

produces

```
The value is          1.34000000
```

```
printf("%#07X\n",63);
```

produces

```
0000X3F
```

```
printf("<%10.5s><%-10.5s>","interpreter");
```

produces

```
<     inter><inter     >
```

sprintf, cprintf, fprintf, Help command (printf).

PRINT_WINDOW HiSoft C

```
ret  =  print_window(window_no,  string);
int  ret,  window_no;
char  *string;
```

Writes a string of characters at the cursor position in a window (`window_no`) that has been opened using `open_window`.

Section 3.2.6, Help command (print_wi).

PROTOBT GEMDOS

```
Protobt(buf, serial_no, disk_type, executable);
int serial_no, disk_type, executable;
char buf[512];
```

Produces a boot sector image for track 0 sector 1. `buf` is used for the boot sector image. `disk_type` is 2 for 80 track single-sided and 3 for 80 track double-sided. `executable` is 1 if this is to be an executable boot sector, 0 otherwise.

PRTBLK GEMDOS

```
ret = Prtblk();
int ret;
```

Produces a screen dump on the printer.

This function returns 0 if the operation was successful.

PTERM0 GEMDOS

```
Pterm0();
```

Terminates the running program and returns to the calling program (normally the GEM Desktop).

PTERMRES GEMDOS

```
Ptermres(bytes, return_code);
int bytes, return_code;
```

Terminates the current program, freeing only `bytes` of memory and return to the Desktop. Used for so-called 'TSR' programs. Not useful in **HiSoft C**.

PUNTAES GEMDOS

```
Puntaes();
```

Reboots the AES; i.e. the whole machine.

PUTC ANSI

```
#include <stdio.h>
ret = putc(ch, fp);
int ret, ch;
FILE *fp;
```

Writes the character `ch` to the file `fp`. The returned value is the same as the value written, unless an error occurs. In this case, the value -1 is returned and the variable `errno` indicates the nature of the error.

putchar, fputchar, errno.

PUTCHAR UNIX

```
ret= putchar(ch);
int ret, ch;
```

Writes the character `ch` to the standard output file `stdout` (the screen by default). The returned value is the same as the value written, unless an error occurs. In this case, the value -1 is returned and the variable `errno` indicates the nature of the error.

putc, fputchar, errno.

PUTS ANSI

```
ret= puts(string);
int ret;
char *string;
```

Writes a string of characters to the file `stdout` (the screen, by default). A new-line is written after the string. The value returned is 0 or -1 if there was an error. `errno` will give the type of error in this case.

`puts` moves to the next line after writing the string. This is not the same as `fputs`.

`puts(string)` is *not* the same as `fputs(string,stdout)`.

fputs, errno.

RAND — ANSI

```
num = rand();
int num;
```

Returns a random 32-bit unsigned integer.

srand, Random.

RANDOM — GEMDOS

```
num = Random();
int num;
```

Returns a random 24-bit unsigned integer.

READ — UNIX

```
num = read(nf, buffer, bytes);
int num, nf, bytes;
char *buffer;
```

Reads bytes from the file `nf` which has been opened using the UNIX call `open`.

`buffer` is the address to where the bytes are read. `bytes` is the number of bytes to be read.

`num` is the number of bytes successfully read. Normally this will be equal to `bytes`. However if the end of file is reached, then `num` will be less.

If `num` is -1 then an error has occurred and `errno` will indicate which error.

The buffer that you are reading should be at least `bytes` long; otherwise you will destroy other variables, with possibly fatal consequences.

open, write, errno.

READBUT_BOX — HiSoft C

```
ret= readbut_box(box_no, button_no);
int ret, box_no, button_no;
```

Returns 1 if a button (`button_no`) of a dialog box (`box_no`) is selected.

Section 3.3.7, Help command (readbut_).

READSTR_BOX HiSoft C

```
string =readstr_box(box_no, field_no);
char *string;
int box_no, field_no;
```

Returns the text entered by the user for an editable text field (`text_no`) in a dialog box (`box_no`).

edit_box, **Section 3.3.9**, Help command (readstr_).

REALLOC ANSI

```
new_block = realloc(old_block, new_size);
unsigned int new_size;
char *new_block, *old_block;
```

This function allocates a block of memory whose length is given by `new_size`. The contents of an old block (the memory pointed to by `old_block`) are then copied to the new one and finally the old block is freed.

If `new_size` is larger than the original block size the extra space is filled with zeros. If the `new_size` is smaller than the original size then only part of the block is copied.

calloc, malloc, free.

RECT_INIT HiSoft C

```
rect_init(rectangle, x, y, w, h);
short rectangle[4];
int x, y, w, h;
```

Assign the values `x`,`y`,`w` and `h` to an array `rectangle`.

`x,y,w,h` give the co-ordinates of the top left, the width and the height of the rectangle respectively.

Thus `rectangle[0]` will contain `x`, `rectangle[1]` will contain `y`, `rectangle[2]` `w`, and `rectangle[3]` `h`.

⚠ This could also be written

```
#include  <gemlib.h>
rect_init(&rectangle,  x,  y,  w,  h);
GRECT  rectangle;
int x, y, w, h;
```

`GRECT` is a structure containing four fields, `g_x`, `g_y`, `g_w` and `g_h`. The values passed as parameters are assigned to the corresponding fields in the structure.

see rect_union.

RECT_INTERSECT HiSoft C

```
inter  =  rect_intersect(rect1,  rect2);
short  rect1[4], rect2[4];
int  inter;
```

This function determines the intersection of two rectangles (see `rect_init`) `rect1` and `rect2`. The function returns 1 if they do overlap in which case `rect2` will contain the intersection. If there is no intersection 0 is returned.

RECT_POINT HiSoft C

```
within  =  rect_point(rect,  x,  y);
short  rect[4];
int  x,y;
```

Returns 1 if the co-ordinates (`x`, `y`) are within the rectangle `rect` (See `rect_init`).

RECT_UNION HiSoft C

```
rect_union(rect1,  rect2);
short  rect1[4], rect2[4];
```

Calculates the union of two rectangles. That is to say the smallest rectangle containing `rect1` and `rect2`. `rect2` is returned containing the union.

These two programs below do exactly the same thing: they display the union of two rectangles.

```
#include <gemlib.h>
GRECT arr1;
GRECT arr2;

void main()
{
     rect_init(&arr1, 100, 100, 50, 100);
     rect_init(&arr2, 150, 150, 50, 100);
     rect_union(&arr1, &arr2);
     printf("%d,%d,", arr2.g_x, arr2.g_y);
     printf("%d,%d\n", arr2.g_w, arr2.g_h);
}
short arr1[4],arr2[4];
void main()
{
rect_init(arr1, 100, 100, 50, 100);
rect_init(arr2, 150, 150, 50, 100);
rect_union(arr1, arr2);
printf("%d,%d,%d,%d\n", arr2[0], arr2[1], arr2[2], arr2[3]);

}
```

Second program:

```
short arr1[4]={100,100,50,100};
short arr2[4]={150,150,50,100};

void main()
{
     rect_union(arr1, arr2);
     printf("%d,%d,%d,%d\n", arr2[0], arr2[1], arr2[2], arr2[3]);
}
```

REMOVE ANSI

```
ret=  remove(name);
char  *name;
int  ret;
```

Deletes a disk file whose name is given as a parameter.

This function returns 0 if the operation was successful or another value if an error occurred, in which case `errno` will indicate why.

unlink, Ddelete, errno.

RENAME UNIX

```
ret= rename(old_name, new_name);
int ret;
char *old_name, *new_name;
```

Changes the name of the file. The file `old_name` is renamed to `new_name`.

This function returns 0 if the operation was successful or another value if an error occurred, in which case `errno` will indicate why.

Frename, errno.

REPMEM

```
repmem(buf, element, size, num);
char *buf, *element;
int size, num;
```

This function initialises the area of memory pointed to by `buf` and sets up each element to have the same value.

`element` is a pointer to a buffer containing the initial value of the elements. `size` is the size of each element in bytes. `num` is the number of times to duplicate.

`buf` must be big enough, at least `size*num` bytes.

memchr.

REWIND ANSI

```
#include <stdio.h>
ret= rewind(fp);
int ret;
FILE *fp;
```

Move the file position of the file `fp` to the beginning. This is the position at which reading and writing occurs. The file `fp` must have been opened using `fopen`.

The value returned is 0 if the operation was successful; non-zero otherwise.

Calling this function is equivalent to `fseek(fp,0,0)`.

fseek, ftell, errno.

RMDIR UNIX

```
ret=  rmdir(dir_name);
char  *dir_name;
int  ret;
```

Deletes the directory (folder) named `dir_name`. The directory must be empty.

This function returns -1 if an error occurred and 0 if the operation was successful. As usual `errno` will indicate the reason for the error.

⚠ Dcreate, errno.

RS_ADDRALERT HiSoft C

```
alert_adr  =  rs_addralert(alert_no);
int  alert_no;
char  *alert_adr;
```

This function returns the address of an alert box within the resource file loaded by `rsrc_load`.

`alert_no` is the number of the alert box of which you wish to find the address.

RS_ADDRBUTTON HiSoft C

```
#include  <gemlib.h>
text_adr  =  rs_addrbutton(dial_adr,  button_no);
int  button_no;
OBJECT *  dial_adr;
char *text_adr;
```

This function returns the address of the string of a button or a non-editable text object. This is the address of the string that is displayed.

`dial_adr` is the address of the dialog box (this can be found using `rs_addrdial`). The box will normally have been loaded using `rsrc_load`.

`button_no` is which item to find the address of.

RS_ADDRDIAL HiSoft C

```
#include  <gemlib.h>
dial_adr  =  rs_addrdial(dial_no);
int  dial_no;
OBJECT  *  dial_adr;
```

This function returns the address of a dialog box that has been loaded from a resource file via `rsrc_load`.

`dial_no` is the number of the dialog box that you wish to find.

see rs_drawdial.

RS_ADDREDIT HiSoft C

```
#include  <gemlib.h>
text_adr  =  rs_addredit(dial_adr,  edit_no);
int  edit_no;
OBJECT  *  dial_adr;
char  *text_adr;
```

This function returns the address of an editable text field in a dialog box. This is the address where the characters typed by the user are stored.

`dial_adr` is the address of the dialog box (this can be found using `rs_addrdial`). The box will normally have been loaded using `rsrc_load`.

`edit_no` is which item to find the address of.

see rs_drawdial.

RSCONF GEMDOS

```
Rsconf(speed,  handshake,  control,  rsr,  tsr,  scr);
int  speed,  handshake,  control,  rsr,  tsr,  scr;
```

Configures the RS232 port. If any of the parameters is -1 then the corresponding attribute is not changed.

`speed` contains a value between 0 and 15 which indicates the speed of transmission as a baud rate.

0 : 19200	4 : 2400	8 : 600	12 : 134
1 : 9600	5 : 2000	9 : 300	13 : 110
2 : 4800	6 : 1800	10 : 200	14 : 75
3 : 3600	7 : 1200	11 : 150	15 : 50

The handshake parameter indicates what sort of handshaking is to be used.

0	no handshaking (the default value)
1	XON/XOFF (^S/^Q) i.e software handshake
2	CTS/RTS i.e. hardware handshake
3	both XON/XOFF and CTS/RTS

control indicates how bytes are transmitted:

bit 7	1
bits 5,6	number of bits (00 = 8, 01= 7, 10=6, 11 = 5)
bits 3,4	start and stop bits (00 = neither, 01= 1 start & 1 stop, 10 = 1 Start & 1.5 stop, 11 =1 start & 2 stop).
bit 2	1 = parity on, 0=parity off
bit 1	1 = even parity, 0=even parity
bit 0	0

The rsr, tsr, scr parameters set the corresponding register in the 68901. Thay are not normally changed.

RS_DRAWALERT HiSoft C

```
button_no  =  rs_drawalert(alert_no);
int  button_no,  alert_no;
```

This function draws an alert box on the screen, waits for the user to click on a button and returns which button was selected.

The index of the alert box to draw is passed as the alert_no parameter. This box must be part of a resource file that has been loaded with rsrc_load.

The box's number not its address is passed as a parameter.

RS_DRAWDIAL HiSoft C

```
#include  <gemlib.h>
rs_drawdial(dial_adr);
OBJECT  *  dial_adr;
```

This function displays a dialog box on the screen.

The box is only drawn, this function does *not* wait for a mouse button click.

The screen is saved before drawing. It can be re-displayed by using the `rs_erasedial` function.

The address of the form to display is passed as the `dial_adr` parameter. This is the value returned by `rs_addrdial`. This form must be part of a resource file loaded by `rsrc_load`.

The address of the box, *not* its index is passed to this function.

This function isn't enough to handle the complete interaction between the user and a form. You should also call `form_do` (GEM AES), and then restore the screen using `rs_erasedial`.

This program loads a resource file containing a dialog box (with an editable field `EDIT`, two buttons `DRIVEA` and `DRIVEB` to change the editable field `EDIT`, four radio buttons `CHOICE1`, `CHOICE2`, `CHOICE3` and `CHOICE4`, and two exit buttons `OK` and `CANCEL`) and an alert string. This program is one of the examples on Disk 2.

```
#include "example.h"
#include <gemlib.h>

main()
{
OBJECT*adr_dial;
int object_no;
int finished;
        /* load resource file */
        if (!rsrc_load("example.rsc"))
        {
        /* check if error */
        printf("Fatal error...\n");
        exit(0);
        }

        /* Address of form called DIAL*/
        adr_dial = rs_addrdial(DIAL);

        /* draw the dialog box */
        rs_drawdial(adr_dial);
        graf_mouse(M_ON, 0);

        restart:
        finished = 0;

        /* clear the editable text field */
        strcpy(rs_addredit(adr_dial, TEXT), "");
        do
        {
        /* give the user control */
        object_no = form_do(adr_dial, TEXT);

        /* de-select the button that caused the exit */
        rs_objunselect(adr_dial, object_no);
        rs_drawobject(adr_dial, object_no);
        /* act depending on exit button */
        switch (object_no)
        {
        case DRIVEA:
             /* put A:\*.* in the text field */
             strcpy(rs_addredit(adr_dial, TEXT), "A:\\*.*");
             /* display the new text */
             rs_drawobject(adr_dial, TEXT);

             break ;
        case DRIVEB:
             strcpy(rs_addredit(adr_dial, TEXT), "B:\\*.*");
             rs_drawobject(adr_dial, TEXT);
             break ;
        case OK:
             finished = 1;
             break ;
        case CANCEL:
             finished = 2;
             break ;
        }
        }
        while (!finished);

        /* if you click on OK, ask for confirmation */
        if (finished == 1)
        if (rs_drawalert(ALERT) == 2)
             goto restart;

        /* remove the form & restore the screen */
        rs_erasedial();

        graf_mouse(M_OFF, 0);

        /* display the result */
        cprintf("Folder selected : %s\n",
                          rs_addredit(adr_dial, TEXT));
        if (rs_objstate(adr_dial, CHOICE1))
        cprintf("Choice 1 and ");
        else
        cprintf("Choice 2 and ");
        if (rs_objstate(adr_dial, CHOICE3))
        cprintf("Choice 3");
        else
        cprintf("Choice 4");
        cprintf(" have been selected\n");

        /* Free memory used by the resource file */
        rsrc_free();
}
```

RS_DRAWOBJECT HiSoft C

```
#include  <gemlib.h>
rs_drawobject(dial_adr,  object_no);
OBJECT  *  dial_adr;
int  object_no;
```

This function re-draws a single object (`object_no`) in the dialog box given by the address `dial_adr`.

For an example of its use see above.

RS_ERASEDIAL HiSoft C

```
rs_erasedial();
```

This parameterless function removes a dialog box that has been drawn with `rs_drawdial`. The screen is re-drawn as it was before the dialog box was drawn.

You must only use this function after calling `rs_drawdial` and similarly if you call `rs_drawdial` you should always call `rs_erasedial`.

See the example above.

RS_OBJSELECT HiSoft C

```
#include  <gemlib.h>
rs_objselect(dial_adr,  object_no);
OBJECT  *  dial_adr;
int  object_no;
```

This function selects object number `object_no` in the form at address `dial_adr`.

The must be re-drawn for it to appear in inverse video.

rs_objunselect, rs_drawobject, rs_drawdial.

RS_OBJSTATE HiSoft C

```
#include  <gemlib.h>
state  =  rs_objstate(dial_adr,  object_no);
OBJECT  *  dial_adr;
int  object_no,  state;
```

This function returns the state of the object `object_no` in the dialog box given by `dial_adr`. The value returned is 1 if the object is selected and 0 if not.

rs_objselect, rs_drawdial.

RS_OBJUNSELECT HiSoft C

```
#include <gemlib.h>
rs_objunselect(dial_adr, object_no);
OBJECT * dial_adr;
int object_no;
```

This function de-selects the object `object_no` within the tree `dial_adr`.

You should re-draw the object so that it appears as normal.

⚠ rs_objselect, rs_drawobject, rs_drawdial.

RS_OBJXYWH HiSoft C

```
#include <gemlib.h>
rs_objunselect(dial_adr, object_no, rectangle);
OBJECT * dial_adr;
int object_no;
short rectangle[4];
```

This function returns the screen co-ordinates of thr object `object_no` within the dialog box `dial_adr`.

The co-ordinates are stored in the array `rectangle` whose elements will contain the x,y co-ordinates of the top left of the object and its width and height.

⚠ rs_drawobject, rs_drawdial, rect_init, rect_point.

RSRC_FREE GEM AES

```
ret= rsrc_free();
int ret;
```

Frees the space used by resources loaded with `rsrc_load`. You should always call this function before terminating a program that calls `rsrc_load`.

If an error occurs the function returns 0.

RSRC_GADDR GEM AES

```
#include  <gemlib.h>
ret=  rsrc_gaddr(object_type,  object_no,  &object_adr);
int  ret,object_type,object_no;
OBJECT  *  object_adr;
```

This function returns in `object_adr` the address of the given object (`object_no`) of the given type (`object_type`) of the loaded resource file.

If an error occurs the function returns 0.

RSRC_LOAD GEM AES

```
ret=  rsrc_load(file_name);
char  *file_name;
int  ret;
```

Load a resource file into memory.

The value returned is 0 if an error occurs, for example if the file is not found or there is insufficient memory to load the file.

⚠ The area where resource files are loaded is fixed during the initialisation of **HiSoft C**. If you are loading a substantial resource file this will be too small. The size of this area can be changed. See **Section 1.4.13**.

RSRC_OBFIX GEM AES

```
#include  <gemlib.h>
rsrc_obfix(tree_adr,  object_no);
int  object_no;
OBJECT  *  tree_adr;
```

Converts the co-ordinates of the object `object_no` in the tree `tree_adr` from character co-ordinates to screen co-ordinates.

RSRC_SADDR GEM AES

```
#include  <gemlib.h>
ret=  rsrc_saddr(object_type,  object_no,  object_adr);
int  ret,object_type,  object_no;
OBJECT  *  object_adr;
```

This function sets the address field of the object `object_no` of type `object_type` to be `object_adr`.

If an error occurs the function returns 0.

RWABS GEMDOS

```
ret= Rwabs(read_write, buffer, sectors,
sector_no,drive_no);
int ret, read_write, sectors, sector_no, drive_no;
char *buffer;
```

Read (if `read_write` = 0) or write (if `read_write` = 1) logical sectors on a disk.

The number of sectors to read or write is given by `sectors`. The i/o starts at sector number `sector_no`.

`drive_no` indicates which drive to read from 0=A, 1=B etc.

SCANF ANSI

```
object_no = scanf(format, arg1, arg2,...);
char *format;
int object_no;
```

The `scanf` function performs formatted input on the file `stdin` (the keyboard by default).

The first argument (`format`) indicates the format of the reading. Extra arguments are needed according to the format specified. Each of these extra arguments is a pointer to a variable. Each value read from the keyboard is stored in the variables given by these pointers.

The value returned by `scanf` is the number of values successfully read from the keyboard and assigned to the variables `arg1,arg2,...`

The `format` parameter is a string indicating how to perform the read. There are three sorts of elements that can make up the format string:

• White space (spaces, tab characters and newlines).

> A sequence of white space characters in the format string causes all white space characters input to be ignored until a non-blank character is entered. Exactly which white space characters are used in the format string is irrelevant.

• Conversion specifications.

> Conversion specifications start with a per-cent sign `%`. The rest of the format of conversion specifications is explained later.

• All other characters.

> Each such character must correspond to the same character in the input. A `%` in the input is represented by `%%` in the format string.

Reading stops when:

- The end of file is reached (if `stdin` has been re-directed). Clearly reading must stop at this point.
- There is a mis-match in the format and the characters read. Input stops as soon as this occurs. The remaining conversion specifications will be ignored.
- The whole format string has been scanned.

The value returned by `scanf` is the number of successful conversions. That is to say the number of values assigned to the variables `arg1`, `arg2`...

If no characters can be read the function returns -1.

The syntax for `format` statements is as follows:

```
%*mlc
```

They consist of the following elements in order.

1. A `%` character.

2. An optional asterisk `*` which indicates that the conversion should be made as usual, but that the value obtained should not be stored in a variable. Thus no argument is 'consumed' by this conversion.

3. A strictly positive decimal number `m` which gives the maximum number of characters to be read during the conversion. This is only used when reading strings of characters (`%s`).

4. An optional letter `l` indicating that the type of a parameter is long. This is used with the conversion characters `d`, `e`, `f`, `g`, `o`, `u` and `x`.

5. A conversion character. This must be present and should be one of `c`, `d`, `e`, `f`, `g`, `h`, `n`, `o`, `s`, `u`, `x`.

The `scanf` specifiers are similar to the `printf` ones. In some cases the format strings can be the same. But this doesn't mean that a string that can be used for output via `printf` can always be used to input the string.

The conversion characters are as follows:

`c` Character.	The corresponding argument must be a pointer to a character, i.e of type `char*`. The first character present in the input is read and stored at the address pointed to by the argument. Blanks are not skipped.
`d` Decimal integer.	The corresponding argument must be a pointer to a integer, i.e. of type `int*`, or `long*` if the long type (`l`) indicator is present. Blanks, if any, are skipped. A + or - sign may be present. Characters are read until a non-valid digit is found. There is no check for overflow. The converted value is stored in the integer pointed to by the argument.

e,f,g Floating point.	Floating point. These three types are equivalent. The corresponding argument must be a pointer to a float or double. As both float and double are double-precision, the long type indicator (l) must be used. Otherwise the conversion will not be done correctly. The format required is `[blanks][sign]digits[.digits][exponent]` 1. Optional blank characters are ignored. 2. An optional + or – . 3. A sequence of digits. 4. An optional decimal point, followed by further digits if present. 5. An optional exponent consisting of a letter e or E followed by an optional sign and sequence of digits. The value calculated from the characters read is stored in the double precision variable pointed to by the corresponding argument.
h short decimal integer.	The corresponding argument must be a pointer to a short integer, i.e. of type short*. Blanks, if any, are skipped. A + or – sign may be present. Characters are read until a non-valid digit is found. There is no check for overflow. The converted value is stored in the integer pointed to by the argument.
o Octal integer.	The corresponding argument must be a pointer to a integer, i.e. of type int*, or long* if the long type (l) indicator is present. Blanks, if any, are skipped. Characters are read until a non-valid octal digit is found (i.e not 0-7). There is no check for overflow. The converted value is stored in the integer pointed to by the argument.
s Character string	The corresponding argument must be a pointer to a character, i.e. of type char*, or array of characters.
x Hexa-decimal integer.	The corresponding argument must be a pointer to an integer, i.e. of type int*, or long* if the long type (l) indicator is present. Blanks, if any, are skipped. Characters are read until a non-valid hex digit is found (i.e not 0-9, a-f or A-F). There is no check for overflow. The converted value is stored in the integer pointed to by the argument.

There are some characters that can be typed which will not be read by scanf because the scanf function has not been asked to read them.

For example, the newline character typed at the end of a line won't be read by `scanf` unless the string finishes with a space.

This can also happen if the format string does not correspond to the user's input. If `scanf` is waiting for a number (`%d`) and the user types digits then the letters won't be read by `scanf`.

All characters not read by `scanf` remain in the input buffer and these characters will be handled by subsequent calls to `scanf`.

This problem can appear particularly if you are trying to read a character using `%c`. When using this conversion character, white space (i.e. spaces and new lines) are not skipped.

If for example, you execute two `scanf("%c",&c);` statements one after another and on the first call you type a character followed by `Return`, then the first character will be stored in `c` as you expected. But when it comes to the second call the character read will be the `Return`, which is probably not what you expected.

There are two ways to get round this problem.

The first is to always call `getchar()` after `scanf` to read the newline character.

The second method is to force the input buffer to be emptied before calling `scanf`. To do this use `fseek(stdin,0,2)`. Don't forget to include the `<stdio.h>` file.

⚠ fscanf, sscanf, sprintf.

SCRP_READ — GEM AES

```
ret= scrp_read(address);
int ret;
char *address;
```

Reads the name of the clipboard directory.

SCRP_WRITE — GEM AES

```
ret= scrp_write(address);
int ret;
char *address;
```

Changes the name of the clipboard directory.

SELECT_MENU HiSoft C

```
state =  select_menu(title);
int  state,  title;
```

Select or de-select a menu title.

The menu must have been created using the `init_menu` function.

`title` is the number of the menu title as returned by `title_menu`. The value returned is the new state of the menu. 1 meaning selected, 0 otherwise.

init_menu, item_menu, **Section 3.4.6**, Help command (select_m).

SETBUF ANSI

```
#include  <stdio.h>
setbuf(fp,buffer);
FILE *fp;
char  buffer[BUFSIZ];
```

This function must be used after a file has been opened with `fopen`, but before any other operation on the file (read or write).

The `buffer` whose address is passed as a parameter is used to replace the default buffer for input-output on this file.

This buffer must be of size `BUFSIZ`.

If the address passed is zero then the i/o on this file is performed without using a buffer, causing a physical i/o operation for each system call.

The first parameter is the file pointer for the file concerned.

SETCOLOR GEMDOS

```
Setcolor(colour_no,  pallete_value);
int  colour_no,  pallete_value;
```

Sets the palette (to `pallete_value`) for the colour `colour_no`.

SETEXC GEMDOS

```
setexc(vector_no,  vector_value);
int  vector_no;
char  *vector_value;
```

Sets the interrupt or exception vector (given by `vector_no` between 0 and 255) to the `vector_value` passed as a parameter.

SETNBUF ANSI

```
#include  <stdio.h>
setnbuf(fp);
FILE  *fp;
```

This function lets you suppress the buffering of a given file.

After calling this function every i/o call for the file given by the file pointer `fp` will cause a physical i/o operation.

SETPALLETE GEMDOS

```
Setpallette(palette_adr);
short  palette[16];
```

Sets the pallette of all 16 colours.

SETPRT GEMDOS

```
n_config  =  Setprt(config);
int  config;
```

Read or write the printer configuration.

If the parameter is -1, then the configuration is read by the function.

Otherwise this parameter is a bitmap giving the new configuration as follows:

bit	**value 0**	**value 1**
0	dot matrix	daisy wheel
1	colour	black and white
2	Atari printer	Epson
5	draft mode	final mode
4	parallel printer	serial
5	continuous stationery	single sheet
15	must be 0	

The function returns the new printer configuration.

SETSCREEN GEMDOS

```
Setscreen(logical_adr,  physical_adr,  resolution);
char  *logical_adr,  *physical_adr;
int  resolution;
```

Modifies the screen addresses and/or the screen resolution.

The parameters are the logical screen address, the physical screen address and the desired resolution (0, 1 or 2).

If a parameter is negative the corresponding parameter is not changed.

SETTIME GEMDOS

```
ret=  settime(date_time);
long  ret,  date_time;
```

Sets the intelligent keyboard controller's idea of the date and time. The same format as `Tgetdate` and `Tgettime` is used.

The value returned is the time that has been set.

SHEL_ENVRN GEM AES

```
shel_envrn(address,name);
char  *address,  *name;
```

Finds the address of an environment variable.

SHEL_FIND GEM AES

```
ret=  shel_find(buffer);
int  ret;
char  *buffer;
```

Searches for a file name using the AES path.

SHEL_READ GEM AES

```
shel_read(application_name,  command_line);
char  *application_name,  *command_line;
```

Reads the name of the running application and its command line.

SIN ANSI

```
ret=  sin(val);
double  ret,val;
```

Calculates the sine of an angle in radians. Both the argument and the result are double reals.

cos, tan.

SINH ANSI

```
ret=  sinh(val);
double  ret,val;
```

Calculates the hyperbolic sine of the angle given as a parameter. Both the argument and the result are double reals.

If the argument is too large for the `sinh` to be calculated then `errno` will indicate this error.

cosh, tanh, errno.

SPRINTF ANSI

```
length  =  sprintf(string,  format,  arg1,  arg2,...);
int  length;
char  *string,  *format;
```

This function is similar to `printf` except that instead of writing to the screen it writes to a string that is passed as a parameter.

This function lets you easily convert from numeric types to ASCII.

The following function converts an integer (`number`) into a string (`string`) containing its hexadecimal representation.

```
void  conv_hex(number,string)
int  number;
char  *string;
{
     sprintf(string,"%#x",number);
}
```

cprintf, fprintf, printf.

SQRT ANSI

```
ret=  sqrt(val);
double  ret,  val;
```

Returns the square root of the number passed as a parameter. Both the parameter and the result are double precision real numbers.

SRAND ANSI

```
srand(value);
unsigned  int  value;
```

Re-seed the random number generator.

⚠ rand.

SQSORT UNIX

```
sqsort(tab,n);
short  *tab;
int  n;
```

This function sorts an array of short integers.

The array of shorts is modified so that they are in increasing order.

⚠ lqsort, dqsort, tqsort.

SSCANF ANSI

```
number  =  sscanf(string,  format,  arg1,  arg2,...);
int  number;
char  *string,  *format;
```

This function is similar to `scanf` except that the characters, instead of being read from the keyboard, are read from a string (`string`) that is passed as a parameter to the `sscanf` function.

⚠ This function may be used to perform ASCII to numeric conversion for all the types used by `scanf`.

The following function converts a string (string) containing a hexadecimal digit into an integer. The converted value is returned by the function. If an error occurs -1 is returned.

```
int  conv_hex(string)
char  *string;
{
int  number;
if  (!sscanf(string,"%x",&number))
     number  =  -1;
return(number);
}
```

cscanf, fscanf, scanf.

STD ANSI

```
#include <stdio.h>
FILE *stdin;                    /* default standard input file*/
FILE *stdout;                   /* default standard output file */
FILE *stderr;                   /* default error file */
FILE *stdaux;                   /* serial i/o file */
FILE *stdprn;                   /* printer file */
```

These identifiers are defined in the file stdio.h.

They represent the files that are always opened when you run your program.

`stdin` is the input file that is used by `scanf`, `getchar`, etc. If you wish to take input from a file rather than from the screen, all you need to do is re-direct to this file.

`stdout` is the output file used by `printf`, `putchar`, etc. If you want the output to go to a file all you need to do is re-direct `stdout`.

`stderr` is the file that is used to write error messages.

`stdaux` corresponds to a file open for read/write to the serial port.

`stdprn` is an opened output file for the printer.

The following program re-directs the standard output (`stdout`) to the file `text.txt` on disk.

```
#include <stdio.h>
FILE *sv_stdout;
FILE *fp;
void main()
{
     /* make a copy of stdout */
     sv_stdout = fdopen(dup(fileno(stdout)), "w");
     /* open the file to replace stdout */
     if (fp = fopen("text.txt", "w"))
     {
          /* assign the new file to stdout */
          dup2(fileno(fp), fileno(stdout));
          printf("This is written to the text file");
          /* restore the normal stdout */
          dup2(fileno(sv_stdout), fileno(stdout));
          printf("This is written to the screen\n");
          fclose(fp);
     }
     else
          printf("Error number %d\n", errno);
          fclose(sv_stdout);
     }
}
```

Writes the contents of some variables to the printer.

```
#include  <stdio.h>
void  main()
{
     int i;
     char  time[10];
     i  =  timer_value();
     strtime(time);
     fprintf(stdprn,  "It  is  %s\n",time);
     fprintf(stdprn,  "Timer  value  :  %d\n",  i);
     fflush(stdprn);
}
```

STOP HiSoft C

```
stop();
```

Stops the execution of the file and returns to the **HiSoft C** editor.

STRCAT ANSI

```
ptr = strcat(string1, string2);
char *ptr, *string1, *string2;
```

This function copies the string `string2` to the end of `string1`. `string1` thus becomes longer including all the characters of `string2`.

The value returned is a pointer to `string1`.

`string1` must correspond to an area large enough to contain all the characters of the two strings.

This program writes the string `qwertyuiopasdfgh`.

```
char  string1[20]  =  "qwertyuiop";
char  string2[8]  =  "asdfgh";
void  main()
{
   puts(strcat(string1,string2));
}
```

strncat.

STRCHR ANSI

```
p  =  strchr(string,  ch);
char  *p,  *string;
char  ch;
```

Searches the string passed as a parameter for the first occurrence of the character `ch`.

The pointer is returned pointing to the place in the string where the character was found.

If the character isn't present in the string, 0 is returned.

strrchr, memccpy.

STRCMP ANSI

```
comparison = strcmp(string1, string2);
char *string1, *string2;
int comparison;
```

This function compares `string1` and `string2`. The two strings are compared character by character. The comparison continues until all the characters have been compared if the strings are equal or until the first time two different strings are found.

`string1` is less than `string2` if the first character that is found to be different is less in `string1` than in `string2`.

The value returned is zero if the strings are equal, positive if `string1` is greater than `string2` or negative if `string1` is less than `string2`.

strncmp, stricmp.

STRCPY ANSI

```
ptr = strcpy(string1, string2);
char *ptr, *string1, *string2;
```

This function copies `string2` to `string1`. The characters that were originally in `string1` are deleted.

The value returned is a pointer to `string1`.

`string1` must point to an area of memory large enough to contain `string2`.

STRCSPN ANSI

```
lng = strcspn(string, chs);
int lng;
char *chs, *string;
```

This function searches for the first character in `string` that is *not* one of the characters in `chs`.

The value returned is the number of characters ignored until a character in `chs` was found.

strspn.

STRDATE HiSoft C

```
date  =  strdate(buffer);
char  buffer[9];
char  *date;
```

This function returns the current date in the form of a string: `"dd/mm/yy"`.

The date is written to the `buffer` that is passed as a parameter. The value returned is equal to the parameter.

```
char  buffer[9];
void  main()
{
    printf("Today  is  %s\n",strdate(buffer));
}
```

strtime.

STRGETFN HiSoft C

```
strgetfn(file_name,  drive,  path,  name,  extension);
char  *file_name;
char  *drive,  *path,  *name,  *extension;
```

This function creates a full file name from a drive, a path, a filename and an extension.

The first parameter is an array of characters where the name will be stored.

The four other parameters are strings containing the drive, directory, name and extension respectively.

The `file_name` array must be big enough!

```
char file_name[60] = "d:\\hc\\examples\\strgetfn.c";
char drive[4], path[40], name[10], ext[4];
void main()
{
     strsplfn(file_name, drive, path, name, ext);
     /* drive is now "d:", path "\hc\examples" */
     /* name is now "strgetfn", ext "c" */
     strgetfn(file_name, "a:", "\\", "hc", "prg");
     /* file_name is now "a:\hc.prg" */
}
```

strsplfn.

STRICMP ANSI

```
comparison  =  stricmp(string1,string2);
int  comparison;
char  *string1,*string2;
```

This function compares two strings. It is similar to `strcmp` except that it ignores differences between upper and lower case letters.

The value returned is negative, zero, or positive depending on whether `string1` is less, equal or greater than `string2` respectively as for `strcmp`.

strcmp, strnicmp, strncmp.

STRLEN ANSI

```
length  =  strlen(string);
int  length;
char  *string;
```

This function returns the length of the string passed as a parameter.

`strlen("abc")` returns 3.

STRLWR UNIX

```
ch  =  strlwr(string);
char *ch, string;
```

This function converts all upper case letters in the string passed as a parameter to lower case.

The value returned is a pointer to a string equal to the value passed as a parameter.

`puts(strlwr("Abc12DE&"));` writes `abc12de&` on the screen

strupr.

STRNCAT ANSI

```
ptr  =  strncat(string1,  string2,  lng);
char  *ptr,  *string1,  *string2;
int  lng;
```

This function copies `string2` to the end of `string1`. Thus `string1` is made longer using characters from `string2`.

A maximum `lng` characters will be added to `string1`.

The value returned is a pointer to `string1`.

`string1` must point to an area of memory that is large enough to contain all the characters of the new string.

This program writes the string `"qwerty"` on the screen

```
char  string1[10]  =  "qwer";
char  string2[6]  =  "tyuio";
void  main()
{
      puts(strncat(string1,  string2,  2));
}
```

strcat.

STRNCMP ANSI

```
comparison  =  strcmp(string1,  string2,  lng);
char  *string1,  *string2;
int  comparison,  lng;
```

This function compares `string1` and `string2`. The two strings are compared until either

(a) `lng` characters have been compared and the strings are equal thus far. 0 is returned in this case.

(b) all the characters have been compared and the strings are equal. Again 0 is returned.

(c) Two different characters are found. If the character from `string1` is less than that from `string2` then a negative integer is returned, otherwise a positive integer is returned.

Thus `strncmp` is like `strcmp` but a maximum of `lng` characters are compared.

strcmp, stricmp.

STRNCPY ANSI

```
ptr = strncpy(string1, string2, lng);
char *ptr, *string1, *string2;
int lng;
```

This function copies up to `lng` characters from `string2` to `string1`. The characters that were originally in `string1` are deleted.

The value returned is a pointer to string1.

If `string2` is shorter than `lng`, null (0) characters are added until a total of `lng` characters has been added.

If `string2` is longer than `lng` only the first `lng` characters are copied and the string is *not* terminated by zero.

`string1` must point to an area of memory that is large enough to receive the new string.

strcpy, memcpy.

STRNICMP ANSI

```
comparison = strnicmp(string1, string2, lng);
int comparison, lng;
char *string1,*string2;
```

This function compares up to the first `lng` characters of the two strings `string1` and `string2`. This is similar to `strncmp` except that the corresponding upper and lower case letters are treated as the same.

The value returned is negative, zero or positive, depending on whether `string1` is less than, equal, or greater than `string2` as for `strncmp`.

strcmp, stricmp, strncmp.

STRREV UNIX

```
ch = strrev(string);
char *ch, *string;
```

This function reverses the order of the characters in the string that is passed as a parameter.

The returned value is a pointer to the same string.

STRSPLFN HiSoft C

```
strsplfn(file_name, drive, path, name, extension);
char  *file_name;
char  *drive, *path, *name, *extension;
```

This function splits a full file name into its individual components.

These components are the drive, the path (directory), the name and the extension.

The first parameter is a string of characters containing the file name to analyse.

The four other parameters are arrays of characters where the drive, path, name and extension will be stored.

The arrays must be big enough!

strgetfn.

STRSPN ANSI

```
lng  =  strspn(string,  chs);
int  lng;
char  *string,  *chs;
```

This finds the number of characters at the start of `string` that are contained in `chs`.

Thus the returned value is the number of characters ignored whilst finding the first character that is not in the string `chs`.

strcspn.

STRTIME HiSoft C

```
time = strtime(buffer);
char buffer[9];
char *time;
```

This function returns the current time in a string of characters of the form `"hh:mm:ss"`.

The time is stored in the buffer passed as a parameter. The returned value is also equal to `buffer`.

```
char buffer[9];
void main()
{
    printf("It is %s\n",strtime(buffer));
}
```

strdate, timer_value.

STRTOK ANSI

```
ch = strtok(string, chs);
char *ch, *string, *chs;
```

Splits a string into a string of tokens.

This function considers the parameter `string` to be made up of tokens separated by one or more characters from the string `chs`.

A sequence of calls to this function returns these tokens one at a time.

The first time the function is called a pointer to the first token found in `string` is returned. To find subsequent tokens, 0 is passed instead of `string`. The function will then return a pointer to the second token, then the third etc.

When the `strtok` function returns 0 this means that no further tokens are present in the string.

The following program writes the words contained in the string `"aaa bbb ccc ddd"` on separate lines.

```
char *string = "aaa bbb ccc ddd";
char *ch;
void main()
{
    ch = strtok(string," ");
    do
        puts(ch);
    while(ch = strtok(0, " "));
}
```

STRTOL UNIX

```
n = strtol(string, &end_pos, base);
int n, base;
char *string, *end_pos;
```

This function converts a string of characters into a long integer and returns this as the value of the function.

The `string` must contain only valid digits from 0-9, a-z and A-Z to form a number in the given `base` from 2 to 36.

The function ignores blanks at the start of the string and will note an initial + or - sign. It stops, however, when an illegal character is found.

When it returns `end_pos` points to the first illegal character (or the null at the end) of the string.

`strtol("1000",&p,16)` returns the value 4096 = 1000 Hex.

sscanf.

STRUPR UNIX

```
ch = strupr(string);
char *ch, string;
```

This function converts all the lower case letters in the string passed as a parameter into upper case.

The value returned is a pointer to the string.

`puts(strlwr("Abc12DE&"))` writes `ABC12DE&` on the screen.

strlwr.

SUPER GEMDOS

```
old_pointer = Super(stack_ptr);
char *old_pointer, *stack_ptr;
```

Change to 68000 supervisor or user mode.

If you are about to enter supervisor mode, `stack_ptr` is the value to use as the supervisor stack pointer. The pointer returned is the current user stack pointer.

When returning to user mode, `stack_ptr` is the value to be used for the user mode stack.

SUPEXEC GEMDOS

```
ret=  Supexec(routine_adr);
long  ret;
char  *routine_adr;
```

Executes an assembly language routine in supervisor mode. The value returned by the function is the value returned by the routine.

SVERSION GEMDOS

```
version_no =  Sversion();
int  version_no;
```

Returns the GEMDOS version number.

TAN ANSI

```
ret=  tan(val);
double  ret,val;
```

Calculates the tangent of the angle (in radians) that is passed as a parameter. The argument and the return value are both double reals.

⚠ sin, cos.

TANH ANSI

```
ret=  tanh(val);
double  ret,val;
```

Calculates the hyperbolic tangent of the argument passed as a parameter.

The argument and the return value are both double reals.

⚠ sinh, cosh, errno.

TELL — UNIX

```
pos  =  tell(nf);
int  pos;
int  nf;
```

Returns the current file position in the UNIX file `nf` (opened with `open`) where the next byte will be read or written.

⚠ This position may be changed using `lseek`.

TEXT_BOX — HiSoft C

```
no_object  =  text_box(box_no,  x,  y,  text);
int no_object,  box_no,  x,  y;
char  *text;
```

Adds an object of type text into a dialog box created by `init_box`.

See **Section 3.3.4** for a description of this function.

TGETDATE — GEMDOS

```
date  =  Tgetdate();
int  date;
```

Returns the current date in the following format:

bits 0 to 4	Day (0-31)
bits 5 to 8	Month (1-12)
bits 9 to 15	Year (0-127) 0=1980, 1=1981...

TGETTIME — GEMDOS

```
time  =  Tgettime();
int  time;
```

Returns the current time in the following format:

bits 0 to 4	Seconds (0-29) 0=0 sec, 1=2 sec etc
bits 5 to 10	Minutes (0-59)
bits 11 to 15	Hours (0-11)

TICKCAL GEMDOS

```
base = Tickcal();
int base;
```

Returns the clock tick interval in milliseconds.

TIMER_VALUE HiSoft C

```
value = timer_value();
int value;
```

This function returns the video clock frequency in Hertz. 60 (colour monitor) or 70 (monochrome monitor).

TITLE_MENU HiSoft C

```
no_title = title_menu(title_name);
int no_title;
char *title_name;
```

Adds a title bar to a menu. The menu must have been created with `init_menu`.

See **Section 3.4.3** for the description of this function.

TOASCII ANSI

```
new_ch = toascii(ch);
int new_ch, ch;
```

This function converts a character (between 0 and 255) to ASCII (code<128) by removing the top bit.

The function returns the new character code.

TOLOWER ANSI

```
nch = tolower(ch);
int nch, ch;
```

This function converts an upper case letter (passed as an integer) to lower case. If the character isn't an upper case letter then the value is returned unchanged.

⚠ toupper, islower.

TOUPPER ANSI

```
nch  =  toupper(ch);
int  nch,ch;
```

This function converts a lower case letter (passed as an integer) to upper case. If the character isn't a lower case letter then the value is returned unchanged.

tolower, isupper.

TQSORT ANSI

```
tqsort(arr,n);
char  *arr[xx];
int  n;
```

This function formats an array (`arr`) of `n` pointers to string into increasing order.

The array of pointers is modified in situ.

It is not the values of the pointers, but the values of the strings that are used for the sort.

The following program sorts an array of three strings into alphabetical order, and writes them in that order.

The array `arr` is initialised to contain three pointers. The first points to the string `"zzzz"`, the second to `"aa"` and the third to `"zer"`. After the sort, the order of the pointers is modified. The first points to "aa", the second to `"zer"` and the third to `"zzzz"`. The strings themselves do not move.

```
char  *tab[3]={"zzzz","aa",  "zer"};
void  main()
{
     tsqort(arr,3);
     puts(arr[0]);
     puts(arr[1]);
     puts(arr[2]);
}
```

lqsort, sqsort, dqsort.

TRACE_OFF — HiSoft C

```
trace_off();
```

This function switches off single step mode. At the same, the display of variables is suppressed.

⚠ trace_on, var_on, var_off, **Section 1.4.3.**

TRACE_ON — HiSoft C

```
trace_on();
```

This function switches on single step mode.

⚠ trace_off, var_on, var_off, **Section 1.4.3.**

TSETDATE — GEMDOS

```
ret=  Tsetdate(date);
int  date,  ret;
```

Sets the date using the following format

bits 0 to 4	Day (0-31)
bits 5 to 8	Month (1-12)
bits 9 to 15	Year (0-127) 0=1980, 1=1981...

The value returned is 0 if the date is valid.

TSETTIME — GEMDOS

```
ret=  Tsettime(time);
int  time,  ret;
```

Sets the time, using the following format:

bits 0 to 4	Seconds (0-29) 0=0 sec, 1=2 sec etc
bits 5 to 10	Minutes (0-59)
bits 11 to 15	Hours (0-11)

UNGETC — ANSI

```
#include  <stdio.h>
ret=  ungetc(ch,  fp);
int ret,  ch;
FILE  *fp;
```

This function cancels the affect of the last call to the `fgetc` function. The next character to be read from the file `fp` will be `ch`.

This function may not be called twice for a file between two reads of that file. Only one character can be 'put back'.

UNLINK — UNIX

```
ret=  unlink(name);
char  *name;
int ret;
```

Deletes the disk file called `name` that is passed as a parameter.

This function returns 0 if the operation was successful, or a non-zero if an error occurred in which case `errno` will indicate the source of the error.

remove, Ddelete, errno.

V_ARC — GEM VDI

```
v_arc(vdi_handle,  x,  y,  radius,  angle1,  angle2);
int  vdi_handle,  x,  y,  radius,  angle1,  angle2;
```

Draws a circular arc of centre (x,y) of the given `radius` between the angles `angle1` and `angle2`. The angles should be between 0 and 3600 (tenths of degrees).

VAR_OFF — HiSoft C

```
var_off();
```

Stops the display of variables during execution.

trace_on, **Section 1.4.4.**

VAR_ON — HiSoft C

```
var_on();
```

Starts the display of variables on the screen during execution. Trace mode must already be active.

⚠ trace_on, **Section 1.4.4.**

V_BAR — GEM VDI

```
v_bar(vdi_handle, arr_xy);
int vdi_handle;
short arr_xy[4];
```

Draw a rectangle with the current attributes.

`arr_xy` contains the (x,y) co-ordinates of two opposite corners of the rectangle.

V_CIRCLE — GEM VDI

```
v_circle(vdi_handle, x, y, radius);
int vdi_handle, x, y, radius;
```

Draws a circle centred at (x,y) with the given `radius`.

V_CLRWK — GEM VDI

```
v_clrwk(vdi_handle);
int vdi_handle;
```

Clears the screen.

V_CLSVWK — GEM VDI

```
v_clsvwk(vdi_handle);
```

Close the virtual workstation opened using `v_opnvwk`.

V_CONTOURFILL — GEM VDI

```
v_contourfill(vdi_handle, x, y, fill_colour);
int vdi_handle, x, y, fill_colour;
```

Performs a seed fill in the `fill_colour` starting at the co-ordinates (x,y).

V_CURDOWN GEM VDI

```
v_curdown(vdi_handle);
int  vdi_handle;
```

Moves the text cursor one line down.

v_enter_cur.

V_CURHOME GEM VDI

```
v_curhome(vdi_handle);
int  vdi_handle;
```

Moves the text cursor to the top left of the screen.

v_enter_cur.

V_CURLEFT GEM VDI

```
v_curleft(vdi_handle);
int  vdi_handle;
```

Move the cursor left one character.

v_enter_cur.

V_CURRIGHT GEM VDI

```
v_curright(vdi_handle);
int  vdi_handle;
```

Move the text cursor right one character.

v_enter_cur.

V_CURTEXT GEM VDI

```
v_cur(vdi_handle, string);
int  vdi_handle;
char  *string
```

Display a string of text at the current cursor position.

⚠ v_enter_cur.

V_CURUP GEM VDI

```
v_cur(vdi_handle);
int  vdi_handle;
```

Move the text cursor up one line.

⚠ v_enter_cur.

V_DSPCUR GEM VDI

```
v_dspcur(vdi_handle, x, y);
int  vdi_handle, x, y;
```

Display the mouse cursor at position (x,y).

V_EEOL GEM VDI

```
v_eeol(vdi_handle);
int  vdi_handle;
```

Clear the screen from the current cursor position to the end of the current line.

⚠ v_enter_cur.

V_EEOS GEM VDI

```
v_eeos(vdi_handle);
int  vdi_handle;
```

Clears the text screen starting at the current cursor position.

⚠ v_enter_cur.

V_ELLARC GEM VDI

```
v_ellarc(vdi_handle, x, y, x_radius, y_radius,
                    start_angle, finish_angle);
int vdi_handle, x, y, x_radius, y_radius,
                    start_angle, finish_angle;
```

Draws an elliptical arc based on centre (x,y) and radii x_radius and y_radius between start_angle and finish_angle. The two angles are measured in tenths of degrees (that is between 0 and 3600).

V_ELLIPSE GEM VDI

```
v_ellipse(vdi_handle, x, y, x_radius, y_radius);
int vdi_handle, x, y, x_radius, y_radius;
```

Draws an ellipse with centre (x,y), x radius, x_radius and y radius, y_radius.

V_ELLPIE GEM VDI

```
v_ellpie(vdi_handle, x, y, x_radius, y_radius,
                          start_angle, finish_angle);
int vdi_handle, x, y, x_radius, y_radius,
                          start_angle, finish_angle;
```

Draws an elliptical pie slice given the centre (x,y), radius, start_angle and finish_angle. The angles are measured in tenths of degrees between 0 and 3600.

V_ENTER_CUR GEM VDI

```
v_enter_cur(vdi_handle);
int  vdi_handle;
```

Enter text mode. The screen is cleared and a flashing cursor appears.

VEX_BUTV GEM VDI

```
vex_butv(vdi_handle, new_adr, old_adr);
int  vdi_handle;
char *new_adr,  *old_adr;
```

Modifies the mouse interrupt vector.

VEX_CURV GEM VDI

```
vex_curv(vdi_handle, new_adr, old_adr);
int vdi_handle;
char *new_adr, *old_adr;
```

Modifies the end of mouse cursor drawing vector.

V_EXIT_CUR GEM VDI

```
v_exit_cur(vdi_handle);
int vdi_handle;
```

Leaves text mode. The screen is cleared and the flashing cursor disappears.

VEX_MOTV GEM VDI

```
vex_motv(vdi_handle, new_adr, old_adr);
int vdi_handle;
char *new_adr, *old_adr;
```

Modifies the mouse movement vector.

VEX_TIMV GEM VDI

```
vex_timv(vdi_handle, new_adr, old_adr);
int vdi_handle;
char *new_adr, *old_adr;
```

Modifies the timer interrupt vector.

V_FILLAREA GEM VDI

```
v_fillarea(vdi_handle, n, arr_xy);
int vdi_handle, n;
short arr_xy[n*2];
```

Draws a filled polygon containing `n` points. The array `arr_xy` contains the co-ordinates of all the points in the polygon (x0,y0, x1,y1,...).

V_GET_PIXEL GEM VDI

```
v_get_pixel(vdi_handle, x, y, &rgb_colour, &colour_no);
int vdi_handle, x, y;
short rgb_colour, colour_no;
```

Returns the colour of the point `(x,y)`.

V_GTEXT GEM VDI

```
v_gtext(vdi_handle, x, y, string);
int vdi_handle, x, y;
char *string;
```

Displays a `string` of graphics text characters. The position of the first character is given by (x,y).

V_HIDE_C GEM VDI

```
v_hide_c(vdi_handle);
int vdi_handle;
```

Hides the mouse cursor.

V_JUSTIFIED GEM VDI

```
v_justified(vdi_handle, x, y, string,
    length, word_spacing, char_spacing);
int vdi_handle, x, y, length, word_spacing, char_spacing;
char *string;
```

Display a justified graphics text string starting at co-ordinates (x,y) in a width of `length` pixels.

If `word_spacing = 1` then spaces may be added between words and/or if `char_spacing = 1` then space may be added between characters.

V_OPNVWK GEM VDI

```
v_opnvwk(arr1, &vdi_handle, arr2);
short vdi_handle, arr1[11], arr2[57];
```

Initialise a virtual workstation for GEMVDI.

It is not necessary to call this function when using **HiSoft C** because **HiSoft C** has already done this.

To find the `vdi_handle` use `graf_handle`.

V_PIESLICE GEM VDI

```
v_pieslice(vdi_handle, x, y, radius, start_angle, finish_angle);
int vdi_handle, x, y, radius, start_angle, finish_angle;
```

Draws a circular pie slice centred on (x,y) of the specified `radius`, between the `start_angle` and the `finish_angle`. The angles are expressed in tenths of a degree.

V_PLINE GEM VDI

```
v_pline(vdi_handle, n, arr_xy);
int vdi_handle, n;
short arr_xy[n*2];
```

Draws a set of straight lines joining the n points stored in the array arr_xy (x0, y0, x1, y1, x2, y2...).

V_MARKER GEM VDI

```
v_pmarker(vdi_handle, n, arr_xy);
int vdi_handle, n;
short arr_xy[n*2];
```

Draw n markers whose co-ordinates are stored in the array arr_xy (x0,y0,x1,y1...).

VQ_CHCELLS GEM VDI

```
vq_chcells(vdi_handle, &screen_height, &screen_width);
int vdi_handle;
short screen_height, screen_width;
```

Returns the size of the screen in characters.

VQ_COLOR GEM VDI

```
vq_color(vdi_handle, colour_no, value_type, rgb_arr);
int vdi_handle, colour_no, value_type;
short rgb_arr[3];
```

Returns the representation of a colour in rgb units.

VQ_CURADDRESS GEM VDI

```
vq_curaddress(vdi_handle, &row, &column);
int vdi_handle;
short row, column;
```

Return the current text cursor position.

v_enter_cur.

VQ_EXTND GEM VDI

```
vq_extnd(vdi_handle, which, tab);
int vdi_handle, which;
short tab[57];
```

Returns information on the given virtual workstation.

VQ_INMODE GEM VDI

```
vq_inmode(vdi_handle, dev_type, &input_mode);
int vdi_handle, dev_type;
short input_mode;
```

Returns the current input mode of a given device.

VQF_ATTRIBUTES GEM VDI

```
vqf_attributes(vdi_handle,fill_type);
int vdi_handle;
short fill_type[4];
```

Returns the current fill area attributes (type, colour, style and writing mode in that order).

VQ_KEY_S GEM VDI

```
vq_key_s(vdi_handle, &key_state);
int vdi_handle;
short key_state;
```

Returns the state of the control keys: right Shift (bit 0), left Shift (bit 1), Control (bit 2), and Alternate (bit 3).

The corresponding bit is 1 if the key is down.

VQL_ATTRIBUTES GEM VDI

```
vql_attributes(vdi_handle,line_type);
int vdi_handle;
short line_type[4];
```

Returns in the array `line_type` information on the current line attributes (type, colour, writing mode and line width in that order).

VQM_ATTRIBUTES GEM VDI

```
vqm_attributes(vdi_handle,marker_type);
int  vdi_handle;
short  marker_type[4];
```

Returns in the array `marker_type` the current marker attributes (type, colour, writing mode and height in that order).

VQ_MOUSE GEM VDI

```
vq_mouse(vdi_handle,  &button,  &x,  &y);
int  vdi_handle;
short  button,  x,  y;
```

Returns the state of the mouse buttons.

VQT_ATTRIBUTES GEM VDI

```
vqt_attributes(vdi_handle,text_type);
int  vdi_handle;
short  text_type[10];
```

Returns in the array `text_type` information on the current text attributes (font, colour, angle, horizontal alignment, vertical alignment, writing mode, character width, character height, cell width, cell height in that order).

VQT_EXTENT GEM VDI

```
vqt_extent(vdi_handle,  string,  extent);
int  vdi_handle;
short  extent[8];
char  *string;
```

Returns in the `extent` array (x0, y0, x1,y1, x2,y2, x3,y3) the co-ordinates of a box that would surround the text of the `string` passed as a parameter.

VQT_FONTINFO GEM VDI

```
vqt_fontinfo(vdi_handle,  &first_char,  &last_char,
    distances,  &maxwidth,  effects);
int  vdi_handle;
short  first_char,  last_char,  maxwidth;
short  distances[5],  effects[3];
```

Returns information on the current text font.

VQT_NAME GEM VDI

```
vqt_name(vdi_handle, font_no, font_name);
int vdi_handle, font_no;
char *font_name;
```

Returns the name of the font whose index is passed as a parameter.

VQT_WIDTH GEM VDI

```
vqt_width(vdi_handle, character, &cell_width,
    &left_offset, &right_offset);
int vdi_handle, character;
short cell_width, left_offset, right_offset;
```

Returns the `cell_width`, and `left_offset` and `right_offset` of the given `character`.

V_RBOX GEM VDI

```
v_rbox(vdi_handle, arr_xy);
int vdi_handle;
short arr_xy[4];
```

Draws a rectangle with rounded corners using the line attributes. `arr_xy` contains the two opposite corners of the rectangle.

V_RFBOX GEM VDI

```
v_rfbox(vdi_handle, arr_xy);
int vdi_handle;
short arr_xy[4];
```

Draws a filled rectangle with rounded corners using the fill attributes. `arr_xy` contains the two opposite corners of the rectangle.

V_RMCUR GEM VDI

```
v_rmcur(vdi_handle);
int vdi_handle;
```

Removes the mouse cursor.

VRO_CPYFM GEM VDI

```
#include  <gemlib.h>
vro_cpyfm(vdi_handle,  operation,  arr_coords,
     mfdb_source,  mfdb_destination);
int vdi_handle,  operator;
short  arr_coords[8];
FDB  *mfdb_source,  *mfdb_destination;
```

Copies a block of memory from a source area (`mfdb_source`) to a destination area (`mfdb_destination`) performing a raster `operation` on the block.

VR_RECFL GEM VDI

```
vr_recfl(vdi_handle,  arr_xy);
int vdi_handle;
short  arr_xy[4];
```

Draws a filled rectangle without a border using the fill attributes. `arr_xy` contains the two opposite corners of the rectangle.

VRT_CPYFM GEM VDI

```
#include  <gemlib.h>
vro_cpyfm(vdi_handle,  operator,  arr_coords,
     mfdb_source,  mfdb_destination,  colours);
int  vdi_handle,  operator;
short  arr_coords[8],  colours[2];
FDB *mfdb_source,  *mfdb_destination;
```

Similar to the `vro_cpyfm` function except that the source area is considered to be monochrome and is given the colours in the array `colours` when copied.

VR_TRNFM GEM VDI

```
vr_trnfm(vdi_handle,  mfdb_source,  mfdb_dest);
int vdi_handle;
FDB  *mfdb_source,  *mfdb_dest;
```

Copies a memory area (described by `mfdb_source`) in standard format to a destination area (described by `mfdb_dest`) in device dependent format.

V_RVOFF GEM VDI

```
v_rvoff(vdi_handle);
int  vdi_handle;
```

Cancels the effect of `v_rvon`.

V_RVON GEM VDI

```
v_rvon(vdi_handle);
int  vdi_handle;
```

After a call to this function, all text written with `v_curtext` will be written in inverse video (white on black).

VSC_FORM GEM VDI

```
vsc_form(vdi_handle,  mouse_form);
int  vdi_handle;
mouse_form[37];
```

Re-defines the mouse form. The elements of the array are as follows:

0	X co-ordinate of hot spot
1	Y co-ordinate of hot spot
2	always 1
3	mask colour (normally 0)
4	data colour (normally 1)
5 to 20	mask
21 to 37	data

VS_CLIP GEM VDI

```
vs_clip(vdi_handle,  clip,  clip_arr);
int  vdi_handle,  clip;
short  clip_arr[4];
```

Enables (`clip=1`) or disables (`clip=0`) clipping within the VDI rectangle `clip_arr` (x0,y0,x1,y1).

VS_COLOR GEM VDI

```
vs_color(vdi_handle,  colour_no,  rgb_values);
int vdi_handle,  colour_no;
short  rgb_values[3];
```

Sets the palette for a colour `colour_no` to be the red, green & blue values (between 0 and 1000) in the array `rgb_values`.

VS_CURADDRESS GEM VDI

```
vs_curaddress(vdi_handle, row, column);
int vdi_handle, row, column;
```

Positions the text cursor at position given by `row, column`.

v_enter_cur.

VSF_COLOR GEM VDI

```
vsf_color(vdi_handle, colour_no);
int vdi_handle, colour_no;
```

Selects the colour used for filling areas.

VSF_INTERIOR GEM VDI

```
vsf_interior(vdi_handle, fill_type);
int vdi_handle, fill_type;
```

Selects the type of fill depending on the value of fill_type:

0	hollow fill
1	fill with the colour given by `vsf_interior`
2	use a pattern defined by `vsf_style`
3	use a hatch defined by `vsf_style`
4	use a user defined fill area defined by `vsf_updat`

VSF_PERIMETER GEM VDI

```
vsf_perimeter(vdi_handle, perimeter);
int vdi_handle, perimeter;
```

Indicates if GEM is to draw a perimeter round filled objects (if `perimeter=1`) or not (`perimeter=0`).

VSF_STYLE GEM VDI

```
vsf_style(vdi_handle, style);
int vdi_handle, style;
```

Selects a fill pattern style (1 to 24) or hatch (1 to 12).

VSF_UDPAT GEM VDI

```
vsf_udpat(vdi_handle, image, planes);
int vdi_handle, planes;
short *image;
```

Sets up a user defined fill pattern.

`planes` indicates the number of planes in the design (16 =1 plane, 32= 2 planes, 64 = 4 planes).

`image` contains the bit pattern for the new pattern and should contain the same number of words as the value of planes.

V_SHOW_C GEM VDI

```
v_show_c(vdi_handle, count);
int vdi_handle, count;
```

Makes the mouse appear if `count` is zero.

VSL_COLOR GEM VDI

```
vsf_color(vdi_handle, colour_no);
int vdi_handle, colour_no;
```

Select the colour that is used for drawing lines.

VSL_END GEM VDI

```
vsl_ends(vdi_handle, start_type, end_type);
int vdi_handle, start_type, end_type;
```

Sets the style used for the ends of lines: 0 = normal, 1 = arrow, 2= rounded.

VSL_TYPE GEM VDI

```
vsl_type(vdi_handle, line_type);
int vdi_handle, line_type;
```

Sets the line style (0 to 7) that is used in drawing lines.

VSL_UDSTY GEM VDI

```
vsl_udsty(vdi_handle, style);
int vdi_handle, style;
```

Defines the user defined line style.

VSL_WIDTH GEM VDI

```
vsl_udsty(vdi_handle, width);
int vdi_handle, width;
```

Defines the line width to be `width` (between 1 and 39).

VSM_COLOR GEM VDI

```
vsf_color(vdi_handle, colour_no);
int vdi_handle, colour_no;
```

Selects the marker colour.

VSM_TYPE GEM VDI

```
vsm_type(vdi_handle, type);
int vdi_handle, type;
```

Selects the type of marker to be used (1 to 6).

VST_ALIGNMENT GEM VDI

```
vst_alignment(vdi_handle, horiz_align, vert_align,
     &ret_horizontal, &ret_vertical);
int vdi_handle, horiz_align, vert_align;
sort ret_horizontal, ret_vertical;
```

Defines the horizontal alignment (in `horiz_align` from 0 to 2) and vertical alignment (in `vert_align` from 0 to 5) for text.

The values returned in `ret_horizontal` and `ret_vertical` are the values set.

VST_COLOR GEM VDI

```
vsf_color(vdi_handle, colour_no);
int vdi_handle, colour_no;
```

Selects the text colour.

VST_EFFECTS GEM VDI

```
vst_effects(vdi_handle, effect);
int vdi_handle, effect;
```

Selects the effects to apply to text using the bitmap `effect`:

bit	**effect**
0	bold
1	light (greyed)
2	italic
3	underlined
4	outlined
5	shadowed

VST_FONT GEM VDI

```
vst_font(vdi_handle, font_no);
int vdi_handle, font_no;
```

Selects which font to use (`font_no`).

VST_HEIGHT GEM VDI

```
vst_height(vdi_handle, requested_height, &char_height,
&char_width,
    &cell_height, &cell_width);
int vdi_handle, requested_height;
short char_height, char_width, cell_height, cell_width;
```

Attempts to set the height of characters to `requested_height`.

This returns the size actually selected which is never bigger than that which was requested.

VST_LOAD_FONTS GEM VDI

```
fonts = vst_load_fonts(vdi_handle, reserved);
int vdi_handle, reserved, fonts;
```

Load the fonts indicated in ASSIGN.SYS. Requires GDOS for use.

VST_POINT GEM VDI

```
height = vst_point(vdi_handle, points, &char_height,
&char_width,
    &cell_height, &cell_width);
int vdi_handle, points, height;
short char_height, char_width, cell_height, cell_width;
```

Selects the height of a character using points (1/72th of an inch) as the units.

VST_ROTATION GEM VDI

```
vst_rotation(vdi_handle, angle);
int vdi_handle, angle;
```

Specifies the angle of rotation of text characters. Possible values are : 0=normal, 900, 1800 and 2700.

VST_UNLOAD_FONTS GEM VDI

```
vst_unload_fonts(vdi_handle, reserved);
int vdi_handle, reserved;
```

Frees the space used by loaded fonts.

VSWR_MODE GEM VDI

```
mode_selected = vswr_mode(vdi_handle, writing_mode);
int vdi_handle, writing_mode, mode_selected;
```

Selects the writing mode (1=normal, 2=transparent, 3=XOR, 4=inverse transparent).

This function returns the value that has been selected.

VSYNC GEMDOS

```
Vsync();
```

Waits until the vertical screen interrupt.

WIND_CALC GEM AES

```
ret= wind_calc(interior, attributes,
    x1, y1, w1, h1,
    &x2, &y2, &w2, &h2);
int ret, interior, attributes, x1, y1, w1, h1;
short x2, y2, w2,h2;
```

Calculates the size of a window needed, returned in (x2, y2, w2, h2), to give a workspace size of (x1, y1, w1, h1) if interior = 0. Or the size of the work area given the border size if interior =1.

This doesn't apply to a particular window, but is useful for windows in general.

attributes gives which 'gadgets' a window has (see **Section 3.2.3**).

WIND_CLOSE GEM AES

```
ret= wind_close(window_no);
int ret, window_no;
```

Close the window (window_no) which has been opened with wind_open.

The value returned is zero if an error occurs.

WIND_CREATE GEM AES

```
window_no = wind_create(attributes, x, y, w, h);
int window_no, attributes, x, y, w, h;
```

Initialises a window without opening it.

The attributes are the same as for open_window.

x,y,w,h are the co-ordinates of the window.

The value returned is the window's handle, or negative if the window cannot be created.

WIND_DELETE GEM AES

```
ret= wind_delete(window_no);
int ret, window_no;
```

Deletes a window (window_no) after it has been closed.

The value returned is zero if an error occurred.

WIND_FIND GEM AES

```
window_no  =  wind_find(x,y);
int  window_no,  x,  y;
```

Returns the handle of the window under co-ordinate position (x,y). The Desktop is window number 0.

WIND_GET GEM AES

```
#include  <gemlib.h>
ret=  wind_get(window_no,  type_info,  &x,  &y,  &w,  &h);
int  ret,  window_no,  type_info;
short  x,  y,  w,  h;
```

Returns in the parameters (x, y, w, h) information about the window depending on the value of `type_info`:

WF_WXYWH	co-ordinates of the work area
WF_CXYWH	window co-ordinates
WF_PXYWH	previous co-ordinates of the window
WF_FXYWH	maximum co-ordinates of the window
WF_HSLIDE	position of the horizontal slider (1 to 1000) in x
WF_VSLIDE	position of the vertical slider (1 to 1000) in x
WF_TOP	handle of the top window in x
WF_FIRSTXYWH	co-ordinates of the first clipping rectangle
WF_NEXTXYWH	co-ordinates of subsequent clipping rectangles
WF_HSLISIZ	size of the horizontal slider (1 to 1000) in x
WF_VSLISIZ	size of the vertical slider (1 to 1000) in x

The value returned by this function is 0 if an error occurs.

WIND_OPEN GEM AES

```
ret=  wind_open(window_no,  x,  y,  w,  h);
int  ret,  window_no,  x,  y,  w,  h;
```

Opens a window that has been created with `wind_create`. The co-ordinates should be the same as, or smaller than the co-ordinates given to `wind_create`.

WIND_SET GEM AES

```
#include  <gemlib.h>
ret=  wind_set(window_no,  type_info,  x,  y,  w,  h);
int ret, window_no, type_info;
int x, y, w, h;
```

Sets, using the parameters (x, y, w, h), the attributes of the window window_no depending on the value of type_info:

WF_KIND	the window attributes
WF_NAME	the window title (x top 16 bits, y bottom 16 bits)
WF_INFO	the window information line (x top 16 bits, y bottom 16 bits)
WF_CXYWH	window co-ordinates
WF_HSLIDE	position of the horizontal slider (1 to 1000) in x
WF_VSLIDE	position of the vertical slider (1 to 1000) in x
WF_TOP	handle of the top window in x
WF_HSLISIZ	size of the horizontal slider (1 to 1000) in x
WF_VSLISIZ	size of the vertical slider (1 to 1000) in x

WIND_UPDATE GEM AES

```
#include  <gemlib.h>
ret=  wind_update(param);
int  ret,  param.
```

Permit or ban certain of GEM's automatic control of the system depending on the value of param:

BEG_UPDATE	The application is writing to the screen. GEM won't let the user pull down menus or 'play' with windows.
END_UPDATE	Cancel the effect of BEG_UPDATE. Menus may be activated once more.
BEG_MCTRL	The application is about to take control of the mouse.
END_MCTRL	Return mouse control to GEM.

The value returned by this function is 0 if an error occurs.

WRITE UNIX

```
num  =  write(nf,  buffer,  bytes);
int  num,  nf,  bytes;
char  *buffer;
```

Writes bytes to the file number nf that has been opened using open.

buffer is the address from which the bytes are read from the file.

`bytes` is the number of bytes to write to the file.

`num` is the number of bytes that were actually written to the file. Generally this will be the same as `bytes`. However `num` may be less than this if an error occurs (the disk is full for example).

If the value returned by the function is -1 or is less than `bytes` an error has occurred. In this case the variable `errno` will contain an indication of which error.

open, read, errno.

XBIOS — GEMDOS

```
ret= xbios(no, arg1, arg2...);
int no;
long ret,arg1,arg2,...
```

Execute an XBIOS function using TRAP #14.

`no` is the function number.

`ret` is the value returned by the XBIOS function.

`arg1, arg2`... are the parameters for the particular XBIOS function.

bios, gemdos.

XBTIMER — GEMDOS

```
no_it = Xbtimer(timer_no, control_reg, data_reg,
routine_adr);
int no_it, timer_no, control_reg, data_reg;
char *routine_adr;
```

Initialises a 68901 timer.

`timer_no` (0-3) indicates which timer is to be initialised (A, B, C or D).

`control_reg` and `data_reg` are the values (bytes) to write to the control and data registers of the timer.

Finally the interrupt routine for each timer event is passed as the `routine_adr` parameter.

Appendix A . Exercise Answers

Exercise 1

```
main()
{
        printf("he\nllo ");
        printf("How are\nyou ?\n");
        printf("press a key\n");
        evnt_keybd();
}
```

Exercise 3

```
char ch1, ch2;
main()
{
        ch1 = evnt_keybd();
        ch2 = evnt_keybd();
        putchar(ch1);
        putchar(ch2);
        evnt_keybd();
}
```

Notes :

• The variables `ch1` and `ch2` could be declared as `char`, `short` or `int` - it does not matter.

• More than one variable can be declared in the same statement by separating them with commas.

Exercise 4

```
int v1, v2, v3;
main()
{
printf("Here are three values : %d, %d and %d\n",v1,v2,v3);
evnt_keybd();
}
```

There is nothing complicated here. The `evnt_keybd()` call just waits for a key to be pressed so you can admire the results of the program.

Exercise 5

```
int num;
main()
{
        num = Random ();
        if (num%2 == 1)
        {
                printf("The number %d is odd\n", n);
                num = num + 1;
        }
        else
                printf("The number %d is even\n", n);
        evnt_keybd();
}
```

If the number `num` is odd, we write a message to this effect and add one to this number.

If `num` is already even, we just need to write a message on the screen.

Exercise 6

```
int i;
main()
{
        i = 20;
        while (i)
        {
                printf("%d has square %d\n", i, i*i);
                i--;
        }
        evnt_keybd();
        }
```

There's a bizarre bit in this program, `i--`. This is actually very simple. These three characters subtract one from `i`. This is equivalent to the instruction:

```
        i = i - 1;
```

But `i--`, takes less time to write and can execute quicker.

There's a similar trick to add one to a variable. For example to add one to the variable `sum` we can write

```
        sum = sum + 1;
```

or equally well :

```
        sum++;
```

There's another trick in this program. A condition is either true or false. False is represented by zero. True is represented by any other value. Thus 0 is taken as false and any other value as true.

In our example the condition in the statement `while (i)` is therefore true whilst the variable is not zero. The set of two instructions are thus executed until `i` is zero. When `i` becomes 0 the program stops.

Exercise 7

```
int i;
main()
{
        for(i = 20; i>0 ; i--)
                printf("%d squared is %d\n", i, i*i);
        evnt_keybd();
}
```

The variable `i` varies between 20 and 1 in the loop.

`i--` is used to subtract 1 from the variable `i`.

Exercise 8

```
int i, sum;
main()
{
        sum = 0;
        for (i = 1; i <= 100; i++)
                sum += i;
        evnt_keybd();
}
```

The variable `sum` contains the sum of the hundred numbers. The variable `i` is the index that varies from one to a hundred.

The instruction

```
sum += i;
```

looks a bit funny. And in fact it is. This is equivalent to

```
sum = sum + i;
```

The only difference is that it is to quicker to type and execute.

The operators `-= /= *= %=` work in a similar way:

```
        sum -= i;                    sum = sum - i;
        sum /= i;                    sum = sum / i;
        sum*= i;                     sum =sum * i;
        sum %= i;                    sum = sum % i;
```

The statements on the left are equivalent to the statements on the right in the above table.

The operator `%` is the remainder operator.

The statement

```
i++;
```

is used to add 1 to `i` as we have seen in Exercise 6.

Exercise 9

```
int ch;
main()
{
        for (ch = 32; ch < 256; ch++)
        {
                putchar(ch);
                if (ch % 16 == 0)
                        printf("\n");
        }
        evnt_keybd();
}
```

This program displays the characters for the ASCII codes 32 to 255.

The statement:

```
putchar(ch);
```

writes the character corresponding to the ASCII value in `ch`.

The conditional

```
if (ch % 16 == 0)
        printf("\n");
```

is to tidy the display. The program writes 16 characters per line. After 16 have been written, i.e. when `i` is divisible by 16, then the following characters are written on the next line. To do this **HiSoft C** executes the statement:

```
        printf("\n");
```

which puts the cursor on the start of the following line.

Exercise 10

Here's the version of the program using a `switch` statement:

```
char c;
main()
{
        do
        {
                c = evnt_keybd();
                switch (c)
                {
                case 'a':
                case 'b':
                case 'A':
                case 'B':
                        putchar(c);
                        break;
                default :
                        putchar('*');
                }
        }
        while (c != ' ');
}
```

The variable `c` is of type `char`; it's used to store the character that is typed at the keyboard.

We've used a `do...while` loop to repeat the reading of a character and the echoing on the screen.

The C language lets us manipulate constants that represent the ASCII code for characters. In fact these constants are of integer type. The syntax for these character constants is that they are enclosed in single quotes. So you can write, for example,

```
c = 'a';
```

This instruction stores the ASCII code for `a` in the variable `c`.

These constants can be used anywhere an integer constant could be used, in particular in switch statements. The ASCII code for `A` is 65, so `case 'A'` is equivalent to `case 65`.

In our example, if the character typed on the keyboard is `a`, `b`, `A` or `B`, then it is echoed to the screen using the statement:

```
                    putchar(c);
```

Remember that `putchar` is a function to write a character on the screen. Its parameter is the ASCII code of the character to write. The default part lets us cope with all the other characters. If the key pressed is not `a` or `b` then a star is written on the screen.

The looping condition is `(c != ' ')`. This is to test if the space bar was pressed. `' '` represents the space character; it is a single space between two apostrophes.

The two characters `!=` mean "is different to". This is equivalent to `<>` in BASIC. Note that the test "is equal to" which is `=` in BASIC, is `==` in C. C distinguishes between the assignment operator which is written with a single equals sign and the equality operator which is two equals signs.

The following program is equivalent to the previous one but it uses an `if` statement instead of a `switch`.

```
char c;
main()
{
        do
        {
                c = evnt_keybd();
                if (c=='a' || c=='b' || c=='A' || c=='B')
                        putchar(c);
                else
                        putchar('*');
        }
        while (c != ' ');
}
```

You can see that the variable `c` is of type `char`. The program loops reading a character and then writing one to the screen.

In place of the `switch` statement we've got an `if` statement. The condition is a bit odd. The expression `c=='a'` tests if the variable `c` contains the ASCII code for the character `a`. We also test if `c` is equal to the codes for the characters `b`, `A` and `B`. Between these four tests there are a two vertical bars `||`. They mean "or" and are equivalent to OR in BASIC. The `if` condition in the program above is therefore:

if (`c` is equal to `'a'` or `c` is equal to `'b'` or `c` is equal `'A'` or `c` is equal to `'B'`).

In this case we echo the character to the screen, otherwise (`else`) we display a star.

Exercise 11

```
main()
{
        none();
}
none()
{
}
```

Another version:

```
main(){none();}none(){}
```

Exercise 12

```
char c;
int value;
main()
{
printf("Type two numbers separated by a space,\n");
printf("then press Return\n ");
printf("Sum = %d\n", read_number() + read_number());
evnt_keybd();
}

read_number()
{
value = 0;
c = getchar();
while (digit(c))
{
        value = value*10 + c-'0';
        c = getchar();
}
return (value);
}

digit(ch)
char ch;
{
if (ch >= '0' && ch <= '9')
        return (1);
else
        return (0);
}
```

The function `digit` has a parameter. It is a character. If it is a digit the function returns 1 (true). If not it returns 0 (false).

The two characters `&&` mean "and". This is equivalent to BASIC's AND operator. The condition:

```
        ch >= '0' && ch <= '9'
```

is true if `ch >= '0'` and `ch <= '9'` are both true.

Thus `digit(ch)` is true if it is a digit, and false otherwise.

The `read_number` function reads a positive number from the keyboard. The digits are read one by one as they are typed using the statement:

```
        c = getchar();
```

The variable `value` contains the value of this number. It is calculated as the individual digits that make up the number are read.

The expression `c - '0'` represents the value of the digit between 0 and 9 when c contains the ASCII code of a digit between `'0'` and `'9'`.

For example, suppose we have already typed the two digits `10` on the keyboard and then press `5` to make `105`. The `value` variable will contain 10 and `c` will be `'5'` for this digit. `c-'0'` is 5, so `value*10+c-'0'` is 105.

Finally the variable `value` is returned to the main program.

The main function has three `printf` function calls. The third one displays the value `read_number() + read_number()`. This is the sum of the two numbers that have been read from the keyboard.

Exercise 13

The program below is well commented. The comments are enclosed with /* and */ character sequences.

```
int win1, win2;
main()
{
/* open window 1 */
win1 = open_window(4095, 20, 20, 200, 150,
                    " window 1 ", " press a key ");
/* write to the window */
print_window(win1, "hi !");
print_window(win1, "This is window 1");
evnt_keybd();

/* open a second window */
win2 = open_window(4095, 200, 80, 200, 110,
                    " window 2 ", " press a key ");
/* position the text in window 2 line 4 column 2 */

pos_window(win2, 4, 2);
/* write "hi !" in window 2 line 4 column 2 */
print_window(win2, "hi !");
evnt_keybd();
/* clear inside window 1 */
clear_window(win1);
evnt_keybd();
/* clear inside window 2 */
clear_window(win2);
evnt_keybd();
/* write to window 2 */
print_window(win2, "This is window 2");
evnt_keybd();

/* close window 2 */
close_window(win2);
/* clear inside window 1 */
clear_window(win1);
/* write in window 1 */
print_window(win1, "This window 1 again");
evnt_keybd();
/* close window 1 */
close_window(win1);

}
```

Exercise 14

```
int window;
int x, y, w, h;
int i;
main()
{
x = Random() % 100;
y = Random() % 50 + 20;
w = Random() % (600-x) + 50;
h = Random() % (200-y) + 50;

window = open_window(4095, x, y, w, h,
                " moire ", " press a key");
size_window(window, &x, &y, &w, &h);
for (i = y+h; i > y; i -= 4)
        draw(x+w, i, x, y+h);
for (i = x+w; i > x; i -= 6)
        draw(i, y, x, y+h);
evnt_keybd();
close_window(window);
}
```

The integers `x,y,w` and `h` represent the size of the window. They are defined with the help of the four calls to the `Random` function.

Then the window is opened. All the gadgets are drawn because the the first parameter to `open_window` is 4095.

The `size_window` function stores the size of the usable area of the window in `x,y,w,h`. This is not the same as the original size of the whole window because of the presence of the sliders, title etc.

The `for` loops draw the lines as rays from the bottom left corner at co-ordinates (`x, y+h`)

The program then waits for a key to be pressed and closes the window.

Appendix B HiSoft C Language Reference

HiSoft C for the ST corresponds to the language described in Kernighan & Ritchie in their book "The C language" (first edition).

B.1 Lexicographic elements

A lexicographic element is the smallest unit in the language. They are the bricks that make up a program.

B.1.1 The keywords

The following words are the keywords of the C language. They must be typed in lower case.

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

These words can only be used as keywords.

The names of library functions are also considered keywords. They must only be used for calling these functions. You may *not* re-define a library function identifier.

When you type the name of a library function, only the first eight characters are required. **HiSoft C** can automatically add the remaining characters. For example, if you type `vst_unlo` **HiSoft C** automatically displays `vst_unload_fonts`.

B.1.2 Identifiers

A C identifier represents a variable.

It is made up of a set of alphanumeric characters. Only the first eight characters are taken into account.

The legal characters in an identifier are letters (upper or lower case), digits and the underline "_". Identifiers may not start with a digit.

Upper and lower case letters are considered different. Thus, `ident` is not the same identifier as `IDENT`.

B.1.3 Integer constants

Integer constants may be expressed in decimal, hexadecimal or octal.

- A decimal constant is a sequence of digits which does *not* start with zero.
- An octal constant starts with a zero and is followed by a set of digits between `0` and `7`. An error message is given if `8` or `9` is used in an octal constant.
- A hexadecimal constant starts with a `0`, followed by an `x` or `X` and a sequence of hexadecimal digits. The hexadecimal digits are `0` to `9`, `a` to `f` (or `A` to `F`) with the letters indicating the values 10 to 15 respectively.

A constant may be followed by a letter `l` or `L` which indicates that the constant is of type long.

Note that a sign does not make up part of a constant but is considered a unary operator.

B.1.4 Floating point constants

A floating point (or real) constant is a collection of the following elements:

a whole part, decimal point, fractional part, exponent.

The decimal point and fractional part or the exponent may be omitted, but not both.

All floating point numbers are considered double precision.

B.1.5 Character constants

A character constant consists of a character enclosed in two apostrophes: `'A'`.

There are ways of writing certain special characters:

`'\n'`	new line
`'\b'`	backspace
`'\t'`	tab character
`'\r'`	carriage return (no linefeed)
`'\''`	apostrophe
`'\0'`	null character

You can also create character constants by specifying their ASCII code in base 8,10 or 16: `'\23'` (character with ASCII code 23 decimal), `'\012'` (character with ASCII code 12 octal) or `'\x45'` (character with ASCII code 45 hex).

The value of a character constant is an unsigned integer equal to the ASCII code for the character.

B.1.6 String constants

A string constant consists of a set of characters enclosed in double quotes and may include all the possible special characters (`\n`, `0xC` etc, as described above.

A double quote may be included in a string constant by using `\"`.

Strings may be of any length, even 0 characters.

HiSoft C inserts a null character (code 0) at the end of every string to mark its end.

The value of a string is a pointer to the first byte of that string. Thus, when the interpreter encounters the string `"qwerty"` it returns the address where the string is stored in memory not the string itself.

B.1.7 Operators

An operator consists of one, two or three characters.

Single character operators:

```
+   -   *   /   %   ^   &   ~   |   =   <   >   ?   :   !   (   )   [   ]   ,
```

Multi character operators

```
++   --   <<   >>   ->   <=   >=   ==   !=   &&   ||
-=   +=   *=   /=   %=   &=   ^=   |=   <<=  >>=
```

These operators consisting of several characters may *not* have their components separated by white space.

B.1.8 Comments

Comments start with `/*` and finish with `*/`.

Comments may not be nested.

They must be opened and closed on a single line. The **HiSoft C** editor will not let you enter comments that are not closed.

When loading a file containing comments that are several lines long, **HiSoft C** automatically inserts the `/*` and `*/` symbols at the start and end of each line.

B.1.9 Separators

The separators split up the lexicographic elements. These are spaces, tab characters and end of line characters.

The `;` (semi-colon) character terminates statements.

The curly brackets `{` and `}` characters are used at the start and end of blocks of statements.

B.2 The Pre-processor

B.2.1 Define

The `#define` directive has several differences to that given in K & R because **HiSoft C** is interpreted rather than compiled.

The syntax is as follows:

```
#define name constant

#define name(arg1, arg2,...) expression
```

(without a semi-colon at the end).

The value given in `#define` must be used in expressions. A macro may not be used outside of expressions.

Thus it is impossible to re-define a type in this way:

```
#define int WORD
```

You must use `typedef` instead.

It is also illegal to define a variable with the same name as a macro.

Nested defines are also illegal. That is a `define` may not reference a value that is given by another `define`.

You can also write:

```
#define PI 3.141592
#define hello "hello"
```

to define both floating point and string constants.

Examples :

```
#define SIZE 512

#define UPPER(a) ((a) >= 'A' && (a) <= 'Z')
```

B.2.2 Include

The `#include` directive loads a header file. The syntax is

```
#include "file_name"

#include <file_name>
```

The version with double quotes looks for the file in the current directory. The version with greater than and less than signs looks in the `\header` directory on the current drive.

Include files may themselves include other files.

B.2.3 Conditional Interpretation

It is possible to either interpret, or not interpret depending on a condition. This is performed with the help of the `#if`, `#ifdef` and `#ifndef` directives.

```
#if expression
```

is true if the expression is non-zero (and if the identifiers in the expression are properly defined).

```
#ifdef identifier
```

is true if the identifier is defined.

```
#ifndef identifier
```

is true if the identifier is not defined.

If one of these three directives is used there must be a following `#endif`.

So, if the condition is true the lines between the `#if` and the `#endif` are interpreted. If the condition is false these lines are ignored.

The `#else` directive may be used. This causes lines between an `if` directive and the `#else` to be interpreted if the condition is true and the lines between the `#else` and the `#endif` to be interpreted if the condition is false.

Example:

```
#if LABEL
/* lines interpreted if LABEL is non-zero */
...
#else
/* lines executed if LABEL is zero */
...
#endif
```

The identifier `IC` is always defined as 1 so that code may be added that will only be used under the **HiSoft C** interpreter.

B.3 Operators

The C language operators are summarised below together with their priority:

15	`()`	function call
15	`[]`	array index
15	`.`	direct selection of a structure element
15	`->`	indirect selection of a structure element
14	`*`	indirection
14	`&`	address calculation
14	`-`	unary minus
14	`!`	unary logical negation
14	`~`	bitwise negation
14	`++`	increment
14	`--`	decrement
13	`*`	multiplication
13	`/`	division
13	`%`	modulus
12	`+`	addition
12	`-`	subtraction
11	`<<`	shift left
11	`>>`	shift right
10	`<`	less than comparison
10	`>`	greater than comparison
10	`<=`	less than or equals comparison
10	`>=`	greater than or equal comparison
9	`==`	equality comparison
9	`!=`	inequality
8	`&`	bitwise AND
7	`^`	bitwise exclusive OR
6	`|`	bitwise OR
5	`&&`	logical AND

4	`||`	logical OR
3	`?:`	conditional operator
2	`=`	simple assignment
2	`+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `!=`	operation & assignment
1	`,`	sequential evaluation

Operator priorities may be changed by using parentheses.

B.4 Variable types

HiSoft C recognises all the types defined in K&R.

B.4.1 Simple types

`void`	the type of a function that doesn't return anything. This type may only be used with functions and may not be used with structured types
`char`	8 bit *un*signed integer
`short`	16 bit signed integer
`unsigned short`	16 bit unsigned integer
`int`	32 bit singed integer
`unsigned int`	32 bit unsigned integer
`long`	32 bit signed integer
`unsigned long`	32 bit unsigned integer
`float`	64 bit double precision floating point in IEEE format
`double`	64 bit double precision floating point in IEEE format

B.4.2 Structured types

As well as the simple types described above, types may be constructed from other types:

- Arrays of any type
- Pointers to any type
- Functions returning a particular type
- Structures consisting of simple types, arrays, pointers, unions or other structures.
- Unions containing simple types, arrays, pointers, structures or other unions.

The rules for building structured types can be used recursively, so that an array may contain arrays and a pointer may point to an array of structures which contains pointers etc.

Be careful; in **HiSoft C** the maximum number of array dimensions is 16 and for the same reason the maximum depth of pointer indirections is 16.

B.5 Declarations

A variable declaration is constructed as follows:

```
memory_class type decl_item1, decl_item2,...;
```

B.5.1 Memory classes

Memory classes indicate where the declared variable is to be physically stored. It may be stored in ordinary memory, on the stack, in a register or even in another module.

The class also indicates the scope of the variable, that is to say, the area of the program where the variable may be accessed.

The memory class may be omitted. In this case if the variable is declared outside a function the variable is considered global to the module. If the variable is declared within a function it is considered local to that function. No more than 40 local variables, including the parameters may be declared within a function. This would be very poor style, you should split the function into several smaller ones.

The available memory classes are as follows:

`extern`	indicates that a variable is declared in another module.
`static`	indicates that the scope of the variable is strictly limited to the module/function in which it is declared.
`register`	used inside a function definition indicates that the variable is to be stored in a register rather than on the stack.
`typedef`	indicates that the "variable" declared is to be considered a new type of variable. This new type is equivalent to the type specified in the `typedef`.

B.5.2 The type

The type part of a declaration consists of one of the types described above or a name defined using `typedef`.

The `typedef` is optional when declaring functions. In this case the function is taken as of type `int`.

B.5.3 Declared items

The following element in a declaration is a list of declared items, separated by commas if there are several.

A declared item consists of an identifier (the name of the variable) together with characters indicating if it is a pointer, a function, a structure or an array, separated by commas if there are several.

A declared item may contain an initialiser preceded by an = sign.

B.5.4 Initialisers

For variables which are declared outside functions (global variables), simple, pointer and array types may be initialised. However `structure` types may not be initialised.

For variables declared inside functions, only simple and pointer types may be initialised. Thus it is impossible to initialise arrays or structures within functions.

B.6 Statements

HiSoft C supports all the statements described in K&R.

- `if else`
- `while`
- `do while`
- `for`
- `switch case default`
- `break`
- `continue`
- `return`
- `goto`

A `break` instruction in a `switch` statement may only be used at the end of a list of statements. Thus you may not use:

```
case '1':
        if (a)
        {
                a--;
                break;
        }
        a++;
        break;
default:
        /* etc... */
```

The program section above should be replaced by

```
case '1':
        if (a)
                a--;
        else
                a++;
        break;
default:
        /* etc... */
```

The `goto` statement must reference a label in the same C block as the `goto` statement itself. It is impossible to `goto` another block.

A C block is a sequence of statements enclosed in curly brackets.

The following two programs are not accepted by **HiSoft C**:

```
{
        {
                goto here;
        }

here:
        {
        }
```

and

```
        here:

        {
                goto here;

        }
```

On the other hand, the following is allowed:

```
{
    here:
    {
    }
    if (cond)
        goto here;
}
```

B.7 Operations on types

B.7.1 Pointer conversions

A pointer to a type may not be assigned to another pointer if it is not of equivalent type (the same level of indirection) without an explicit type conversion.

So, storing an integer in a pointer variable gives an error message unless you use a cast.

```
{
char *p;
int i;
p = i;          /* illegal */
p = (char*)i; /* allowed */
}
```

Be careful when assigning a pointer to an integer. Except with pointers to characters, pointers must have even addresses; using odd addresses will give an error message.

B.7.2 Arrays and pointers

Arrays and pointers are strictly equivalent. In practice, the statement `tab[index]` is converted to `*(tab+index)`.

When adding an integer to a pointer the value of the integer is first multiplied by the size of the object that the pointer points to.

So, if p is a pointer to `int`, `p+1` adds 4 to `p` (4 * the size of an `int`).

B.7.3 Functions

As far as **HiSoft C** is concerned there are two classes of functions:

- **HiSoft C** library functions and loaded compiled (or assembled) functions. These are stored in machine language and are executed directly by the 68000.

- User functions written with the Interpreter's editor. These functions are stored in source form and are not executed directly by the 68000 but are interpreted by **HiSoft C**.

These two classes of functions are distinct; the 68000 cannot execute an interpreted C routine and **HiSoft C** cannot interpret a library function. This differs from a compiled C program where all functions are in 68000 code.

With an interpreted C function there are two ways of calling it. You can call it directly or find its address and store this as a pointer to a function. This provides a method of passing functions as parameters to other functions.

With a compiled C function (library function or a compiled one that has be loaded) you can only call it; it is not possible to find its address, this will give an error message.

Thus **HiSoft C** pointers to functions can only point to interpreted functions not compiled ones.

B.7.4 Structures and unions

The possible operations on pointers are accessing one of its members, (via `"."` or `"->"`) or calculating its address. All other operations, such as assignment or passing as a parameter, are illegal.

When you access a structure member (using `"."` or `"->"`), the identifier on the left hand side must be a structure (with `"."`) or a pointer to a structure (with `"->"`), and the identifier on the right must be a member of a structure or union.

Appendix C Error Messages

Here is a detailed commented listing of **HiSoft C**'s error messages.

C.1 Interpreter Error Messages

When an error is found whilst running a program, a window opens with a description of the error that has occurred. Finally the cursor is positioned at the precise place that the error was found. However sometimes the actual cause of the error may be elsewhere (e.g. omitting a declaration).

0 Program execution stopped

The program has been interrupted. The user has pressed two of the `Control`, `Shift` or `Alt` keys. Alternatively a `stop`, `exit` or `abort` statement has been encountered.

1 Syntax error

HiSoft C does distinguish between some little syntax mistakes. This error doesn't appear as frequently as with some BASICs!

2 Missing ;

A semi-colon was expected at the end of a declaration or statement.

3 Missing parenthesis

A parenthesis has been opened but not closed or vice versa.

4 Out of memory

You've run out of memory. Well done! Seriously though, try removing Desk accessories, ramdisks or other memory resident programs.

5 Missing integer

HiSoft C was expecting to find an integer, perhaps, as an array index. Empty array indices are only allowed in function parameter declarations.

6 Too many dimensions

The maximum number of dimensions in an array and the maximum number of indirections off a pointer is 16.

7 Duplicate declaration

A global variable has been declared twice. This is strictly illegal. Neither may you have two structure field identifiers with the same name.

8 Undefined structure name

A reference has been made to a structure that has not been defined. Please define it.

9 Missing structure name

HiSoft C has expected to find the name of a structure but it has found something else.

10 Empty structure

Either there are no elements declared in the structure or the structure name is used with indirection.

11 Missing identifier

HiSoft C expected to find a variable identifier. This error can occur if you omit the parameters in a function definition.

12 Duplicate declaration of local variable

A local variable is declared twice within the same function.

13 Bad type for a function argument

Function arguments may not be of type function, union or structure.

If you need to pass a union or function as a parameter, pass pointers to them.

14 Name mismatch in the parameter list

The function argument list and the declarations of those arguments disagree.

15 Too many local variables

The maximum number of local variables in a function (together with parameters) is 40. This error is probably due to very bad programming style; split your function up.

17 Parameter not declared

A parameter is present in the argument list but there is no declaration for it.

18 Missing }

A block has been opened using `{` but not closed with `}`. Use the `Home` key (see the Cursor key item on the Help menu).

19 Undefined label

There is a reference to a variable that has not been declared. Perhaps it has been mistyped.

20 Illegal function declaration

There is an error in the declaration section of a function, or the entire body (the statements) has been omitted.

21 Variable type error

The type specified is illegal. For example there is no such thing as an `unsigned struct`.

22 No reference to local label

A local variable has been declared but is not used.

23 Missing main function

The function `main` has been left out. Every program must have a function `main` so that execution can start.

24 Keyword found in an expression

A language keyword has been found in an expression.

25 "lvalue" required

An "lvalue" is a value that you can assign to, i.e. can appear on the left of an = sign. You can have `a=1;` because `a` is an lvalue. But `1=5` is illegal because 1 is not an lvalue.

So this error occurs when the left hand side of an assignment statement is illegal.

It will also happen when using `&` the "address of" operator. The argument to this must be an lvalue. Note that the name of an array is *not* an lvalue.

26 Pointer assigned to an integer

HiSoft C forbids assigning pointers to integers without using a cast.

27 Integer assigned to a pointer

HiSoft C forbids assigning integers to pointers without using a cast.

28 Pointer type mismatch, operator '='

You cannot assign a pointer value to another one, unless they point to the same type, without using a cast.

29 Illegal operator for pointer arguments

For example, pointers can not be multiplied together.

30 Division by zero

An attempt to divide by 0 has been made. Often this will be caused by a variable that has been assigned an inappropriate value.

31 Integer used with operator "*"

The indirection operator * must be followed by a pointer or array.

32 Pointer used instead of an integer

A pointer, rather than an integer has been used to index an array.

33 Incorrect pointer value

The interpreter has spotted the use of an un-initialised pointer, and rather than crashing the machine gives you this error message.

34 DO without WHILE

A `DO` statement has been found, but there is no corresponding `WHILE`. Perhaps some curly brackets have been added or left out.

35 ELSE without IF

An `ELSE` statement has been found, but there is no corresponding `WHILE`. Perhaps some curly brackets have been added or left out.

36 CASE without SWITCH

A `CASE` statement has been found, but there is no corresponding `SWITCH`. Perhaps some curly brackets have been added or left out.

37 DEFAULT without SWITCH

A `DEFAULT` statement has been found, but there is no corresponding `SWITCH`. Perhaps some curly brackets have been added or left out.

38 Missing : in a CASE instruction

The correct syntax of a `CASE` statement is:

```
case [expression] :
```

However the colon has not been found.

39 Missing CASE in a SWITCH instruction

Immediately after a SWITCH statement there must be either a CASE (or DEFAULT) statement, but there is something else instead.

40 Missing { in a SWITCH instruction

Immediately after a SWITCH statement there should be a curly bracket. For example:

```
switch (a)
{
case 1: /* etc */
```

41 Missing parameters in function call

A **HiSoft C** library call has been made and there aren't enough parameters. Alternatively you may have got the function name wrong. The Help command can be used to check the number of parameters.

42 Missing , in a function call

Each parameter should be separated by a comma. A character that is neither a `,` nor a `)` has been found after a parameter.

43 Unknown operator

Error in an operator. Or perhaps you have used a unary operator as a binary one or vice versa. e.g. `++`,`--`,`!` and `~` can only be unary operators. Thus `i++` is legal but `i++j` is not.

44 Too many parameters in a function call

A **HiSoft C** library call has been made and there are too many parameters. Alternatively you may have got the function name wrong. The Help command can be used to check the number of parameters.

45 Floating point value used with SWITCH

Only integer or pointer values can be used in SWITCH statements, not floating point ones.

46 Bad type for a function name

An attempt to pass an entire structure to a function has been made. Only pointers to structures may be passed.

47 Bad function pointer usage

An attempt has been made to use a pointer to a function as a function. For example,

```
char (*fct[5])()  /* declare an array of pointers */
fct();            /* call function */
```

To access an element in the `fct` array the following should be used:

```
fct[3]()              /* call function */
```

48 Stack full

There have been too many recursive function calls and the stack has overflowed. This is almost certainly because of a programming error. For example,

```
function()
{
        i = 5;
        if (i > 2)
        function();
}
```

49 Bad usage for BREAK or CONTINUE

`BREAK` and `CONTINUE` statements may be used inside `WHILE`, `DO` or `FOR` loops to change the order of execution. `BREAK` statements may also be used in `SWITCH` statements. All other uses of these statements are illegal.

50 Bad value for shift number

The second argument of << and >> (the shift operators) must not be negative.

51 Integer value required with this operator

The logical operators (^, ~, |, &) and modulo (%) can only be used with integers, not floating point numbers.

52 Floating point parameter expected

A library function needing a floating point number has been passed an integer instead.

53 Structure identifier expected

Only structure variables may be used before the "." and "->" operators.

54 Structure item expected

A field structure name must be used after the "." and "->" operators.

55 You can't use this operator on structure type

Using "." on a pointer to a structure or using "->" on a structure variable.

56 Forbidden usage made on structure type

This operator may not be used on a structure variable. `&` to find the address may be used and members may be accessed using ".". Apart from this all others are forbidden.

57 Forbidden character

An illegal character not recognised by the language, has been found outside a string or comment, for example a control character or character greater than 127.

58 Missing '(' after a library function name

This message is also given if you try to calculate the address of a library function.

59 Missing variable type

The name of the type of a variable is expected after `sizeof`.

60 Incorrect variable type

An array type can not be used with `sizeof` or casts.

61 Preprocessor keyword expected

A preprocessor keyword is expected after a `#` sign.

62 Missing ':' after '?'

The only valid character after `?` is the `:` of the `?:` conditional operator.

63 Bad function return value

The value in a `return` statement is not the same as in its declaration.

64 Too many items in an initialization

Too many items have been used in an initialiser (normally an array) to match the declaration.

65 Macro name expected after #define

A non-empty macro name is required after a `#define`.

66 Empty macro definition

The entire body of a macro may not be empty.

67 Forbidden operator in an initialization

Only constants may be used in initialisers. For example, you can't call a function inside a macro.

68 You cannot initialize aggregates

`struct` type variables may not be initialised.

69 File name expected after #include

An invalid file name has been used after `#include`.

70 Too many include files

No more than 8 include files may be loaded at the same time.

71 You must check the "link at runtime" option to use #include

To use the `#include` facility you must first select the Link at runtime option on the Run menu.

72 Cannot load include file

The include file can not be loaded. It needs to be in the `\HEADER` directory on the drive that **HiSoft C** was loaded from if < and > have been used and in the current directory if quotes are used.

73 Bad type usage

Error in a type when using `sizeof` or a cast.

74 Bus error

68000 exception. An attempt to access an illegal address has been made, such as one of the ST's low memory variables.

75 Odd Address access

68000 exception. An attempt to indirect using an odd address has been made.

76 Unknown instruction

68000 exception. An illegal 68000 instruction has been encountered.

77 Division by 0

68000 exception. An attempt to divide by 0 has been made.

78 CHK instruction

68000 exception. An exception has been generated by a CHK instruction.

79 TRAPV instruction

68000 exception. An exception has been caused by a TRAPV instruction.

80 Supervisor mode required

68000 exception. An instruction that can only be used in supervisor mode has been encountered.

81 Trace mode

68000 exception. An exception has been generated because we are in trace mode.

82. Cannot initialize a local complex type variable

Arrays and structures that are local to a function may not be initialised. See **Appendix B**.

83 Goto out of local block

`goto`'s may not reference a label that is not in the same C block (`{...}`) as the `goto`. See **Appendix B**.

84 Label name expected after goto

An item which is not an identifier has been found after a `goto` or the identifier used is declared as a variable.

85 #endif or #else without #if

A `#endif` or `#else` directive has been found without an associated `#if`, `#ifdef` or `#ifndef`.

86 #endif expected after #if

A `#if` directive has been used, but the end of the text has been found without encountering a `#endif`.

87 This label is already used in the function library

An attempt to re-define a variable which is already used in a library function has been made.

C.2 Editor Error Messages

101 Cursor is inside the block. Can't copy the block

You may not copy a block to a position inside that block.

102 The line is too long. Maximum length is 127 chars

The length of lines is limited to 127 characters.

103 This line number does not exist

You have entered a non-existent line number (there aren't that many lines in the file).

104 No more windows!

All the GEM windows have been used up. Close desk accessories or return to the Desktop.

105 You ran out of memory...

You've used all the available space.

106 The file you want to load does not exist

If this error occurs after the use of `#INCLUDE` make sure that the disk containing the `HEADER` folder is present in the drive from which **HiSoft C** was loaded.

107 Search failed: String not found

108 Disk error or disk full

Either the disk is full or the disk can't be read or written to, perhaps because there is no disk in the drive.

109 Comment is not closed on current line

A comment must only be one line long. Add a `*/` at the end of the line.

110 Error in char or int constant on current line

HiSoft C has found the start of a constant (octal beginning with `0`, hex beginning with `0x` or a character beginning with `'`) but it is incorrectly formed.

111 Can't find end of string on current line

A string of characters must be closed on the same line as it is opened. Perhaps there is a " missing.

112 Syntax error in macro command

The contents of a macro command, given by `Shift` and a function key is wrong.

113 I can't give you help about this keyword

There is no help file corresponding to this keyword.

114 Error in project file line XXX

The contents of the project file that you are trying to load has an error on the given line.

C.3 Loading Error Messages

The following errors may be given when **HiSoft C** is loading:

I can't load the resource file

Make sure that you have all the `F1.C` to `F9.C` files present on the current disk.

I can't run in low resolution

Switch to Medium resolution using the Desktop.

Too much memory left to the system

So there's not enough for **HiSoft C** to run. Use the System memory size command on the Run menu.

Appendix D Porting programs

D.1 Porting from the interpreter to a compiler

Compiling a program written in **HiSoft C** doesn't pose too much of a problem if you have a compiler.

HiSoft C is almost totally compatible with Lattice C interpreted programs can be compiled almost immediately with Lattice C 5: see the Lattice C 5 manual for details.

With other compilers, you may have problems because of the size of integers. Aztec C, for example, has a flag to enable 4 byte integers and you should use this.

Moving to a new implementation of C can be tricky, you need to know the compiler, include files and libraries well. It is not something recommended for beginners. One problem to watch out for is that **HiSoft C** expects integers to be 32 bit: so you should use the appropriate compiler flags to ensure that this is the default.

The general procedure to use is as follows:

• Convert the program to ASCII format. **HiSoft C** interpreted programs are tokenised in a special way. To convert a program to ASCII load **HiSoft C** and choose Editor mode from the File menu (see **Section 1.8**). Then load the file and it will automatically be converted to ASCII and you can save it as ASCII.

• For some **HiSoft C** specific functions (the GEM toolbox), the source is supplied in the `SOURCE` directory on Disk 2 for you to compile.

If you have used the toolbox functions you will need to compile these files (`compil.c`, `libwind.c`, `libdial.c`, `libmenu.c` and `libresou.c`) and link with the object files produced. There's an example of a linker file in `c.lnk`.

`define.h` contains some constants that are built into **HiSoft C**.

If you are using Lattice C 5 you need not re-compile the Toolbox: you can use the library that is supplied with Lattice.

Whichever compiler you are using, you must also rename your `main` function to be `ic_main`. This is because the toolbox has to perform some initialisation.

D.2 Porting from compilers to HiSoft C

Porting programs in the other direction, written for a compiler to the Interpreter can be more difficult.

First of all because programs run more slowly under the interpreter than when compiled.

Also some compilers have extensions to the K&R standard (for example some of the features of the new ANSI C),

The other possible main sources of incompatibilities are the fact that **HiSoft C** variables may not have the same name as a library function and the restrictions on the use of `#define`. See **Appendix B** for details.Remember that HiSoft C uses long (32-bit) integers as standard, so be especially careful if porting code that assumes that 16 bit integers are being used. See **Appendix B** for details.

Appendix E Bibliography

This bibliography contains our suggestions for further reading on the subject of C, the ST, and GEM. The views expressed are our own and as with all reference books there is no substitute for looking at the books in a good bookshop before making a decision.

Please also note that none of these books on C describe **HiSoft C** for the ST in particular. Should an example program fail to work as expected please study the appropriate sections in this manual.

Books about C

The C programming Language by Kernighan & Ritchie, published by Prentice-Hall

The C programmer's bible, which is very expensive, unfortunately. There are two editions; the first one is more applicable to HiSoft C (and most compilers). The second edition covers the new ANSI standard which isn't implemented by many compilers (at least not until recently). If you have a lot of experience in, say Pascal, this is also a good tutorial book.

Learning to program in C by Thomas Plum, published byPrentice-Hall

A tutorial book which starts from first principles.

The Big Red Book of C by Sullivan, published by Sigma Press

A cheap tutorial book.

The C programming Tutor by Wortman & Sidebottom, published by Prentice Hall

A cheaper tutorial from the publishers of K&R.

C at a glance by Adam Denning, published by Chapman & Hall

Another low cost tutorial book that mentions the 'other' **HiSoft C** for Z80 computers.

ST Technical Manuals

GEM Programmer's Guide Volumes 1 & 2 - VDI and AES by Digital Research

The definitive guide to the VDI and AES, but marred by mistakes. Only available to registered developers.

GEMDOS Specification by Digital Research

The definition of the GEMDOS calls. Only available to registered developers.

A Hitchhikers Guide to the BIOS by Atari Corp

The definition of the BIOS and XBIOS calls, and corrections to the GEMDOS manual. This is accurate, a good read and updated regularly. Normally only available to developers.

The Anatomy of the Atari ST by Data Becker/Abacus

This book is the best documentation available for the user who is not a registered developer. It describes the hardware and non-GEM aspects of the operating system, including an (out-of-date) BIOS listing. Thoroughly recommended, despite its inaccuracies.

GEM on the Atari ST by Data Becker/Abacus

This describes programming under GEM, though is not as complete as the DR manual, but has similar errors. It describes calls mainly from C, although there is more reference to the 68000 than in the DR manual. Better than no book at all on GEM.

Concise Atari 68000 Programmer's Reference by Katherine Peel, published by Glentop

An alternative to The Anatomy of the Atari ST. It contains information on the ST's hardware, the operating system and GEM. Its coverage of the various levels of the machine is comprehensive, though a couple of sections are very inaccurate and some features are described that simply don't exist. It is rather difficult to find one's way around as the layout is based on large numbers of tables and it lacks an index.

Tricks and Tips on the Atari ST by Data Becker/Abacus

This contains a wide variety of material, including an accurate description of the more esoteric ST BASIC commands, and good sample listings including a RAM-disk driver and desk accessory.

Appendix F Technical Support

So that we can maintain the quality of our technical support service we are detailing how to take best advantage of it. These guidelines will make it easier for us to help you, fix bugs as they get reported and save other users from having the same problem. Technical support is available in four ways:

Phone	our technical support hour is normally between 3pm and 4pm, though non-European customers' calls will be accepted at other times.
Post	if sending a disk, *please* put your name & address on it.
BIX™	our username is (not surprisingly) *hisoft*. Would UK customers please use CIX or more old fashioned methods; it's cheaper for everyone.
CIX™	our username is (still not surprisingly) *hisoft*.

For bug reports, please always quote the program, computer and the version number of the program (the one displayed by the About box) and the serial number found on your master disk.

If you think you have found a bug, try and create a small program that reproduces the problem. It is always easier for us to answer your questions if you send us a letter and, if the problem is with a particular source file, enclose a copy on disk (which we will return).

Upgrades

As with all our products, **HiSoft C** is undergoing continual development and, periodically, new versions become available. We make a small charge for upgrades, though if extensive additional documentation is supplied the charge may be higher. All users who return their registration cards will be notified of major upgrades.

Suggestions

We welcome any comments or suggestions about our programs and, to ensure we remember them, they should be made in writing.

Index

HiSoft
High Quality Software

HiSoft C for the Atari ST

ISBN 0 948517 25 5

£49.95 inc. Software

Program by
loriciels