HiSoft DevpacST

Assembler/Editor/Debugger

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

Copyright © HiSoft 1988

DevpacST Version 2 April 1988

Printing History:

1st Edition	August 1986 (ISBN 0 948517 04 2)
Reprinted	April 1987 & October 1987
2nd Edition	April 1988 (ISBN 0 948517 11 5)

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

ISBN 0 948517 11 5

Table of Contents

CHAPTER 1
Introduction

Always make a back-up

Before using DevpacST you should make a back-up copy of the distribution disk and put the original away in a safe place. It is not copy-protected to allow easy back-up and to avoid inconvenicnce. This disk may be backed-up using the Desktop or any back-up utility. The disk is single-sided but may be used in double-sided drives.

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

Serial No:

Registration Card

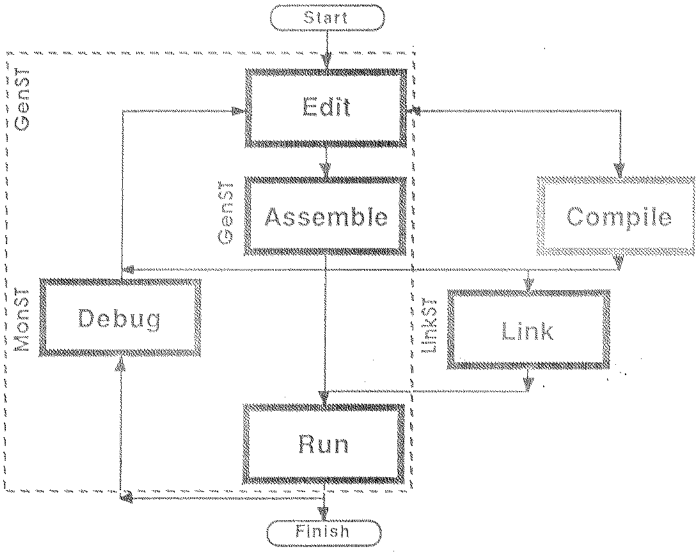
Enclosed with this manual is a registration card which you should fill in and return to us after reading the licence statement. Without it you will not be entitled to technical support or upgrades. Be sure to fill in all the details especially the serial number and version number. Also supplied is a 68000 Pocket Guide which details the entire 68000 instruction set.

The README File

As with all HiSoft products DevpacST is continually being improved and the latest details that cannot be included in this manual may be found in the README.S 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 direct it to a printer by clicking on the Print button.

The Development Cycle

The purpose of DevpacST is to allow you to enter assembly language programs, assemble them to machine-code and debug them if (or should that be 'when') they don't work. Depending on your application, you may also be using a linker to join together separate modules, possibly with the output from a high level language compiler. Of course the faster the development cycle, the faster you can get your programs up and running and DevpacST was designed to be as fast and powerful as possible. The usual development cycle is best illustrated by a diagram.



Of course the faster the cycle, the faster you can get your programs up and running and DevpacST was designed to be as fast and powerful as possible. The Link stage is optional, as is the Compile stage.

DevpacST Disk Contents

The supplied single-sided 3.5" disk contains these files:

Programs

GENST2.PRG	GEM screen editor and assembler
MONST2.PRG	the GEM program debugger
MONST2.TOS	the TOS program debugger
GENST2.TTP	stand-alone version of assembler
AMONST2.PRG	auto-resident debugger
CHECKST.PRG	diagnostic program
LINKST.TTP	GST-format linker
NOTRACE.PRG	trace exception dis-abler
MENU2ASM.TTP	menu compiler

Text Files

README.S	latest details about DevpacST
DEMO.S	very simple TOS program used in tutorial
GEMTEST.S	simple GEM demo program
DESKACC.S	example desk accessory
GEMMACRO.S	macros for AES/VDI interface
AESLIB.S	AES library source
VDILIB.S	VDI library source
NOTRACE.S	source to NOTRACE.PRG
MENUTEST.S	example GEM program using menu
MENUTEST.MDF	sample menu definition file
MAKEGEM.S	creates GEMLIB
GEMLIB.LNK	control file for GEMLIB

Binary Files

GEMLIB.BIN	AES & VDI library

Folders

OLDGEM	updated GEM examples from GenST 1

How to Use this Manual

This manual makes no attempt to teach 68000 assembly language programming or to detail the instruction set. For the former, the bibliography lists suitable books, while for the latter the supplied Pocket Guide is very useful. The Appendices give an overview of the technical aspects of the Atari ST but they are not intended as a complete technical description of the machine.

This manual is set out in five chapters, this introduction, a chapter on the screen editor, a chapter on the macro assembler, a chapter on the debugger, then a chapter on the linker. In addition there are eight Appendices which detail various additional information. We suggest you use the manual in a way that depends on what type of user you are:

DevpacST Version 1 Users

Turn to **Appendix H** and read the section describing the new features, then read the **Reference** section of **Chapter 4** if you intend using MonST, as it has changed considerably. The other section you may need to read is that on **File Formats** in **Chapter 3** if you are interested in generating linkable code.

Beginners

If you are a newcomer to assembly language then we recommend that you read one of the books in the **Bibliography** alongside this manual.

At the end of this chapter there is a simple tutorial which you should follow to familiarise yourself with the use of the main parts of the program suite.

Chapter 2 details the editor and is well worth reading, though much of **Chapter 3**, detailing the assembler, is liable to mean nothing until you become more experienced. The **Overview** section of **Chapter 4**, the debugger, is strongly recommended, though **Chapter 5** and the **Appendices** can be left for a while. Looking at the supplied source code may be helpful, but the GEM programs may be hard going as they were not written with the beginner in mind.

Experienced Users

If you are experienced in the use of 68000 assembly language but have not used DevpacST before then here is a very quick way of assembling a source file:

Load `GENST2.PRG`. Press `Alt-L` and select your file which will load into the editor. Press `Alt-A` and select the options which you require - if generating executable code then click on the Memory button for additional speed. Pressing `Return` will start the assembler, which may be paused by pressing `Ctrl-S`, `Ctrl-Q` resumes. Any assembly errors will be remember and on return to the editor you will be placed on the first one. Subsequent errors may be found by pressing `Alt-J`.

To run your successfully-assembled program (if assembled to memory) press `Alt-X`. If assembled to disk press `Alt-O` then select the program.

As a quick introduction to the debugger the following tutorial is recommended. If you have any problems *please* read the relevant section of the manual before contacting us for technical support.

A Very Quick Tutorial

This is a quick tutorial intended to let you see how quick and easy it is to edit, assemble and debug programs with DevpacST.

In this tutorial we are going to assemble and run a simple program, which contains two errors, and debug it. The program itself is intended to print a message and wait for a key to be pressed before quitting.

To start with load `GENST2.PRG` from your backup copy (you have made a backup, haven't you?) which must also contain the files `MONST2.PRG` and `DEMO.S`, at minimum, by double-clicking on its icon. After a short delay the screen will show an empty window; to load the file you should move the mouse over the File menu and click on Load. The standard GEM file selector will then appear and the file we want is called `DEMO.S`. You may either double-click on the name or type it in and press `Return` to load the file.

When the file has loaded the window will show the top lines of the file. If you want to have a quick look at the program you may click on the scroll bar or use the cursor keys.

With most shorter programs it is best to have a trial assembly that doesn't produce a listing or binary file to check the syntax of the source and show up typing errors and so on. Move the mouse to the Program menu and click on Assemble.

A dialog box will appear, which should be left alone except the button near the bottom, labelled None, should be clicked on. Click on the Assemble button or press `Return` and the assembly will begin.

The assembler will report an error, `instruction not recognised`, and pressing any key will return you to the editor. The cursor will be placed on the incorrect line and the error message displayed in the status line.

The program line should be changed from `MOV.W` to `MOVE.W`, so do this, then click on Assemble from the Program menu again. This time click on the Memory button, this means the program will be assembled into memory, instead of onto disk. This is very much faster and allows you to try things out immediately, which is exactly what we want. Clicking on the Assemble button will again assemble it, and after you press a key to return to the editor it's ready to run.

The assembly worked this time, so click on Run from the Program menu, and what happens? Not a lot it would seem, except that a couple of bombs appeared briefly on the screen - oh, there's a bug.

The tool for finding bugs is a debugger, so click on Debug from the Program menu. The debugger is described more fully later on, but for now we just want to run the program from the debugger to 'catch' the bombs and find out what causes them, so press `Ctrl-R`.

After a brief delay the message `Bus Error` will appear in the bottom window, with the disassembly window showing the current instruction

```
        MOVE.W   1,-(A7)
```

This will cause a bus error because location 1 is in protected memory which cannot be accessed in user mode - there should a hash sign before the 1 to put the immediate value of 1 on the stack. To return to the editor press `Ctrl-C`, so we can fix this bug in the source code.

Press Alt-T, to go to the top of the file, then click on Find from the Search menu. We are going to find the errant instruction so enter

```
move.w
```

then press Return to start the search. The first occurrence has a hash sign, so press Alt-N to find the next, which is the line

```
move.w    c_conin,-(a7)
```

Ahah! - this is the one, so add a hash to change it to

```
move.w    #c_conin,-(a7)
```

then assemble it again. If you click on Run from the Program menu you should see the message, and pressing any key will return you to the editor.

However, did you notice how messy the screen was - the desktop pattern looked very untidy and you possibly got mouse 'droppings' left on the screen. This was because DEMO is a TOS program running with a GEM screen - to change this, click on Run with GEM from the Program menu - the check mark next to it should disappear. If you select Run again you can see the display is a lot neater, isn't it? If you run a GEM program you must ensure the check mark *is* there beforehand, otherwise nasty things can happen.

Although the program now works we shall use MonST, the debugger, to trace through the program, step by step. To do this click on Debug from the Program menu, and the debugger will appear with the message Breakpoint, showing your program.

There are various windows, the top one displaying the machine registers, the second a disassembly of the program, the third some other memory, and the bottom window displaying various messages.

If you look at window 2, the disassembly window, you will see the current instruction, which in our case is

```
MOVE.L    #string,-(A7)
```

As the debug option was specified in the source code any symbols will appear in the debugger.

Let's check the area around string. Press Alt-3 and you should see window 3's title inverted. Next press Alt-A and a dialog box will appear, asking Window start address? - to this enter

```
string
```

(it must be in lower-case) and press Return. This will re-display window 3 at that address, showing the message in both hex and ASCII.

To execute this MOVE instruction press Ctrl-Z. This will execute the instruction then the screen will be updated to reflect the new values of the program counter and register A7. If you press Ctrl-Z again the MOVE.W instruction will be executed. If you look at the hex display next to A7 you should see a word of 9, which is what you would expect after that instruction.

The next instruction is TRAP #1, to call GEMDOS to print a string, but hang on - would we notice a string printed in the middle of the MonST display? Never fear, MonST has its own screen to avoid interference with your program's, to see this press the V key, which will show a blank screen, ready for your program. Pressing any other key will return you to MonST.

To execute this call press Ctrl-Z, which will have printed the string. To prove it press V again, then any key to return to MonST.

Press Ctrl-Z twice more until you reach the next Trap. This one waits for a key press so hit Ctrl-Z and the program display will automatically appear, waiting for a key. When you're ready, press the q key. You will return to MonST and if you look at the register window the low 8 bits of register D0 should be $71, the ASCII code for q, and next to that it will be shown as q (unless in low-resolution).

The final Trap quits the program, so to let it run its course press Ctrl-R, you will then return to the editor as the program has finished.

Note the way we have used the courier font to indicate text or values that appear on screen or input to be typed from the keyboard. Also, Ctrl-X means hold the Ctrl key down on the keyboard and press X, while Return indicates that you should press the Return key on the keyboard. These conventions will be used throughout the manual.

CHAPTER 2
Screen Editor

Introduction

To enter and assemble your programs you need an editor of some sort and an assembler. GenST combines both of these functions together in one integrated program, giving a GEM-driven full-screen editor and a fast, full-specification assembler. It also allows you to run your assembled programs directly from memory without having to quit the program or do a disk access and to access the debugger at the press of a key. The fact that all these features are combined in one program means that correcting errors and making changes is as fast as possible without the need for slow disk accesses and other programs.

This chapter details the use of the editor and how to assemble programs - it does not detail the assembler or the debugger themselves, they are covered in the following chapters.

To run GenST, double click on the `GENST2.PRG` icon from the Desktop. When it has loaded a menu bar will appear and an empty window will open, ready for you to enter and assemble your programs.

The Editor

A text editor is a program which allows you to enter and alter lines of text, store them on disk, and load them back again. There are two types of text editors: line editors, which treat each line separately and can be very tricky to use, and screen editors, which display your text a screen at a time. The latter tend to be much easier to use.

The editor section of GenST is a screen editor which allows you to enter and edit text and save and load from disk, as you would expect. It also lets you print some or all of your text, search and replace text patterns and use any of the ST's desk-accessories. It is GEM-based, which means it uses all the user-friendly features of GEM programs that you have become familiar with on your computer such as windows, menus and mice. However, if you're a die-hard, used to the hostile world of computers before the advent of WIMPs, you'll be pleased to know you can do practically everything you'll want to do from the keyboard without having to touch a mouse.

The editor is 'RAM-based', which means that the file you are editing stays in memory for the whole time, so you don't have to wait while your disk grinds away loading different sections of the file as you edit. As the ST range has so much memory, the size limitations often found in older computer editors don't exist with GenST; if you have enough memory you can edit files of over 300k (though make sure your disk is large enough to cope with saving it if you do!). As all editing operations, including things like searching, are RAM-based they act blindingly quickly.

When you have typed in your program it is not much use if you are unable to save it to disk, so the editor has a comprehensive range of save and load options, allowing you to save all or part of the text and to load other files into the middle of the current one, for example.

To get things to happen in the editor, there are various methods available to you. Features may be accessed in one or more of the following ways:

- Using a single key, such as a Function or cursor key;
- Clicking on a menu item, such as Save;
- Using a menu shortcut, by pressing the Alternate key (subsequently referred to as `Alt`) in conjunction with another, such as `Alt-F` for Find;
- Using the Control key (subsequently referred to as `Ctrl`) in conjunction with another, such as `Ctrl-A` for cursor word left;
- Clicking on the screen, such as in a scroll bar.

The menu shortcuts have been chosen to be easy and obvious to remember, while the Ctrl commands are based on those used in WordStar, and many other compatible editors since.

If at any time you get stuck, pressing the `Help` key will bring up a comprehensive display of the keys required for functions not visible in any menus.

A Few Words about Dialog Boxes

The editor makes extensive use of Dialog boxes, so it is worth recapping how to use them, particularly for entering text. The editor's dialog boxes contain *buttons*, *radio buttons*, and *editable text*.

Buttons may be clicked on with the mouse and cause the dialog box to go away. Usually there is a default button, shown by having a wider border than the others. Pressing `Return` on the keyboard is equivalent to clicking on the default button.

Radio buttons are groups of buttons of which only one may be selected at a time - clicking on one automatically de-selects all the others.

Editable text is shown with a dotted line, and a vertical bar marks the cursor position. Characters may be typed in and corrected using the `Backspace`, `Delete` and cursor keys. You can clear the whole edit field by pressing the `Esc` key. If there is more than one editable text field in a dialog box, you can move between them using the ↓ and ↑ keys or by clicking near them with the mouse.

Some dialog boxes allow only a limited range of characters to be typed into them - for example the Goto Line dialog box only allows numeric characters (digits) to be entered.

Entering text and Moving the cursor

Having loaded GenST, you will be presented with an empty window with a status line at the top and a flashing black block, which is the *cursor*, in the top left-hand corner.

The status line contains information about the cursor position in the form of Line and Column offsets as well as the number of bytes of memory which are free to store your text. Initially this is displayed as 59980, as the default text size is 60000 bytes. You may change this default if you wish, together with various other options, by selecting Preferences, described later. The 'missing' 20 bytes are used by the editor for internal information. The rest of the status line area is used for error messages, which will usually be accompanied by a 'ping' noise to alert you. Any message that gets printed will be removed subsequently when you press a key.

To enter text, you type on the keyboard. As you press a key it will be shown on the screen and the cursor will be advanced along the line. If you are a very good typist you may be able to type faster than the editor can re-display the line; if so, don't worry, as the program will not lose the keystrokes and will catch up when you pause. At the end of each line you press the `Return` key (or the `Enter` key on the numeric pad) to start the next line. You can correct your mistakes by pressing the `Backspace` key, which deletes the character to the left of the cursor, or the `Delete` key, which removes the character the cursor is over.

The main advantage of a computer editor as opposed to a normal typewriter is its ability to edit things you typed a long time ago. The editor's large range of options allow complete freedom to move around your text at will.

Cursor keys

To move the cursor around the text to correct errors or enter new characters, you use the cursor keys, labelled ← → ↑ and ↓. If you move the cursor past the right-hand end of the line this won't add anything to your text, but if you try to type some text at that point the editor will automatically add the text to the real end of the line. If you type in long lines the window display will scroll sideways if necessary.

If you cursor up at the top of a window the display will either scroll down if there is a previous line, or print the message `Top of file` in the status line. Similarly if you cursor down off the bottom of the window the display will either scroll up if there is a following line, or print the message `End of file`.

You can move the cursor on a character basis by clicking on the arrow boxes at the end of the horizontal and vertical scroll bars.

For those of you used to WordStar, the keys `Ctrl-S`, `Ctrl-D`, `Ctrl-E` and `Ctrl-X` work in the same way as the cursor keys.

To move immediately to the start of the current line, press `Ctrl` ←, and to move to the end of the current line press `Ctrl` →.

To move the cursor a word to the left, press `Shift` ← and to move a word to the right press `Shift` →. You cannot move past the end of a line with `Shift` →. A word is defined as anything surrounded by a space, a tab or a start or end of line. The keys `Ctrl-A` and `Ctrl-F` also move the cursor left and right on a word basis.

To move the cursor a page up, you can click on the upper grey part of the vertical scroll bar, or press `Ctrl-R` or `Shift` ↑. To move the cursor a page down, you can click on the lower grey part of the scroll bar, or press `Ctrl-C` or `Shift` ↓.

If you want to move the cursor to a specific position on the screen you may move the mouse pointer to the required place and click (There is no WordStar equivalent for this feature!).

Tab key

The `Tab` key inserts a special character (ASCII code 9) into your text, which on the screen looks like a number of spaces, but is rather different. Pressing `Tab` aligns the cursor onto the next 'multiple of 8' column, so if you press it at the start of a line (column 1) the cursor moves to the next multiple of 8, +1, which is column 9. Tabs are very useful indeed for making items line up vertically and its main use in GenST is for making instructions line up. When you delete a tab the line closes up as if a number of spaces had been removed. The advantage of tabs is that they take up only 1 byte of memory, but can show on screen as many more, allowing you to tabulate your program neatly. You can change the tab size before or after loading GenST using the `Preferences` command described shortly.

Backspace key

The `Backspace` key removes the character to the left of the cursor. If you backspace at the very beginning of a line it will remove the 'invisible' carriage return and join the line to the end of the previous line. Backspacing when the cursor is past the end of the line will delete the last character on the line, unless the line is empty in which case it will re-position the cursor on the left of the screen.

Delete key

The `Delete` key removes the character under the cursor and has no effect if the cursor is past the end of the current line.

Goto a particular line

To move the cursor to a specific line in the text, click on Goto line... from the Options menu, or press `Alt-G`. A dialog box will appear, allowing you to enter the required line number. Press `Return` or click in the OK button to go to the line or click on Cancel to abort the operation. After clicking on OK the cursor will move to the specified line, re-displaying if necessary, or give the error `End of file` if the line doesn't exist.

Another fast way of moving around the file is by dragging the slider on the vertical scroll bar, which works in the usual GEM-like fashion.

Go to top of file

To move to the top of the text, click on Goto Top from the Options menu, or press `Alt-T`. The screen will be re-drawn if required starting from line 1.

Go to end of file

To move the cursor to the start of the very last line of the text, click on Goto Bottom, or press `Alt-B`.

Quitting GenST

To leave GenST, click on Quit from the File menu, or press `Alt-Q`. If changes have been made to the text which have not been saved to disk, an alert box will appear asking for confirmation. Clicking on Cancel will return you to the editor, while clicking on OK will discard the changes and return you to the Desktop.

Deleting text

Delete line

The current line can be deleted from the text by pressing `Ctrl-Y`.

Delete to end of line

The text from the cursor position to the end of the current line can be deleted by pressing `Ctrl-Q`. (This is equivalent to the WordStar sequence `Ctrl-Q Y`).

UnDelete Line

When a line is deleted using either of the above commands it is preserved in an internal buffer, and can be re-inserted into the text by pressing `Ctrl-U`, or the Undo key. This can be done as many times as required, particularly useful for repeating similar lines or swapping over individual lines.

Delete all the text

To clear out the current text, click on Clear from the File menu, or press `Alt-C`. If you have made any changes to the text that have not been saved onto disk, a confirmation is required and the requisite alert box will appear. Clicking on OK will delete the text, or Cancel will abort the operation.

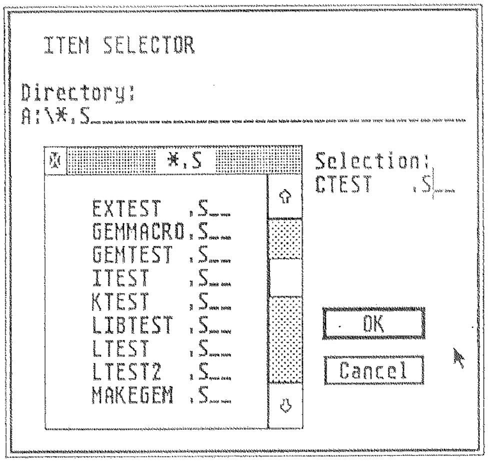
Disk Operations

It is no use being able to type in text if you are unable to save it anywhere permanently, or load it back subsequently, so the editor has a comprehensive set of features to read from and write to disk.

GEM File Selector

Before describing the commands, it is best to detail the GEM File Selector, which is a consistent way for users to select filenames from disk. It is the same in all programs, so if you have used it before then you can skip to the next section.

Figure 2.1 shows an example of the file selector box. At the top the current drive, directory and type selection is shown. To the right is a space for the actual filename, with OK and Cancel buttons below it and a window taking up most of the remainder of the selector. This window displays all of the filenames that correspond to the drive and directory above.



```
EXTEST  .S
GEMMACRO.S
GEMTEST .S
ITEST   .S
KTEST   .S
LIBTEST .S
LTEST   .S
LTEST2  .S
MAKEGEM .S
```

Figure 2.1 - the GEM File Selector

To select a filename, to save or to load, you can either click on the name shown in the window, perhaps after using the scroll bar to go up or down the list, or type it in at the Selection area. If you click on a filename it will automatically be copied into the Selection area. Clicking on OK or pressing `Return` will choose that particular filename, or once you get used to the selector you may double-click on the filename, obviating the need to click on OK or to press `Return`.

If the file you want is not in the sub-directory shown, you can go down a directory level by clicking on the directory name in the window, or you can go up a directory by clicking on the close box of the filename window. By default, GenST displays all files ending in `.S`, as this is the usual extension for assembly language programs. If you want to change this, you have to edit the Directory string and replace the `.S` with the extension of you choice, such as `.ASM`. If you want to be shown all the files, regardless of extension, replace the `.S` with `.*`. If you do edit the Directory string you need to click in the filename area of the window to tell GEM to re-display the filenames. If you want to change the disk drive specifier, you should click on the Directory string with the mouse (or press ↑), edit it to suit and click in the filename area of the window.

Note In all pre-blitter versions of the ST ROMs there is a bug which means that if you press _ (underline) when the cursor is in the Directory string the machine will crash!

Saving Text

To save the text you are currently editing, click on Save As from the File menu, or press `Alt-S`. The standard GEM File Selector will appear, allowing you to select a suitable disk and filename. Clicking OK or pressing `Return` will then save the file onto the disk. If an error occurs a dialog will appear showing a TOS error number, the exact meaning of which can be found in Appendix A.

If you click on Cancel the text will not be saved. Normally if a file exists with the same name it will be deleted and replaced with the new version, but if Backups are selected from the Preferences options then any existing file will be renamed with the extension `.BAK` (deleting any existing `.BAK` file) before the new version is saved.

Save

If you have already done a Save As (or a Load), GenST will remember the name of the file and display it in the title bar of the window. If you want to save it without having to bother with the file selector, you can click on Save on the File menu, or press `Shift-Alt-S`, and it will use the old name and save it as above. If you try to Save without having previously specified a filename you will be presented with the File Selector, as in Save As.

Loading Text

To load in a new text file, click on Load from the File menu, or press `Alt-L`. If you have made any changes that have not been saved, a confirmation will be required. The GEM file selector will appear, allowing you to specify the disk and filename. Assuming you do not Cancel, the editor will attempt to load the file. If it will fit, the file is loaded into memory and the window is re-drawn. If it will not fit an alert box will appear warning you, and you should use Preferences to make the edit buffer size larger, then try to load it again.

Inserting Text

If you want to read a file from disk and insert it at the current position in your text click on Insert File from the File menu, or press `Alt-I`. The standard GEM file selector will appear and assuming that you do not cancel, the file will be read from the disk and inserted, memory permitting.

Searching and Replacing Text

To find a particular section of text click on Find from the Search menu, or press Alt-F. A dialog box will appear, allowing you to enter the Find and Replace strings. If you click on Cancel, no action will be taken; if you click Next (or press Return) the search will start forwards, while clicking on Previous will start the search backwards. If you do not wish to replace, leave the Replace string empty. If the search was successful, the screen will be re-drawn at that point with the cursor positioned at the start of the string. If the search string could not be found, the message Not found will appear in the status area and the cursor will remain unmoved. By default the search is always case-independent, so for example if you enter the search string as test you could find the words TEST, Test and test. If you click on the UPPER & lower case Different button the search will be case-dependent.

To find the next occurrence of the string click on Find Next from the Search menu, or press Alt-N. The search starts at the position just past the cursor.

To search for the previous occurrence of the string click on Find Previous from the Search menu, or press Alt-P. The search starts at the position just before the cursor.

Having found an occurrence of the required text, it can be replaced with the Replace string by clicking on Replace from the Search menu, or by pressing Alt-R. Having replaced it, the editor will then search for the next occurrence.

If you wish to replace every occurrence of the find string with the replace string from the cursor position onwards, click on Replace All from the Search menu. During the global replace the Esc key can be used to abort and the status area will show how many replacements were made. There is deliberately no keyboard equivalent for this to prevent it being chosen accidentally.

Block Commands

A *block* is a marked section of text which may be copied to another section, deleted, printed or saved onto disk. The function keys are used to control blocks.

Marking a block

The start of a block is marked by moving the cursor to the required place and pressing key `F1`. The end of a block is marked by moving the cursor and pressing key `F2`. The start and end of a block do not have to be marked in a specific order - if it is more convenient you may mark the end of the block first.

A marked block is highlighted by showing the text in reverse. While you are editing a line that is within a block this highlighting will not be shown but will be re-displayed when you leave that line or choose a command.

Saving a block

Once a block has been marked, it can be saved by pressing key `F3`. If no block is marked, the message `What blocks!` will appear. If the start of the block is textually after its end the message `Invalid block!` will appear. Both errors abort the command. Assuming a valid block has been marked, the standard GEM file selector will appear, allowing you to select a suitable disk and filename. If you save the block with a name that already exists the old version will be overwritten - no backups are made with this command.

Copying a block

A marked block may be copied, memory permitting, to another part of the text by moving the cursor to where you want the block copied and pressing key `F4`. If you try to copy a block into a part of itself, the message `Invalid block` will appear and the copy will be aborted.

Deleting a block

A marked block may be deleted from the text by pressing `Shift-F5`. The shift key is deliberately required to prevent it being used accidentally. A deleted block is remembered, memory permitting, in the *block buffer*, for later use.

Note This is on a different key to that used in GenST in versions before 2.0.

Copy block to block buffer

The current marked block may be copied to the block buffer, memory permitting, by pressing `Shift-F4`. This can be very useful for moving blocks of text between different files by loading the first, marking a block, copying it to the block buffer then loading the other file and pasting the block buffer into it.

Pasting a block

A block in the block buffer may be pasted at the current cursor position by pressing `F5`.

Note The block buffer will be lost if the edit buffer size is changed or an assembly occurs.

Printing a block

A marked block may be sent to the printer by clicking on Print Block from the File menu, or by pressing `Alt-W`. An alert box will appear confirming the operation and clicking on OK will print the block. The printer port used will depend on the port chosen with the *Install Printer* desk accessory, or will default to the parallel port. Tab characters are sent to the printer as a suitable number of spaces, so the net result will normally look better than if you print the file from the Desktop.

If you try to Print when no block is marked at all then the whole file will be printed.

Block markers remain during all editing commands, moving where necessary, and are only reset by the commands New, Delete block, and Load.

Miscellaneous Commands

About GenST2

If you click on About GenST2... from the Desk menu, a dialog box will appear giving various details about GenST. Pressing `Return` or clicking on OK will return you to the editor.

Help Screen

The key equivalents for the commands not found in menus can be seen by pressing the `Help` key, or `Alt-H`. A dialog box will appear showing the WordStar and function keys, as well as the free memory left for the system.

Preferences

Selecting Preferences... from the Options menu will produce a dialog box allowing you to change several editor settings:

Tabs

By default, the tab setting is 8, but this may be changed to any value from 2 to 16.

Text Buffer Size

By default the text buffer size is 60000 bytes, but this can be changed from 4000 to 999000 bytes. This determines the largest file size that can be loaded and edited. Care should be taken to leave sufficient room in memory for assembly or running MonST - pressing the `Help` key displays free system memory, and for assembly or debugging this should always be at least 100k bytes. Changing the editor workspace size will cause any text you are currently editing to be lost, so a confirmation is required if it has not been saved.

Numeric Pad

The Numeric Pad option allows the use of the numeric keypad in an IBM-PC-like way allowing single key presses for cursor functions, and defaults to Cursor pad mode. The keypad works as shown in **Figure 2.2** below.

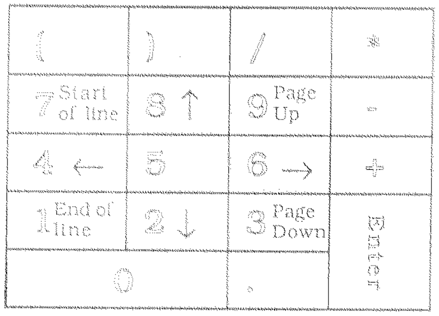


Figure 2.2 Numeric Keypad

This feature can be disabled, if required, by clicking on the Numbers button.

Backups

By default the editor does not make backups of programs when you save them, but this can be turned on by clicking on the Yes radio button.

Auto Indenting

It can be particularly useful when editing programs to indent subsequent lines from the left, so the editor supports an auto-indent mode. When active, an indent is added to the start of each new line created when you press `Return`. The contents of the indent of the new line is taken from the white space (i.e. tabs and/or spaces) at the start of the previous line.

Cursor

By default the GenST cursor flashes but this can be disabled if required.

Load MonST

By default a copy of MonST is loaded during the editor initialisation, allowing it to be accessed at the press of a key. Should this not be required it can be disabled with this option. This will save around 24k of memory. The new value of this option will only have an effect if you save the preferences and re-execute the editor.

Saving Preferences

If you click on the Cancel button any changes you make will be ignored. If you click on the OK button the changes specified will remain in force until you quit the editor. If you would like the configuration made permanent then click on the Save button, which will create the file `GENST2.INF` on your disk. Next time you run GenST the configuration will be read from that file.

In addition to saving the editor configuration the current setting from the Assembly Options dialog box are also saved.

Assembling & Running Programs

All assembly and run options can be found on the Program menu.

Assembly

To assemble the program you are currently editing click on Assemble from the Program menu, or press `Alt-A`. The meaning of the various options, together with the assembly process itself is detailed in the next chapter. The only option covered here is the Output to option.

GenST can assemble to disk, to memory, or nowhere - assembling to nowhere is ideal for syntax checking while assembly to memory is much faster than to disk and good for trying things out quickly. When you assemble to memory you have to specify the maximum program size in the Max: entry in the dialog box - normally this is 20k, enough for an average program with debug or a large program with no debug. This number determines the program buffer size, used by the assembler to store your assembled program. If you get the *program buffer full* error when you assemble something you should change the number to be larger. There is of course a penalty for this - the bigger the program buffer size the smaller the amount of memory left for the assembler itself to use while assembling your program. If the assembler itself aborts with *Out of memory* it means there is not enough left for a complete assembly - you should reduce the buffer size, or if this still fails you will have to assemble to disk.

When you assemble to disk the program buffer size number is ignored, giving maximum room in memory for the assembler itself. If you haven't saved your program source code yet the file will be based on the name `NONAME`.

After you click on Assemble or press `Return` the assembly process will start, described more fully in the next chapter. At the end of the assembly the program will wait for a key press, allowing you to read any messages produced, before returning you to the editor. If there were any errors the editor will go to the first erroneous line and display the error message in the status bar. Subsequent errors (and warnings) may be investigated by pressing `Alt-J`.

Running Programs

If you click on Run from the Program menu or press `Alt-X` (eXecute) you can then run a program previously assembled into memory. When your program finishes it will return you to the editor. If the assembly didn't complete normally for any reason then it is not possible to run the program.

If your program crashes badly you may never return to the editor, so if in doubt save your source code before using this, or the following command.

Note

If only non-fatal errors occurred during assembly (e.g. undefined symbols) you will still be permitted to run your program, at your own risk.

Please Note

When issuing a Run command from the editor the machine may seem to 'hang up' and not run the program. This occurs if the mouse is in the menu bar area of the screen and can be corrected by moving the mouse. Similarly when a program has finished running, the machine may not return to the editor. Again, moving the mouse will cure the problem. This is due to a feature of GEM beyond our control.

Debug

If you wish to debug a program previously assembled to memory click on Debug from the Program menu, or press `Alt-D`. This will invoke MonST to debug your program, included any debugging information specified. Pressing `Ctrl-C` from MonST will terminate both your program and the debugger. The screen type selected is determined by the Run with GEM option, described below.

Note If the Load MonST option is disabled this option is not available and the menu item is disabled.

MonST

Clicking on MonST from the Program menu, or pressing `Alt-M`, will invoke MonST in a similar way to if it was invoked by double-clicking on the program icon from the Desktop, but instantly, as it is already in memory. You will return to the editor on termination of the debugger. The screen type selected is determined by the Run with GEM option, described below.

Note If the Load MonST option is disabled this option is not available and the menu item is disabled.

Run with GEM

Normally when the commands Run, Debug or MonST are used the screen is initialised to the normal GEM type, with a blank menu bar and patterned desktop. However if running a TOS program this can be changed to a blank screen with flashing cursor, by clicking on Run with GEM, or by pressing `Alt-K`. A check-mark next to the menu item means GEM mode, no check mark means TOS mode. The current setting of this option is remembered if you Save Preferences.

Note Running a TOS program in GEM mode will look messy but work, but running a GEM program in TOS mode can crash the machine.

Jump to Error

During an assembly any warnings or errors that occur are remembered, and can be recalled from the editor. Clicking on Jump to error from the Program menu, or pressing `Alt-J` will move the cursor to the next line in your program which has an error, and display the message in the status line of the window. You can step to the next one by pressing `Alt-J` again, and so on, letting you correct errors quickly and easily. If there are no further errors when you select this option the message `No more errors` will appear, or if there are no errors at all the message `What errors!` will appear.

Run Other...

This option lets you run other programs from within the editor, then return to it when they finish. Its main use is to allow you to run programs you have assembled to disk, or to run the linker, without having to quit to the Desktop and double-click them. You can run both TOS and GEM programs using this command, subject to available memory. When you click on Run Other... from the Program menu you will first be warned if you have not saved your source code, then the GEM File Selector will appear, from which you should select the program you wish to run. If it is a `.TOS` or `.TTP` program you will be prompted for a command line, then the screen initialised suitably.

Note Screen initialisation depends on the filename extension, *not* the current Run with GEM option setting.

Window Usage & Desk Accessories

The GEM Editor Window

The window used by the editor works like all other GEM windows, so you can move it around by using the move bar on the top of it, you can change its size by dragging on the size box, and make it full size (and back again) by clicking on the full box. Clicking on the close box is equivalent to choosing Quit from the File menu.

Desk Accessories

If your ST system has any desk accessories, you will find them in the Desk menu. If they use their own window, as *Control Panel* does, you will find that you can control which window is at the front by clicking on the one you require. For example, if you have selected the Control Panel it will appear in the middle of the screen, on top of the editor window. You can then move it around and if you wish it to lie 'behind' the editor window, you can do it by clicking on the editor window, which brings it to the front, then re-sizing it so you can see some part of the control panel's window behind it. When you want to bring that to the front just click on it and the editor window will go behind. The editor's cursor only flashes and the menus only work when the editor's window is at the front.

Automatic Double Clicking

You may configure GenST to be loaded automatically whenever a source file is double-clicked from the Desktop, using the Install Application option.

To do this you first have to decide on the extension you are going to use for your files, which we recommend to be .S for source files. Having done this, go to the Desktop, and click once on GENST2.PRG to highlight it. Next click on Install Application from the Options menu and a dialog box will appear. You should set the Document Type to be S (or whatever you require), and leave the GEM radio button selected. Finally click on the OK button (if you press Return it will be taken as Cancel).

Having done this, you will return to the Desktop. To test the installation, double-click on a file with the chosen extension which must be on the same disk and in the same folder as GenST and the Desktop will load GenST, which will in turn load in the file of your choice ready for editing or assembly.

Note To make the configuration permanent, you have to use the Save Desktop option.

Saved! Desk Accessory Users

If you use the PATH feature of the **Saved! by HiSoft** desk accessory then the restriction of having your data files in the same folder and drive as your assembler described above is not relevant. The editor looks for the GENST2.INF configuration file firstly in the current directory (which is the folder where you double-clicked on the data file), then using the system path. Saving the editor preferences will put the .INF file in the same place it was loaded from, or if it was not found then it will be put in the current directory.

You may invoke **Saved!** from within the editor at any time by pressing Shift-Clr. This will only work if the desk accessory is called SAVED!.ACC or SAVED.ACC on your boot disk.

CHAPTER 3
Macro Assembler

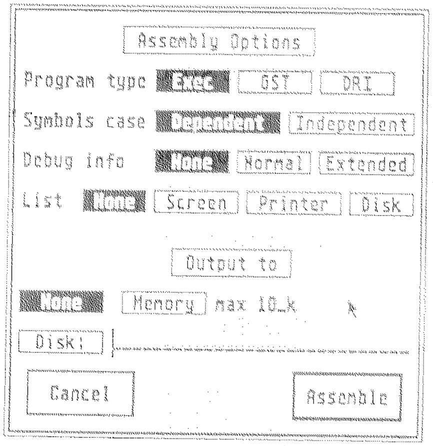
Introduction

GenST is a powerful, fast, full specification assembler, available instantly from within the editor or as a stand-alone program. It converts the text typed or loaded into the editor, optionally together with files read from disk, into a binary file suitable for immediate execution or linking, or into a memory image for immediate execution from the editor.

Invoking the Assembler

From the Editor

The assembler is invoked from the editor by clicking on Assemble from the Program menu, or by pressing `Alt-A`. A dialog box appears which looks like **Figure 3.1** below.



Assembly Options

Program type [✓ Exec] GST DRI

Symbols case [✓ Dependent] Independent

Debug info [✓ None] Normal Extended

List [✓ None] Screen Printer Disk

Output to

[✓ None] Memory max 10 k

Disk:

Cancel   Assemble

Figure 3.1 - the Assembly Options dialog box

Program Type This lets you select between executable, GST or DRI format output. The differences between these are detailed later.

Symbols case This lets you select whether labels are case dependent or not. If case Dependent is selected then `Test` and `test` would be different labels, if case Independent is selected then they would be the same.

Debug Info If you wish to debug your program using your original symbols you can select Normal or Extended debug modes. The advantage of extended debug is that up to 22 characters of each symbol are included in the debug information, whereas normal mode restricts symbols to eight characters.

List selecting Printer will divert the assembly listing to the current printer port, or selecting Disk will send the listing to a file based on the source filename, but with the extension `.LST`

Output To This lets you select where the output file is to be created. None means it is 'thrown away', ideal for syntax checking a program; Memory means it is assembled into a buffer allowing it to be run or debugged instantly from the editor without having to create a disk file; Disk means a file will be created. The selection of the name of this file can be left to the assembler, using rules described shortly.

The first two options may also be specified in the source file using the OPT directive.

Having selected your required options you should click on the Assemble button (or press `Return`) to start the assembly. At the end of assembly you should press any key to return to the editor. If any errors occurred the cursor will be positioned on the first offending line.

Stand-Alone Assembler

If the `.TTP` version of the assembler is invoked the without a command line the programmer will be asked for one, conforming to the rules below, or press `Return` to abort. At the end of assembly there will be a pause, pressing any key will exit the program. If a command line has been supplied the assembler will not wait for a key as it assumes it has been run from a CLI or batch file.

The command line should be of the form

`mainfile <-options> [-options]`

The **mainfile** should be the name of the file requiring assembly and if no extension is specified defaults to `.S`. Options should follow this denoted by a - sign then an alphabetic character. Allowed options are shown below together with equivalent OPT directives:

`B`	no binary file should be created
`C`	case insensitive labels (`OPT C-`)
`D`	debug (`OPT D+`)
`L`	GST linkable code (`OPT L+`)
`L2`	DRI linkable code (`OPT L2`)
`O`	specify output filename (should follow *immediately* after `O`)
`P`	specify listing filename (should follow *immediately* after `P`), defaults to source filename with extension of `.LST`
`Q`	pause for key press after assembly
`X`	extended debugging (`OPT X+`)

The default is to create a executable binary file with a name based on the source file and output file type, no listing, with case sensitive labels. For example,

```
test -b
```
assembles `test.s` with no binary output file

```
test -om:test.prg -p
```
assembles `test.s` into a binary file `m:test.prg` and a listing file to `test.lst`

```
test -l2dpprn:
```
assembles `test.s` into DRI linkable code with debug and a listing to the parallel port. (A listing to the serial port can be obtained by specifying `AUX:` as the listing name).

Output Filename

GenST has certain rules regarding the calculation of the output filename, using a combination of that specified at assembly time (either in the Disk: filename field in the dialog box or using the `-o` option on the command line) and the OUTPUT directive:

```
If an output filename is explicitly given at assembly time then
        name=explicit filename
else
        if the OUTPUT directive has not been used then
                name=source filename + .PRG, .BIN or .O
        elseif the OUTPUT directive specifies an extension then
                name=source filename + extension in OUTPUT
        else
                name=name in OUTPUT
```

Assembly Process

GenST is a two-pass assembler; during the first pass it scans all the text in memory and from disk if required, building up a symbol table. If syntax errors are found on the first pass assembly these will be reported and assembly will stop at the end of the first pass, otherwise, during the second pass the instructions are converted into bytes, a listing may be produced if required and a binary file can be created on the disk. During the second pass any further errors and warnings will be shown, together with a full listing and symbol table if required.

During assembly, any screen output can be paused by pressing `Ctrl-S`, pressing `Ctrl-Q` will resume it. Assembly may be aborted by pressing `Ctrl-C`, although doing so will make any binary file being created invalid as it will be incomplete and should not be executed.

Assembly to Memory

To reduce development time GenST can assemble programs to memory, allowing immediate execution or debugging from the editor. To do this a *program buffer* is used, the size of which is specified in the Assembly Options dialog box. If no debug option is specified the size given can be just a little larger than the output program, but if either form of debug is required a much larger buffer may be needed.

A program running from memory is just like any normal GEMDOS program and should terminate using either *pterm* or *pterm0* GEMDOS calls, for example

```
        clr.w  -(a7)
        trap   #1
```

Programs may self-modify if required as a re-executed program will be in its original state.

The program buffer size and current assembly options can be made the default on re-loading the editor if Save Preferences is used.

Binary file types

There are six types of binary files which may be produced by GenST, for different types of applications. They are distinguished by the extension on the filename:

`.PRG`	GEM-type application i.e. one that uses windows
`.TOS`	TOS-type application i.e. one that doesn't use windows
`.TTP`	TOS-type application that requires a command line
`.ACC`	desk accessory program file
`.BIN`	non-executable file suitable for linking with GST-format files and libraries
`.O`	non-executable file suitable for linking with DRI-format files and libraries

It can also assemble executable code directly to memory when using the integrated version allowing very fast edit-assemble-debug-run times.

The first three are double-clickable, can be run from the Desktop and are known as *executable*. They differ in the initialisation performed before the execution. With `.PRG` files the screen is cleared to the Desktop's pattern, while with the other two the screen clears to white, the flashing cursor appears and the mouse is disabled. When you double-click a `.TTP` file the Desktop will prompt you for a command line to pass to it.

.ACC files are executable files but cannot be double-clicked on from the Desktop. They will only run successfully when executed by the AES during the boot sequence of the machine.

.BIN and .O files cannot be run immediately, but have to be read into a linker, usually with other sections, and are known as *linkable object modules*. There are two different linker formats on the ST, .BIN files are GST format, .O files are DRI format. The differences between these are discussed later in this chapter.

The above extensions are not absolute rules; for example, if you have a TOS type program you may give it a .PRG extension and use the Install Application function from the Desktop, but it's usually much easier to use the normal extensions. One exception is for programs which are designed to be placed in the AUTO folder so they execute during the boot sequence. They have to be TOS type programs, but need the extension .PRG for the boot sequence to find them.

Note Certain versions of the French ST ROMs do not recognise .TTP files from the Desktop so they have to be renamed .TOS then installed as TOS Takes Parameters.

Types of code

Unlike most 8-bit operating systems, but like most 16-bit systems, an executable program under GEMDOS will not be loaded at a particular address but, instead, be loaded at an address depending on the exact free memory configuration at that time.

To get around the problem of absolute addressing the ST file format includes *relocation information* allowing GEMDOS to relocate the program after it has loaded it but before running it. For example the following program segment

```
        move.l  #string,a0
        .
        .
        .
string  dc.b    'Press any key',0
```

places the absolute address of `string` into a register, even though at assembly time the real address of `string` cannot possibly be known. Generally the programmer may treat addresses as absolute even though the real addresses will not be known to him, while the assembler (or linker) will look after the necessary relocation information.

Note For certain programs, normally games or for cross-machine development an absolute start address may be required, for this reason the ORG directive is supported.

The syntax of the assembler will now be described.

Assembler Statement Format

Each line that is to be processed by the assembler should have the following format:

Label	Mnemonic	Operand(s)	Comment
`start`	`move.l`	`d0,(a0)+`	`store the result`

Exceptions to this are comment lines, which are lines starting with an asterisk or semi-colon, and blank lines, which are ignored. Each field has to be separated from the others by *white space* - any number or mixture of space and tab characters.

Label field

The label should normally start at column 1, but if a label is required to start at another position then it should be followed immediately by a colon (:). Labels are allowed on all instructions, but are prohibited on some assembler directives, and absolutely required on others. A label may start with the characters `A-Z`, `a-z`, or underline (_), and may continue with a similar set together with the addition of the digits `0-9` and the period (.).

Labels starting with a period are *local labels*, described later. Macro names and register equate symbols may not have periods in them, though macro names may start with a period. By default the first 127 characters of labels are significant, though this can be reduced if required. Labels should not be the same as register names, or the reserved words `SR`, `CCR` or `USP`.

By default labels are case-sensitive though this may be changed.

Some example legal labels are:

```
test, Test, TEST, _test, _test.end, test5, _5test
```

Some example illegal labels are:

```
5test, _&e, test>,
```

There are certain reserved symbols in GenST, denoted by starting with two underline characters. These are `__LK`, `__RS` and `__G2`.

Mnemonic Field

The mnemonic field comes after the label field and can consist of 68000 assembler instructions, assembler directives or macro calls. Some instructions and directives allow a size specifier, separated from the mnemonic by a period. Allowed sizes are `.B` for byte, `.W` for word, `.L` for long and `.S` for short. Which size specifiers are allowed in each particular case depends on the particular instruction or directive. GenST is case-insensitive to mnemonic and directive names, so `Move` is the same as `move` and the same as `mOvE`, for example.

Operand Field

For those instructions or directives which require operands, this field contains one or more parameters, separated by commas. GenST is case-insensitive regarding register names so they may be in either or mixed case.

Comment Field

Any white space not within quotation marks found after the expected operand(s) is treated as a delimiter before the start of the comment, which will be ignored by the assembler.

Examples of valid lines

```
            move.l d0,(a0)+   comment is here
loop        TST.W    d0
lonely.label
 rts
* this is a complete line of comment
; and so is this
    indented: link A6,#-10 make room
a_string: dc.b 'spaces allowed in quotes' a string
```

Expressions

GenST allows complex expressions and supports full operator precedence, parenthesis and logical operators.

Expressions are of two types - *absolute* and *relative* - and the distinction is important. Absolute expressions are constant values which are known at assembly-time. Relative expressions are program addresses which are not known at assembly-time as the GEMDOS loader can put the program where it likes in memory. Some instructions and directives place restrictions on which types are allowed and some operators cannot be used with certain type-combinations.

Operators

The operators available, in decreasing order of precedence, are:

monadic minus (-) and plus (+)
bitwise not (~)
shift left (<<) and shift right (>>)
bitwise And (&), Or (!) and Xor (^)
multiply (*) and divide (/)
addition (+) and subtraction (-)
equality (=), less than (<), greater than (>)

The comparison operators are signed and return 0 if false or -1 ($FFFFFFFF) if true. The shift operators take the left hand operand and shift it the number of bits specified in the right hand operand and vacated bits are filled with zeroes.

This precedence can be overridden by the use of parentheses (and). With operators of equal precedence, expressions are evaluated from left-to-right. Spaces in expressions (other than those within quotes as ASCII constants) are not allowed as they are taken as the separator to the comment.

All expression evaluation is done using 32-bit signed-integer arithmetic, with no checking of overflow.

Numbers

Absolute numbers may be in various forms:

decimal constants, e.g. `1029`
hexadecimal constants, e.g. `$12f`
octal constants, e.g. `@730`
binary constants, e.g. `%1100010`
character constants, e.g. `'X'`

`$` is used to denote hexadecimal numbers, `%` for binary numbers, `@` for octal numbers and single `'` or double quotes `"` for character constants.

Character Constants

Whichever quote is used to mark the start of a string must also be used to denote its end and quotes themselves may be used in strings delimited with the same quote character by having it occur twice. Character constants can be up to 4 characters in length and evaluate to right-justified longs with null-padding if required. For example, here are some character constants and their ASCII and hex values:

`"Q"`	`Q`	`$00000051`
`'hi'`	`hi`	`$00006869`
`"Test"`	`test`	`$54657374`
`"it's"`	`it's`	`$6974277C`
`'it''s'`	`it's`	`$6974277C`

Strings used in DC.B statements follow slightly different justification rules, detailed with the directive later.

Symbols used in expressions will be either relative or absolute, depending on how they were defined. Labels within the source will be relative, while those defined using the EQU directive will be the same type as the expression to which they are equated.

The use of an asterisk (*) denotes the value of the program counter at the start of the instruction or directive and is always a relative quantity.

Allowed Type Combinations

The table in **Figure 3.2** summarises for each operator the results of the various type combinations of parameter and which combinations are not allowed. An **R** denotes a Relative result, an **A** denotes absolute and a * denotes that the combination is not allowed and will produce an error message if attempted.

	A op A	A op R	R op A	R op R
Shift operators	A	*	*	*
Bitwise operators	A	*	*	*
Multiply	A	*	*	*
Divide	A	*	*	*
Add	A	R	R	*
Subtract	A	*	R	A
Comparisons	A	*	*	A

Figure 3.2 - Allowed Type Combinations

The available addressing modes are shown in the table below. Please note that GenST is case-insensitive when scanning addressing modes, so `D0` and `a3` are both valid registers.

Form	Meaning	Example
Dn	data register direct	`D3`
An	address register direct	`A5`
(An)	address register indirect	`(A1)`
(An)+	address register indirect with post-increment	`(A5)+`
-(An)	address register indirect with pre-decrement	`-(A0)`
d(An)	address register indirect with displacement	`20(A7)`
d(An,Rn.s)	address register indirect with index	`4(A6,D4.L)`
d.W	absolute short address	`$0410.W`
d.L	absolute long address	`$12000.L`
d(PC)	program counter relative with offset	`NEXT(PC)`
d(PC,Rn.s)	program counter relative with index	`NEXT(PC,A2.W)`
#d	immediate data	`#26`

n	denotes register number from 0 to 7
d	denotes a number
R	denotes index register, either `A` or `D`
s	denotes size, either `W` or `L`, when omitted defaults to `W`

When using address register indirect with index the displacement may be omitted, for example

```
move.l   (a3,d2.l),d0
```

will assemble to the same as

```
move.l   0(a3,d2.l),d0
```

Special Addressing Modes

CCR	condition code register
SR	status register
USP	user stack pointer

In addition to the above, SP can be used in place of A7 in any addressing mode, e.g. `4(SP,D3.W)`

The data and address registers can also be denoted by use of the reserved symbols R0 through R15. R0 to R7 are equivalent to D0 to D7, R8 to R15 are equivalent to A0 to A7. This is included for compatibility with other assemblers.

Local Labels

GenST supports local labels, that is labels which are local to a particular area of the source code. These are denoted by starting with a period and are attached to the last non-local label, for example:

```
len1     move.l   4(sp),a0
.loop    tst.b    (a0)+
         bne.s    .loop
         rts
len2     move.l   4(sp),a0
.loop    tst.b    -(a0)
         bne.s    .loop
         rts
```

There are two labels called `.loop` in this code segment but the first is attached to `len1`, the second to `len2`.

The local labels `.W` and `.L` are not allowed to avoid confusion with the absolute addressing syntax.

Symbols and Periods

Symbols which include the period character can cause problems with GenST due to absolute short addressing.

The Motorola standard way of denoting absolute short addresses causes problems as periods are considered to be part of a label, best illustrated by an example:

```
        move.l vector.w,d0
```

where `vector` is an absolute value, such as a system variable. This would generate an undefined label error, as the label would be scanned as `vector.w`. To get around this, the expression, in this case a symbol, may be enclosed in brackets, e.g.

```
        move.l (vector).w,d0
```

though the period may still be used after numeric expressions, e.g.

```
        move.l $402.w,d0
```

Note GenST version 1 also supported the use of \ instead of a period to denote short word addressing and this is still supported in this version, but this is not recommended due to the potential for \W and \L to be mistaken for macro parameters.

Instruction Set

Word Alignment

All instructions with the exception of DC.B and DS.B are always assembled on a word boundary. Should you require a DC.B explicitly on a word boundary, the EVEN directive should be used before it. Although all instructions that require it are word-aligned, labels with nothing following them are not word-aligned and can have odd values. This is best illustrated by an example:

```
        nop             this will always be word aligned
        dc.b 'odd'
start
        tst.l (a0)+
        bne.s start
```

The above code would not produce the required result as start would have an odd value. To help in finding such instructions the assembler will produce an error if it finds an odd destination in a BSR or BRA operand. Note that such checks are not made on any other instructions, so it is recommended that you precede such labels with EVEN directives if you require them to be word-aligned. A common error is deliberately not to do this, as you know the preceding string is an even number of bytes long. All will be well until the day you change the string...

Instruction Set Extensions

The complete 68000 instruction set is supported and certain shorthands are automatically accepted, detailed below. A complete description of the instruction set including syntax and addressing modes can be found in any 68000 reference guide or in the supplied Pocket Guide

Condition Codes.

The alternate condition codes HS and LO are supported in Bcc, DBcc and Scc instructions, equivalent to CC and CS, respectively.

Branch instructions

To force a short branch use Bcc.B or Bcc.S, to force a word branch use Bcc.W or to leave to the optimiser use Bcc. Bcc.L is supported for compatibility with GenST 1 with a warning as it is, strictly speaking, a 68020 instruction. A BRA.S to the immediately following instruction is not allowed and is converted, with a warning, to a NOP. A BSR.S to the immediately following instruction is not allowed and will produce an error.

BTST Instruction

BTST is unique among bit-test instructions in supporting PC-relative addressing modes.

CLR Instruction

CLR An is not allowed, use SUB.L An,An instead (though note that the flags are not effected).

CMP Instruction

If the source is immediate then CMPI is used, else if the destination is an address register then CMPA is used, else if both addressing modes are post-increment then CMPM is used.

DBcc Instruction

DBRA is accepted as an equivalent to DBF.

ILLEGAL Instruction

This generates the op-code word $4AFC.

LINK Instruction

If the displacement is positive or not even a warning will be given.

MOVE from CCR Instruction

This is a 68010 and upwards instruction, converted with a warning to MOVE from SR.

MOVEQ Instruction

If the data is in the range 128-255 inclusive a warning will be given. It may be disabled by specifying a long size on the instruction.

Assembler Directives

Certain pseudo-mnemonics are recognised by GenST. These *assembler directives*, as they are called, are not (normally) decoded into opcodes, but instead direct the assembler to take certain actions at assembly time. These actions have the effect of changing the object code produced or the format of the listing. Directives are scanned exactly like executable instructions and some may be preceded by a label (for some it is obligatory) and may be followed by a comment. If you put a label on a directive for which it not relevant, the result will be undefined but will usually result in the label being ignored.

Each directive will now be described in turn. Please note that the case of a directive name is not important, though they generally are shown in upper case. The use of angled brackets (< >) in descriptions denote optional items, ellipses (...) denote repeated items.

Assembly Control

END

This directive signals that no more text is to be examined on the current pass of the assembler. It is not obligatory.

INCLUDE filename

This directive will cause source code to be taken from a file on disk and assembled exactly as though it were present in the text. The directive must be followed by a filename in normal GEMDOS format.

If the filename has a space in it the name should be enclosed in single or double quotes. A drive specifier, directory and extension may be included as required, e.g.

```
        include b:constants/header.s
```

Include directives may be nested as deeply as memory allows and if any error occurs when trying to open the file or read it, assembly will be aborted with a fatal error.

If no drive or pathname is specified, that of the main source file will be used when trying to open the file.

Note The more memory the better, GenST will read the whole of the file in one go if it can and not bother to re-read the file during pass 2.

INCBIN filename

This takes a given binary file and includes it, verbatim, into the output file. Suggested uses include screen data, sprite data and ASCII files.

OPT option <,option> ...

This allows control over various options within GenST and each one is denoted by a single character normally followed by a + or - sign. Multiple options may be specified, separated by commas. The allowed options are:

Option C - Case-sensitivity and significance

By default, GenST is sensitive to label case and labels are significant to 127 characters. This can be overridden, using `C-` for case-sensitivity, or `C+` for case-insensitivity. The significance may be specified by specifying a decimal number between the C and the sign, for example `C16+` denotes case insensitive labels with 16 character significance. This option may be used at any time in a program but normally only makes sense at the very beginning of a source file.

Option D - Debugging Information

The GEMDOS binary file format supports the inclusion of a symbol table at the end, which may be read by debuggers such as MonST and can be extremely useful when debugging programs. By default this is switched off but it may be activated with `D+` and deactivated with `D-`. The first 8 characters only of all relative labels are written to the file and will be upper-cased if GenST is in case-insensitive mode. The 8-character limit is due to the DRI standard file format and may be improved on by using the Extended Debug option, described below.

Option L - Linker Mode

The default for GenST is to produce executable code but `L+` will make it produce GST linkable code, `L2` will make it produce DRI linkable code, or `L-` will make it revert to executable. This directive must be the *very first line* in the first text file.

Option M - Macro Expansions

When an assembly listing is produced, macro calls are shown in the same form as in the source. If you wish the instructions within macros to be listed, use `M+`, while `M-` will disable the option. You can use this directive as often as required.

Option O - Optimising

GenST is capable of optimising certain statements to faster and smaller versions. By default all optimising is off but each type can be enabled and disabled as required. This option has several forms:

OPT O1+ will optimise backward branches to short if within range, can be disabled with O1-

OPT O2+ will optimise address register indirect with displacement addressing modes to address register indirect, if the displacement evaluates to zero, and can be disabled with O2-. For example

```
move.l next(a0),d3
```

will be optimised to

```
move.l (a0),d3
```

if the value of `next` is zero.

OPT O+ will turn all optimising on

OPT O- will turn all optimising off

OPT O1-, OPT O2-
will disable the relevant optimisation

OPT OW- will disable the warning messages generated by each optimisation, OW+ will enable them.

If any optimising has been done during an assembly the number of optimisations made and bytes saved will be shown at the end of assembly.

Option P - Position Independent checks

With this option enabled with `P+` GenST will check that all code generated is position-independent, generating errors on any lines which require relocation. It can be disabled with `P-` and defaults to off.

Option S - Symbol Table

When a listing is turned on a symbol table will be produced at the end. If you wish to change this, `S-` will disable it, while `S+` will re-enable it. If you use this directive more than once the last one will be taken into account.

Option T - Type Checking

GenST can often spot programming errors as it checks the types of certain expressions. For some applications or styles of programming this can be more of a hindrance than a help so `T-` will turn checks off, `T+` turning them back on. For example the program segment

```
main    bsr     initialise
        lea     main(a6),a0
        move.l  (main).w,a0
```

will normally produce an error as `main` is a relative expression whereas the assembler expects an absolute expression in both cases. However if this code is designed to run on another 68000 machine this may be perfectly valid so the type checking should be disabled.

Option W - Warnings

If you wish to disable the warnings that GenST can produce, you can do so with `W-`. To re-enable them, use `W+`. This directive can be used as often as required.

Option X - Extended Debug

This is a special version of option `D` which uses the HiSoft Extended Debug format to generate debugging information with symbols of up to 22 character significance.

Option Summary

The defaults are shown in brackets after each option description:

C	case-sensitivity & significance (`C127+`)
D	include debugging information (`D-`)
L-	produce executable code (default)
L+	produce GST linkable code
L2	produce DRI linkable code
M	expand macros in listing (`M+`)
O	optimising control (`O-`)
P	position independent code checks (`P-`)
S	symbol table listing
T	type checking (`T+`)
W	warning control (`W+`)
X	Extended debug (`X-`)

For example, the line

```
        opt m+,s+,w-
```

will turn macro expansions on, enable the symbol table list and turn warnings off.

<label>   EVEN

This directive will force the program counter to be even, i.e. word-aligned. As GenST automatically word-aligns all instructions (except DC.Bs and DS.Bs) it should not be required very often, but can be useful for ensuring buffers and strings are word-aligned when required.

CNOP offset,alignment

This directive will align the program counter using the given offset and alignment. An alignment of 2 means word-aligned, an alignment of 4 means long-word-aligned and so on. The alignment is relative to the start of the current section. For example,

```
        cnop 1,4
```

aligns the program counter a byte past the next long-word boundary.

<label> DC.B expression<,expression> ...
<label> DC.W expression<,expression> ...
<label> DC.L expression<,expression> ...

These directives define constants in memory. They may have one or more operands, separated by commas. The constants will be aligned on word boundaries for DC.W and DC.L. No more than 128 bytes can be generated with a single DC directive.

DC.B treats strings slightly differently to those in normal expressions. While the rules described previously about quotation marks still apply, no padding of the bytes will occur and the length of any string can be up to 128 bytes.

Be very careful about spaces in DC directives, as a space is the delimiter before a comment. For example, the line

```
        dc.b      1,2,3 ,4
```

will only generate 3 bytes - the ,4 will be taken as a comment.

<label> DS.B expression
<label> DS.W expression
<label> DS.L expression

These directives will reserve memory locations and the contents will be initialised to zeros. If there is a label then it will be set to the start of the area defined, which will be on a word boundary for DS.W and DS.L directives. There is no restriction on the size, though the larger the area the longer it will take to save to disk.

For example, all of these lines will reserve 8 bytes of space, in different ways:

```
        ds.b 8
        ds.w 4
        ds.l 2
```

<label> DCB.B number,value
<label> DCB.W number,value
<label> DCB.L number,value

This directive allows constant blocks of data to be generated of the size specified. number specifies how many times the value should be repeated.

FAIL

This directive will produce the error `user error`. It can be used for such things as warning the programmer if an incorrect number of parameters have been passed to a macro.

OUTPUT filename

This directive sets the normal output filename though can be overridden by specifying a filename at the start of assembly. If filename starts with a period then it is used as an extension and the output name is built up as described previously.

__G2 (reserved symbol)

This is a reserved symbol that can be used to detect whether GenST 2 is being used to assemble a file using the IFD conditional. The value of this symbol depends on the version of the assembler and is always absolute.

Repeat Loops

It is often useful to be able to repeat one or more instructions a particular number of times and the repeat loop construct allows this.

```
<label>   REPT     expression
          ENDR
```

Lines to be repeated should be enclosed within REPT and ENDR directives and will be repeated the number of times specified in the expression. If the expression is zero or negative then no code will be generated. It is not possible to nest repeat loops. For example

```
        REPT     512/4     copy a sector quickly
        move.l   (a0)+,(a1)+
        ENDR
```

Note Program labels should not be defined within repeat loops to prevent `label defined twice` errors.

Listing Control

LIST

This will turn the assembly listing on during pass 2, to whatever device was selected at the start of the assembly (or to the screen if None was initially chosen). All subsequent lines will be listed until an END directive is reached, the end of the text is reached, or a NOLIST directive is encountered.

Greater control over listing sections of program can be achieved using LIST + or LIST - directives. A counter is maintained, the state of which dictates whether listing is on or off. A LIST + directive adds 1 to the counter and a LIST - subtracts 1. If the counter is zero or positive then listing is on, if it is negative then listing is off. The default starting value is -1 (i.e. listing off) unless a listing is specified when the assembler was invoked, when it is set to 0. This system allows a considerable degree of control over listing particularly for include files. The normal LIST directive sets the counter to 0, NOLIST sets it to -1.

NOLIST

This will turn off any listing during pass 2.

When a listing is requested onto a printer or to disk, the output is formatted into pages, with a header at the top of every page. The header itself consists a line containing the program title, date, time and page number, then a line showing the program title, then a line showing the sub-title, then a blank line. The date format will be printed in the form `DD/MM/YY`, unless the assembler is running on a US Atari ST, in which case the order is automatically changed to `MM/DD/YY`. Between pages a form-feed character (ASCII FF, value 12) is issued.

PLEN expression

This will set the page length of the assembly listing and defaults to 60. The expression must be between 12 and 255.

LLEN expression

This will set the line width of the assembly listing and defaults to 132. The value of the expression must be between 38 and 255.

TTL string

This will set the title printed at the top of each page to the given string, which may be enclosed in single quotes. The first TTL directive will set the title of the first printed page. If no title is specified the current include file name will be used.

SUBTTL string

Sets the sub-title printed at the top of each page to the given string, which may be enclosed in single quotes. The first such directive will set the sub-title of the first printed page.

SPC expression

This will output the number of blank lines given in the expression in the assembly listing, if active.

PAGE

Causes a new page in the listing to be started.

LISTCHAR expression<,expression> ...

This will send the characters specified to the listing device (except the screen) and is intended for doing things such as setting condensed mode on printers. For example, on Epsons and compatibles the line

```
        listchar 15
```

will set the printer to 132-column mode.

FORMAT parameter<,parameter> ...

This allows exact control over the listed format of a line of source code. Each parameter controls a field in the listing and must consist of a digit from 0 to 2 inclusive followed by a + (to enable the field) or a - (to disable it):

0	line number, in decimal
1	section name/number and program counter
2	hex data in words, up to 10 words unless printer is less than 80 characters wide, when up to three words are listed.

Label Directives

label EQU expression

This directive will set the value and type of the given label to the result of the expression. It may not include forward references, or external labels. If there is any error in the expression, the assignment will not be made. The label is compulsory and must not be a local label.

label = expression

Alternate form of EQU statement.

label EQUR register

This directive allows a data or address register to be referred to by a user-name, supplied as the label to this directive. This is known as a *register equate*. A register equate *must* be defined before it is used.

label SET expression

This is similar to EQU, but the assignment is only temporary and can be changed with a subsequent SET directive. Forward references cannot be used in the expression. It is especially useful for counters within macros, for example, using a line like

```
zcount   set      zcount+1
```

(assuming `zcount` is set to 0 at the start of the source). At the start of pass 2 all SET labels are made undefined, so their values will always be the same on both passes.

label REG register-list

This allows a symbol to be used to denote a register list within MOVEM instructions, reducing the likelihood of having the list at the start of a routine different from the list at the end of the routine. A label defined with REG can *only* be used in MOVEM instructions.

<label> RS.B expression
<label> RS.W expression
<label> RS.L expression

These directives let you set up lists of constant labels, which is very useful for data structures and global variables and is best illustrated by a couple of examples.

Let's assume you have a data structure which consists of a long word, a byte and another long word, in that order. To make you code more readable and easier to update should the structure change, you could use lines such as

```
        rsreset
d_next  rs.l    1
d_flag  rs.b    1
d_where rs.l    1
```

then you could access them with lines like

```
        move.l  d_next(a0),a1
        move.l  d_where(a0),a2
        tst.b   d_flag(a0)
```

As another example let's assume you are referencing all your variables off register A6 (as done in GenST and MonST) you could define them with lines such as

```
onstate  rs.b 1
start    rs.l 1
end      rs.l 1
```

You then could reference them with lines such as

```
          move.b   onstate(a6),d1
          move.l   start(a6),d0
          cmp.l    end(a6),d0
```

Each such directive uses its own internal counter, which is reset to 0 at the beginning of each pass. Every time the assembler comes across the directive it sets the label according to the current value (with word alignment if it is .W or .L) then increments it according to the size and magnitude of the directive. If the above definitions were the first RS directives, onstate would be 0, start would be 2 and end would be 6.

RSRESET

This directive will reset the internal counter as used by the RS directive.

RSSET expression

This allows the RS counter to be set to a particular value.

__RS (reserved symbol)

This is a reserved symbol having the current value of the RS counter.

Conditional Assembly

Conditional assembly allows the programmer to write a comprehensive source program that can cover many conditions. Assembly conditionals may be specified through the use of arguments, in the case of macros, and through the definition of symbols in EQU or SET directives. Variations in these can then cause assembly of only those parts necessary for the specified conditions.

There are a wide range of directives concerned with conditional assembly. At the start of the conditional block there must be one of the many IF directives and at the end of each block there must be an ENDC directive. Conditional blocks may be nested up to 65535 levels.

Labels should not be placed on IF or ENDC directives as the directives will be ignored by the assembler.

IFEQ	expression
IFNE	expression
IFGT	expression
IFGE	expression
IFLT	expression
IFLE	expression

These directives will evaluate the expression, compare it with zero and then turn conditional assembly on or off depending on the result. The conditions correspond exactly to the 68000 condition codes. For example, if the label DEBUG had the value 1, then with the following code,

```
        IFEQ DEBUG
logon   dc.b     'Enter a command:',0
        ENDC
        IFNE DEBUG
        opt      d+         labels please
logon   dc.b     'Yeah, gimme man:',0
        ENDC
```

the first conditional would turn assembly off as 1 is not EQ to 0, while the second conditional would turn it on as 1 is NE to 0.

Note IFNE corresponds to IF in assemblers with only one conditional directive.

The expressions used in these conditional statements *must* evaluate correctly.

IFD label
IFND label

These directives allow conditional control depending on whether a label is defined or not. With IFD, assembly is switched on if the label is defined, whereas with IFND assembly is switched on if the label is not defined. These directives should be used with care otherwise different object code could be generated on pass 1 and pass 2 which will produce incorrect code and generate phasing errors. Both directives also work on reserved symbols.

IFC 'string1','string2'

This directive will compare two strings, each of which must be surrounded by single quotes. If they are identical then assembly is switched on, else it is switched off. The comparison is case-sensitive.

IFNC 'string1','string2'

This directive is similar to the above, but only switches assembly on if the strings are not identical. This may at first appear somewhat useless, but when one or both of the parameters are macro parameters it can be very useful, as shown in the next section.

ELSEIF

This directive toggles conditional assembly from on to off, or vice versa.

ENDC

This directive will terminate the current level of conditional assembly. If there are more IFs than ENDCs an error will be reported at the end of the assembly.

IIF expression instruction

This is a short form of the IFNE directive allowing a single instruction or directive to be assembled conditionally. No ENDC should be used with IIF directives.

Macro Operations

GenST fully supports extended Motorola-style macros, which together with conditional assembly allows you greatly to simplify assembly-language programming and the readability of your code.

A macro is a way for a programmer to specify a whole sequence of instructions or directives that are used together very frequently. A macro is first *defined*, then its name can be used in a *macro call* like a directive with up to 36 parameters.

label MACRO

This starts a macro definition and causes GenST to copy all following lines to a macro buffer until an ENDM directive is encountered. Macro definitions may not be nested.

ENDM

This terminates the storing of a macro definition, after a MACRO directive.

MEXIT

This stops prematurely the current macro expansion and is best illustrated by the INC example given later.

NARG (reserved symbol)

This is not a directive but a reserved symbol. Its value is the number of parameters passed to the current macro, or 0 if used when not within any macro. If GenST is in case-sensitive mode then the name should be all upper-case.

Macro Parameters

Once a macro has been defined with the MACRO directive it can be invoked by using its name as a directive, followed by up to 36 parameters. In the macro itself the parameters may be referred to by using the backslash character (\) followed by an alpha-numeric (1-9, A-Z or a-z) which will be replaced with the relevant parameter when expanded or with nothing if no parameter was given. There is also the special macro parameter \0 which is the size appended to the macro call and defaults to W if none is given.If a macro parameter is to include spaces or commas then the parameter should be enclosed in between < and > symbols; in this case a > symbol may be included within the parameter by specifying >>.

A special form of macro expansion allows the conversion of a symbol to a decimal or hexadecimal sequence of digits, using the syntax \<symbol> or \$<symbol>, the latter denoting hex expansion. The symbol must be defined and absolute at the time of the expansion.

The parameter \@ can be useful for generating unique labels with each macro call and is replaced when the macro is expanded by the sequence _nnn where nnn is a number which increases by one with every macro call. It may be expanded up to five digits for very large assemblies.

A true \ may be included in a macro definition by specifying \\.

A macro call may be spread over more than one line, particularly useful for macros with large numbers of parameters. This can be done by ending a macro call with a comma then starting the next line with an & followed by tabs or spaces then the continuation of the parameters.

In the assembly listing the default is to show just the macro call and not the code produced by it. However, macro expansion listings can be switched on and off using the OPT M directive described previously.

Macro calls may be nested as deeply as memory permits, allowing recursion if required.

Macro names are stored in a separate symbol table to normal symbols so will not clash with similarly-named routines, and may start with a period.

Example 1 - Calling the BDOS

As the first example, the general GEMDOS calling-sequence for the BDOS is:

put a word parameter on the stack
invoke a TRAP #1
correct the stack afterwards

A macro to follow these specifications could be

```
call_gemdos        MACRO
          move.w   #\1,-(a7)          function
          trap     #1
          lea      \2(a7),a7          correct stack
          ENDM
```

The directives are in capitals only to make them stand out: they don't have to be. If you wanted to call this macro to use GEMDOS function 2 (print a character) the code would be

```
          move.w   #'X',-(a7)
          call_gemdos 2,4
```

When this macro call is expanded, `\1` is replaced with `2` and `\2` is replaced with `4`. `\0`, if it occurred in the macro, would be `W` as no size is given on the call. So the above call would be assembled as:

```
          move.w   #2,-(a7)
          trap     #1
          lea      4(a7),a7
```

Example 2 - an INC instruction

The 68000 does not have the INC instruction of other processors, but the same effect can be achieved using an ADDQ #1 instruction. A macro may be used to do this, like so:

```
inc       MACRO
          IFC      '','\1'
          fail     missing parameter!
          MEXIT
          ENDC
          addq.\0  #1,\1
          ENDM
```

An example call would be

```
        inc.l     a0
```

which would expand to

```
        addq.l   #1,a0
```

The macro starts by comparing the first parameter with an empty string and causing an error message to be issued using `FAIL` if it is equal. The `MEXIT` directive is used to leave the macro without expanding the rest of it. Assuming there is a non-null parameter, the next line does the `ADDQ` instruction, using the `\0` parameter to get the correct size.

Example 3 - A Factorial Macro

Although unlikely actually to be used as it stands, this macro defines a label to be the factorial of a number. It shows how recursion can work in macros. Before showing the macro, it is useful to examine how the same thing would be done in a high-level language such as Pascal.

```
function factor(n:integer):integer;
begin
        if n>0 then
                factor:=n*factor(n-1)
        else
                factor:=1
end;
```

The macro definition for this uses the SET directive to do the multiplication n*(n-1)*(n-2) etc. in this way:

```
* parameter 1=label, parameter 2='n'
factor  MACRO
        IFND    \1
\1      set     1               set if not yet defined
        ENDC
        IFGT    \2
        factor  \1,\2-1         work out next level down
\1      set     \1*(\2)         n=n*factor(n-1)
        ENDC
        ENDM
* a sample call
        factor  test,3
```

The net result of the previous code is to set `test` to 3! (3 factorial). The reason the second `SET` has `(\2)` instead of just `\2` is that the parameter will not normally be just a simple expression, but a list of numbers separated by minus signs, so it could assemble to

```
test      set       test*5-1-1-1
```

(i.e. `test*5-3`) instead of the correct

```
test      set       test*(5-1-1-1)
```

(i.e. `test*2`).

Example 4 - Conditional Return Instruction

The 68000 lacks the conditional return instructions found on other processors, but macros can be defined to implement them using the `\@` parameter. For example, a return if EQ macro could look like:

```
rtseq     MACRO
          bne.s     \@
          rts
\@
          ENDM
```

The `\@` parameter has been used to generate a unique label every time the macro is called, so will generate in this case labels such as `_002` and `_017`.

Example 5 - Numeric Substitution

Suppose you have a constant containing the version number of your program and wish this to appear as ASCII in a message:

```
showname MACRO
          dc.b      \1,'\<version>',0
          ENDM
          .
          .
version   equ       42
          showname <'Real Ale Search Program v'>
```

will expand to the line

```
          dc.b      'Real Ale Search Program v','42',0
```

Note the way the string parameter is enclosed in <>s as it contains spaces.

Example 6 - Complex Macro Call

Suppose you program needs a complicated table structure which can have a varying number of fields. A macro can be written to only use those parameters that are specified, for example:

```
table_entry       macro
        dc.b      .end\@-*          length byte
        dc.b      \1                always
        IFNC      '\2',''
        dc.w      \2,\3             2nd and 3rd together
        ENDC
        dc.l      \4,\5,\6,\7
        IFNC      '\8',''
        dc.b      '\8'              text
        ENDC
        dc.b      \9
.end\@  dc.b      0
        ENDM

* sample call
        table_entry     $42,,,t1,t2,t3,t4,
&                       <Enter name:>,%0110
```

This is a non-trivial example of how macros can make a programmer's life so much easier when dealing with complex data structures. In this case the table consists of a length byte, calculated in the macro using \@, two optional words, four longs, an optional string, a byte, then a zero byte. The code produced in this example would be

```
        dc.b      .end_001
        dc.b      $42
        dc.l      t1,t2,t3,t4
        dc.b      'Enter name:'
        dc.b      %0100
.end_001 dc.b     0
```

Output File Formats

GenST is very flexible in terms of output file formats. These are detailed in this section together with notes on the advantages and disadvantages of each. Certain directives take different actions, depending on what output file format is specified.

The exact details of using each format will now be described.

Executable Files

These are directly executable, for example by double-clicking from the Desktop. The file may include relocation information and/or symbolic information. Normal file extensions for this type file are `.PRG`, `.TOS`, `.TTP` and `.ACC`.

Advantages true BSS sections, reduced development time.

Disadvantages messy if more than one programmer.

GST Linkable Files

When writing larger programs, or when writing assembly language modules for use from the high-level language, a programmer needs to generate a linkable file. The GST linker format is supported by the majority of high-level languages produced in England as well as others, for example **HiSoft BASIC**, **Lattice C**, **Prospero FORTRAN** and **Prospero Pascal**. GST format files normally have the extension of `.BIN`.

Advantages great degree of freedom - imported labels can be used practically anywhere including within arbitrary expressions, libraries can be created directly from the assembler, import method means assembler can detect type conflicts.

Disadvantages library format means selective library linking can be slow, true GEMDOS sections not supported as standard (though LinkST can create true BSS sections).

This is the original linker format for the Atari ST created by Digital Research originally for CP/M 68K. It is supported, often via a conversion utility, by the majority of US high-level languages. DRI format files normally have the extension of `.o`.

Advantages selective libraries are faster to link than GST format, GEMDOS sections fully supported.

Disadvantages very restrictive on use of imported labels; object files twice as big as executable files, 8 character limit on symbols.

Choosing the Right File Format

If you wish to link with a high-level language there isn't usually much choice - you have to use whichever format is supported by the language.

If you are writing entirely in assembly language then the normal choice has to be *executable* - it is fast to assemble, no linking required, and allows assemble to memory for decreased development time.

If you are writing a larger program, say bigger than 32k object, or writing a program as a team, then *linkable* code often makes most sense. We recommend GST-linkable over DRI because of the much greater flexibility in the format.

Output File Directives

This section details those directives whose actions depend on the output file format chosen. The file format itself can be chosen by one of the following methods: command line options using `GENST2.TTP`; clicking on the radio buttons in the Assembly Options dialog box from the editor; or with the `OPT L` directive at the beginning of the source file.

Icons are used to denote those sections specific to a file format, viz:

Exec Executable-code, also assembled-to-memory code

GST GST-linkable code

DRI DRI-linkable code

Modules & Sections

MODULE modulename

This defines the start of a new module. The module name should be contained within quotes if spaces are included in it. There is a default module called ANON_MODULE so the directive is not obligatory.

Exec This directive is ignored.

DRI This directive is ignored.

GST This directive allows assembly-language library files to be created using multiple modules. Each module is like a self-contained program segment, with its own imports and exports. Relative labels are local to their own module, so you can use two labels with the same name in different modules with no danger of a clash. Absolute labels are global to all modules, ideal for constants and the like.

SECTION sectionname

This defines a switch to the named section. A program may consist of several sections which will be concatenated together with other sections of the same name in the final executable file. By default assembly starts in the TEXT section. You may switch to any section at any time during an assembly.

Exec

Allowed section names are TEXT, the normal program area, DATA, for initialised data, and BSS, a special area of memory reserved by the GEMDOS program loaded. It is initialised to zeroes and takes up no space within the disk file. When in a BSS section no code-generating instructions are allowed except the DS directive. Using a BSS section for global variables can save valuable disk space.

DRI

The rules described above for executable files apply.

GST

There are no rigid rules about section names. Sections with the same name from different files will be concatenated by the linker. The default ordering of sections is the order they are first used in.

Imports & Exports

With both linkable types of program it is crucial to be able to import and export symbols, both relative symbols (i.e. program references) and absolute symbols (i.e. constants). The GST format distinguishes between these types whereas the DRI format does not. The GST format allows the assembler to type check, often finding programming errors that would otherwise be missed.

XDEF export<,export>...

This defines labels for export to other programs (or modules). If any of the labels specified are not defined an error will occur. It is not possible to export local labels.

Exec This directive is ignored.

DRI Note that all symbols will be truncated (without warning) before exporting. `OPT C8` is therefore recommended.

XREF import<,import)>...
XREF.L import<,import>...

This defines labels to be imported from other programs or modules. If any of the labels specified are defined an error will occur. The normal XREF statement should be used to import a relative label (i.e. program reference), while XREF.L should be used to import absolute labels (i.e. constants). Importing a label more than once will not produce an error.

Exec This directive is ignored.

DRI The DRI format does not actually *need* to know the type of imports but it is recommended that both forms of XREF are used to allow the assembler to type check. If you do not type your imports you should turn type-checking off using `OPT T-`. DRI labels are only significant to the first 8 characters.

GST Care should be taken to import labels of the correct type otherwise the relocation information will not be correct.

Exec There are no imports!

DRI Imports may be used in expressions but only one import per expression is allowed. The result of an expression with an import in must be of the form import+number or import-number. Imports can be combined with arbitrarily complex expressions, so long as the complex expression lexically precedes it, for example

```
        move.l    3+(1<<count+5)+import
```

GST Imports may be used in expressions, with up to ten per expression. They may only be added or subtracted from each other though can be combined with arbitrarily complex expressions, so long as the complex expression lexically precedes it, for example:

```
        move.l    3+(1<<count+5)+import1-import2
```

Where exactly an expression involving an import can be used depends on the file format. The following table shows which are allowed.

Expression	GST	DRI	Example
PC-byte	Y	N	`move.w import(pc,d3.w)`
			`bsr.s  import`
PC-word	Y	Y†	`move.w import(pc),a0`
			`bsr    import`
byte	Y	N	`move.b #import,d0`
word	Y	Y	`move.w import(a3),d0`
long	Y	Y	`move.l import,d0`

†so long as it is not a reference to a different section in the same program, which is not allowed.

DRI **GST** Note that a reference to a symbol in a different section is regarded as an import and subject to the above rules.

COMMENT commentstring

Exec This directive is ignored.

DRI This directive is ignored.

GST This directive passes the following string, exactly as entered, into the `.BIN` file and will be shown by the linker.

ORG expression

This will make the assembler generate position-dependent code and set the program counter to the given value. Normal GEMDOS programs do not need an ORG statement even if position-dependent. It is included to allow code to be generated for the ROM port or for other 68000 machines. More than one ORG statement is allowed in a source file but no padding of the file is done.

Exec It should be used with great care as the binary file generated will probably not execute correctly when double-clicked, as no relocation information is written out. The binary file produced has the standard GEMDOS header at the front, but no relocation information.

DRI This directive is not allowed, absolute code generation is an option in the linker.

GST This sends the ORG directive to the linker which will pad the file with zeroes to the given address.

Note This directive is very unlikely to make sense when assembling to memory.

OFFSET <expression>

This switches code generation to a special section to generate absolute labels. The optional expression sets the program counter for the start of this section. No bytes are written to the disk and the only code-generating directive allowed is DS. Labels defined in this section will be absolute. For example to define some of the system variables of the ST:

```
            OFFSET    $400
etv_timer             ds.l      1         will be $400
etv_critic            ds.l      1         404
etv_term              ds.l      1         408
ext_extra             ds.l      5         40C
memvalid              ds.l      1         420
memcntlr              ds.w      1         424
```

__LK (reserved symbol)

This is a reserved symbol that can be used to detect which output mode is specified The value of this symbol is always absolute and one of the following:

0	executable
1	GST linkable
2	DRI linkable

Other values are reserved for future expansion.

DRI Debug Option

Normally only explicitly XDEFed labels are included in the symbol table within the output file. However the format allows what it calls *local labels* (not to be confused with GenST local labels) which are not true exports and cannot be referred to in other modules but will be included in the symbol table in the final output file for debugging purposes. `OPT D+` will cause all relative labels to be output as DRI local labels.

Writing GST Libraries

When using multiple MODULEs to generate a GST format library file care must be taken with backward references to imports. Within a library file, higher level routines should be first, lower level routines last. For example the source file skeleton overleaf not link when used as a selective library.

```
        MODULE   low_level
        XDEF     low_output
low_output
        etc
        MODULE   high_level
        XDEF     high_output
        XREF     low_output
high_output
        etc
```

This is because the second module references a label defined in an earlier module, which is not allowed. The corrected version is:

```
        MODULE   high_level
        XDEF     high_output
        XREF     low_output
high_output
        etc
        MODULE   low_level
        XDEF     low_output
low_output
        etc
```

Simple File Format Examples

This section shows a (non-functional and incomplete) example of the use of each file format.

Executable

```
        SECTION  TEXT
start   lea      string(pc),a0
        move.l   a0,save_str
        bsr      printstring
        bra      quit
        SECTION  DATA
string  dc.b     'Enter your name,0
        SECTION  TEXT
printstring
        move.l   a0,-(sp)
        move.w   #9,-(sp)
        trap     #1
        addq.l   #6,sp
        rts
        SECTION  BSS
save_str ds.l    1
        END
```

DRI Linkable

```
          XREF.L   quit
          SECTION  TEXT
start     move.l   #string,a0
          move.l   a0,save_str
          bsr      printstring
          bra      quit
          SECTION  DATA
string    dc.b     'Enter your name,0
          SECTION  TEXT
printstring
          move.l   a0,-(sp)
          move.w   #9,-(sp)
          trap     #1
          addq.l   #6,sp
          rts
          SECTION  BSS
save_str ds.l      1
          END
```

Note the way the first instruction has been changed as a PC-relative reference is not allowed between sections.

GST Linkable

```
          MODULE   TESTPROG
          COMMENT  needs work
          XREF.L   quit
          SECTION  TEXT
start     lea      string(pc),a0
          move.l   a0,save_str
          bsr      printstring
          bra      quit
          SECTION  DATA
string    dc.b     'Enter your name,0
          SECTION  TEXT
printstring
          move.l   a0,-(sp)
          move.w   #9,-(sp)
          trap     #1
          addq.l   #6,sp
          rts
          SECTION  BSS
save_str ds.l      1
          END
```

Directive Summary

Assembly Control

END	terminate source code
INCLUDE	read source file from disk
INCBIN	read binary file from disk
OPT	option control
EVEN	ensure PC even
CNOP	align PC arbitrarily
DC	define constant
DS	define space
DCB	define constant block
FAIL	force assembly error

Repeat Loops

REPT	start repeat block
ENDR	end repeat block

Listing Control

LIST	enable listing
NOLIST	disable listing
PLEN	set page length
LLEN	set line length
TTL	set title
SUBTTL	set sub-title
PAGE	start new page
LISTCHAR	send control character
FORMAT	define listing format

Label Directives

EQU	define label value
EQUR	define register equate
SET	define label value temporarily
REG	define register list
RS	reserve space
RSRESET	reset RS counter
RSSET	set RS counter

Conditional Assembly

IFEQ	assemble if zero
IFNE	assemble if non-zero
IFGT	assemble if greater than
IFGE	assemble if greater than or equal to
IFLT	assemble if less than
IFLE	assemble if less than or equal to
IFD	assemble if label defined
IFND	assemble if label not defined
IFC	assemble if strings same
IFNC	assemble if strings different
ELSEIF	switch assembly state
ENDC	end conditional
IIFC	immediate IF

Macros

MACRO	define macro
ENDM	end macro definition

Output File Directives

MODULE	start new module
SECTION	switch section
XDEF	define label for export
XREF	define label for import
COMMENT	send linker comment
ORG	set absolute code generation
OFFSET	define offset table

Reserved Symbols

NARG	number of macro parameters
__G2	internal version number
__RS	RS counter
__LK	output file type

CHAPTER 4
Symbolic Debugger

Introduction

Programs written in assembly language are particularly error-prone because even a slight mistake can result in the entire machine crashing. There are various forms of bugs, ranging from the trivial (e.g. a missing CR in a printout), through the usual (e.g. an incorrect result) to the very serious (e.g. the machine completely hanging, perhaps with a weird display).

To help you find and correct all forms of bugs, DevpacST includes MonST. MonST is a symbolic debugger and disassembler which lets you examine programs and memory, execute programs an instruction at a time and trap processor exceptions caused by programmer error. As MonST is symbolic you can look at your program complete with all the original labels, making debugging very much easier than having to battle with 6-digit hex numbers.

Although MonST is a *low-level* debugger, displaying such things as 68000 instructions and bytes of memory, it can also be used for debugging programs written with any compiler that generates machine-code output. If the compiler has the option to dump the symbols into the binary code then you will see your procedure and function names within the code, and you can even view your original source code. We ourselves used MonST when debugging LinkST, which was written using a C compiler. MonST and GenST themselves were written entirely in assembly language.

As MonST uses its own screen memory, the display of your program is not destroyed when you single-step or breakpoint, making it particularly useful for graphical-output programs such as GEM applications or games. It also uses its own screen drivers so it is possible to single-step into the operating system screen routines such as the AES or BIOS without affecting the debugger.

There are three versions of MonST supplied on the disk. All are similar to use and are provided to make the debugging of different types of programs easy. The exact differences are detailed later.

68000 Exceptions

MonST uses the 68000 processor exceptions to stop runaway programs and to single-step, so at this point it would be useful to explain them and what normally happens when they occur on an ST.

There are various types of exception that can occur, some deliberately, others accidentally. When one does occur the processor saves some information on the SSP, goes into Supervisor mode and jumps to an exception handler. When MonST is active it re-directs some of these exceptions so it can take control when they occur. The various forms of exceptions, their usual results, and what happens when they occur with MonST active is shown in the following table:

Exception no.	Exception	Usual effect	MonST active
2	bus error	bombs	trapped
3	address error	bombs	trapped
4	illegal instruction	bombs	trapped
5	zero divide	bombs	trapped
6	CHK instruction	bombs	trapped
7	TRAPV instruction	bombs	trapped
8	privilege violation	bombs	trapped
9	trace	bombs	used for single-stepping
10	line 1010 emulator	fast VDI interface	fast VDI interface
11	line 1111 emulator	internal TOS	internal TOS
32	trap #0	bombs	trapped
33	trap #1	GEMDOS call	GEMDOS call
34	trap #2	AES/VDI call	AES/VDI call
35-44	trap #3-#12	bombs	trapped
45	trap #13	XBIOS call	XBIOS call
46	trap #14	BIOS call	BIOS call
47	trap #15	bombs	trapped

The exact causes of the above exceptions (and how best to recover from them) are detailed at the end of this section, but to summarise:

Exceptions 2 to 8 are caused by a programmer error and are trapped by MonST.

Exception 9 can remotely be caused by programmer error and is used by MonST for single stepping.

Exceptions 10, 11, 33, 34, 45 and 46 are used by the system and left alone.

The rest (i.e. the unused Trap exceptions) are diverted into MonST, but can subsequently be re-defined to be exploited by programs if required.

The 'bombs' entry in the table above means that the ST will attempt to recover from the exception, but it is not always successful.

When an exception occurs, the ST prints on the screen a number of *bomb* shapes (or *mushrooms* on disk-loaded GEMDOS), the number being equal to the exception number. Having done this, it will abort the current program (losing any unsaved data from it) and attempt a return to the Desktop.

If the exception was caused by or resulted in important system variables being destroyed then the attempt may fail and the machine will not recover.

Occasionally very nasty crashes can cause the whole screen to fill with bombs (or mushrooms) which looks very impressive, but is not very useful!

Memory Layout

The usual versions of MonST co-reside with programs being debugged; that is, they are loaded, ask for a filename, and load that file in together with any labels.

It is useful to examine the usual logical memory map (the physical layout is shown in **Appendix C**) both with and without MonST, shown in **Figure 4.1** on the next page.

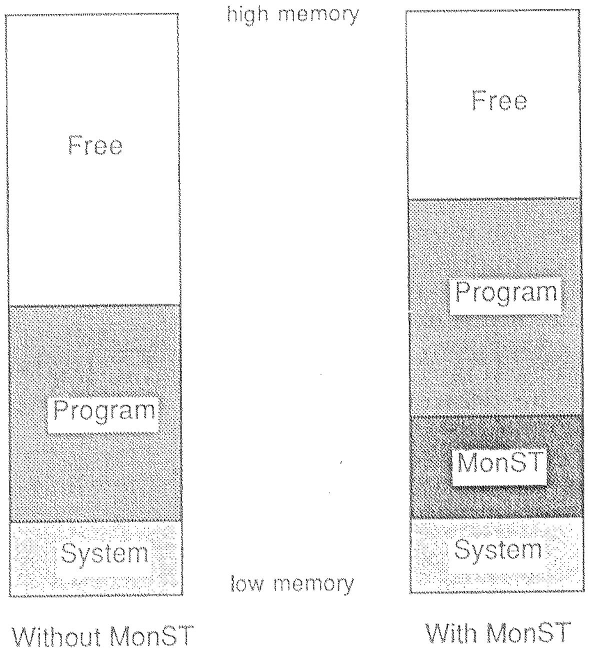


Figure 4.1 - Logical Memory Map

The actual code size of MonST is around 23k, but in addition it requires an additional 32k of workspace. This may seem large but it is required for the copy of the ST screen memory saved by MonST; this is a most useful feature of the debugger.

The three versions of MonST supplied are:

`MONST2.PRG`	GEM interactive version
`MONST2.TOS`	TOS interactive version
`AMONST2.PRG`	Auto-resident version

For now the first two will be described; the auto-resident version is described later but is very similar in use to the others.

Invoking MonST

From the Desktop

The two interactive versions of MonST are actually identical except for the filename extension. The GEM version should be used for GEM-based programs, which require use of the mouse and have initially a grey-pattern screen, while the TOS version should be used for TOS-based programs which require the flashing TOS cursor and have initially a white screen display. Both versions are invoked by double-clicking on their respective icons from the Desktop.

Note If you debug a TOS program with the GEM version of the debugger it will work fine but the screen display will be probably be messy; however, debugging a GEM program with a TOS debugger will cause all sorts of nasty problems to occur and should be avoided.

From the Editor

When GenST is invoked it automatically looks for and loads the file `MONST2.PRG` into memory (unless this option is disabled in the Preferences option in the editor). The debugger is then instantly available at the press of a key from within the editor.

Pressing `Alt-M` or clicking on MonST from the Program menu will then invoke it in a similar way to that described above for the disk-based version only very much more quickly.

Pressing `Alt-D` or clicking on Debug from the Program menu will invoke MonST but will also automatically prepare a program previously assembled to memory to be run, including any symbols within it.

The type of initial screen mode used when invoked from the editor is determined by the Run with GEM menu item on the Program menu - if a check mark is present then GEM screen initialisation is done, otherwise TOS screen initialisation is used. The rules described above about using the wrong type of screen initialisation are also relevant to the in-memory debugger.

Symbolic Debugging

A major feature of MonST is its ability to use symbols taken from the original program whilst debugging. MonST supports two formats for debug information - the DRI standard, which allows up to 8 characters per symbol, and the *HiSoft Extended Debug* format, allowing up to 22 characters. Both GenST and LinkST can produce both formats, and many other vendors' compilers and linkers have an option to produce DRI-format debugging information. We are trying to establish the HiSoft Extended format as a second standard on the ST, but at the time of writing the only other products to support the format are **HiSoft BASIC** and **FTL-Modula 2**.

MonST Dialog and Alert Boxes

MonST makes extensive use of dialog- and alert-boxes which are similar in concept to those in GEM programs but have several differences. MonST does not use genuine GEM-type boxes in order for it to remain *robust* - that is to avoid interaction when debugging programs that themselves use GEM calls. In addition the mouse is not available within the debugger itself which makes things like true GEM buttons impossible.

A MonST dialog box displays the prompt `ESC to abort` above the top left corner of the box together with a prompt, normally followed by a blank line with a cursor. At any time a dialog box may be aborted by pressing `Esc`, or data may be entered by typing. The cursor keys, `Backspace` and `Del` keys may be used to edit entered text in the usual way and the whole line may be deleted by pressing the `Clr` key - note that this is different to GEM dialog boxes which use the `Esc` key to delete a whole line of text. An entered line is terminated by pressing the `Return` key, though if the line contains errors the screen will flash and the `Return` key will be ignored allowing correction of the data before pressing `Return` again. Another difference is that dialog boxes that require more than one line of data to be entered do not allow the use of the cursor up and down keys to switch between different lines - in MonST the lines have to be entered in order.

A MonST alert box is a small box displaying a message together with the prompt `[Return]` and is normally used to inform the user of some form of error. The box will disappear on pressing the `Return` or `Esc` keys, whichever is more convenient.

Initial Display

Unless you have chosen the Debug option within the editor you will be presented with a dialog box prompting for an executable program name. If you wish to debug a program from disk you should enter the filename (which defaults to an extension of `.PRG`) then press `Return`, then you will be prompted for any command line. If you do not wish to debug a program from disk at this stage, for example you wish to investigate memory, press the `Esc` key or enter a blank filename.

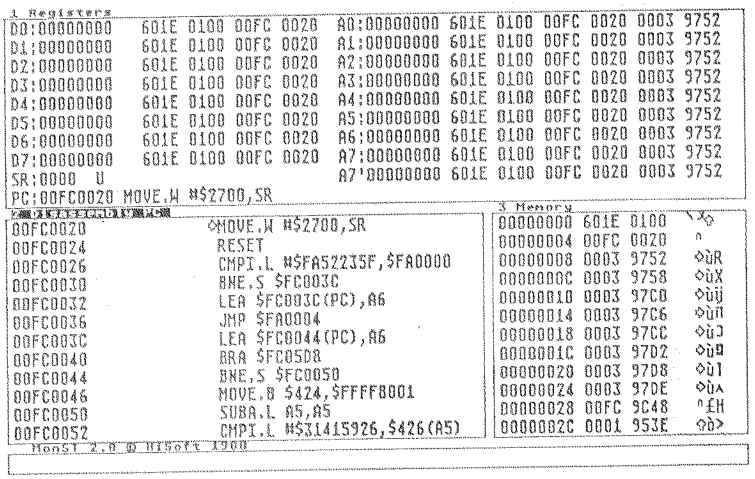
Low Res Certain features work differently or are not available when using MonST in low resolution. They are shown with this icon.

Front Panel Display

The main display of MonST is via a *Front Panel* showing registers, memory and instructions. The name Front Panel stems from the type of panels that were mounted on mainframe and mini computers to provide information on the state of the machine at a particular moment, usually through the use of flashing lights. These lights represent whether or not particular flip-flops (electronic switches) within the computer are open or closed; the flip-flops that are chosen to be shown on this panel are normally those that make up the internal registers and flags of the computer thus enabling programmers and engineers to observe what the computer is doing when running a program.

So these are *hardware* front panel displays; what MonST provides you with is a *software* front panel - the code within MonST works out the state of your computer and then displays this information on the screen.

The initial MonST display consists of four windows, similar to those shown in **Figure 4.1**. In low-resolution the arrangement of two of the windows is slightly different to allow efficient use of the smaller available screen space.



```
1 Registers
D0:00000000   601E 0100 00FC 0020  A0:00000000 601E 0100 00FC 0020 0003 9752
D1:00000000   601E 0100 00FC 0020  A1:00000000 601E 0100 00FC 0020 0003 9752
D2:00000000   601E 0100 00FC 0020  A2:00000000 601E 0100 00FC 0020 0003 9752
D3:00000000   601E 0100 00FC 0020  A3:00000000 601E 0100 00FC 0020 0003 9752
D4:00000000   601E 0100 00FC 0020  A4:00000000 601E 0100 00FC 0020 0003 9752
D5:00000000   601E 0100 00FC 0020  A5:00000000 601E 0100 00FC 0020 0003 9752
D6:00000000   601E 0100 00FC 0020  A6:00000000 601E 0100 00FC 0020 0003 9752
D7:00000000   601E 0100 00FC 0020  A7:00000000 601E 0100 00FC 0020 0003 9752
SR:0000  U                         A7'00000000 601E 0100 00FC 0020 0003 9752
PC:00FC0020 MOVE.W #$2700,SR
2 Disassembly PC                                    3 Memory
00FC0020            ⇒MOVE.W #$2700,SR               00000000 601E 0100   `^
00FC0024             RESET                          00000004 00FC 0020   ^
00FC0026             CMPI.L #$FA52235F,$FA0000      00000008 0003 9752   ùR
00FC0030             BNE.S $FC003C                  0000000C 0003 9758   ùX
00FC0032             LEA $FC003C(PC),A6             00000010 0003 97C0   ùÿ
00FC0036             JMP $FA0004                    00000014 0003 97C6   ùn
00FC003C             LEA $FC0044(PC),A6             00000018 0003 97CC   ùJ
00FC0040             BRA $FC05D8                    0000001C 0003 97D2   ù
00FC0044             BNE.S $FC0050                  00000020 0003 97D8   ùl
00FC0046             MOVE.B $424,$FFFF8001          00000024 0003 97DE   ùA
00FC0050             SUBA.L A5,A5                   00000028 00FC 9C48   ^£H
00FC0052             CMPI.L #$31415926,$426(A5)     0000002C 0001 953E   ò>
  MonST 2.0 © HiSoft 1988
```

Figure 4.1 MonST Initial Display

The top window (number 1) displays the values of the data and address registers, together with the memory pointed to by these registers.

The next window (number 2) is the disassembly window, this displays several lines of instructions, by default based around the program counter (PC), shown in the title area of the window. A ⇒ sign is used to denote the current value of the PC.

Window number 3 is the memory window which displays a section of memory in word-aligned hex and ASCII.

The final window at the bottom of the screen, which is unnumbered, is the smallest window and is used to display messages.

One of the most powerful features of MonST is its flexibility with windows - up to 2 additional windows may be created, the font size can be changed, and windows may be locked to particular registers, these features are detailed later.

Simple Window Handling

MonST has the concept of a *current window* - this is denoted by displaying its title in black. The current window may be changed by pressing the `Tab` key to cycle between them, or by pressing the `Alt` key together with the window number, for example `Alt-2` selects the disassembly window. (AZERTY keyboard users please note - the `Shift` key is *not* required when using `Alt` to select windows). Note that the lowest window can never be made the current window - it is used solely for displaying messages.

Command Input

MonST is controlled by single-key commands which creates a very fast user-interface, though this can take getting used to if you are familiar with a line-oriented command interface of another debugger. Users of **HiSoft Devpac** on other machines will find many commands are identical, particularly with the Spectrum and QL debuggers, though the window commands are unique to MonST.

In general the `Alt` key is the window key - when used in conjunction with other keys it acts on the *current window*.

Commands may be entered in either upper or lower case. Those commands whose effects are potentially disastrous require the `Ctrl` key to be pressed in addition to a command key. The keys used were chosen to be easy to remember, wherever possible. Commands take effect immediately - there is no need to press `Return` and invalid commands are simply ignored. The relevant sections of the front panel display are updated after each command so any effects can be seen immediately.

MonST is a powerful and sometimes complex program and we realise that it is unlikely that many users will use every single command. For this reason the remainder of the MonST manual is divided into two sections - the former is an introduction to the basic commands of the program, while the latter is a full reference section. It is possible for new users and beginners to use the debugger effectively while having only read the Overview; don't be intimidated by the Reference section.

MonST Overview

To start with you will need to **load a program** to debug: if you have assembled a program to memory you can use the Debug option from the editor, else you will need to load a program from disk. When initially loaded you will be prompted for a filename, if you got an error or didn't specify a filename you can have another go by pressing `Ctrl-L`.

A program's symbols will be used by the debugger, if found. A program will have symbols included if you used the Debug or Extended Debug options of the assembler. The extended debug option means you will get longer symbols, the normal option forces them to be truncated to 8 characters.

The most common command in MonST is probably **single-step**, obtained by pressing `Ctrl-Z` (or `Ctrl-Y` if you find it more convenient). This will execute the instruction at the PC, the one shown in the Register window and, normally, also in the Disassembly window. After executing it the debugger re-displays the values of the registers and memory displayed, so you can watch the processor execute your program, step by step. Single-stepping is the best way of going through sections of code that are suspect and require deeper investigation, but it is also the slowest - you may only be interested in a section of code near the end of your program which could take ages to get to if you have to single-step all the way. There is, of course, an answer.

A **breakpoint** is a special word placed into your program to stop it running and enter MonST. There are many types of breakpoint but we will restrict ourselves to the simplest for now. A breakpoint may be set by pressing `Alt-B`, then entering the address you wish to place the breakpoint. You can enter addresses in MonST in hex (the default base), as a symbol, or as a complex expression. Examples of valid addresses are `1A2B0`, `prog_start`, `10+mydata`. If you type in an invalid address the screen will flash and allow you to correct the expression.

Having set a breakpoint you need some way of letting your program actually **run**, and `Ctrl-R` will do this. If will execute your program using the registers displayed and starting from the PC. MonST will be re-entered if a breakpoint has been hit, or if an exception occurs.

MonST uses its own **screen display** which is independent from your programs. If you press the `V` key you will see your current programs display, pressing another key switches you back to MonST. This allows you to debug programs without disturbing their output at all.

MonST uses its own windows to, and any window may be **zoomed** to the full screen size by pressing `Alt-Z`. To return to the main display press `Alt-Z` or the `Esc` key. The `Esc` key is also the best way of getting out of anything you may have invoked by accident. The Zoom command, like all `Alt`-commands, works on the *current window* which you can change by pressing `Tab`. You can dump the current window to your **printer** by pressing `Alt-P`.

To change the **address** from which a window displays it data, press `Alt-A`, then enter the new address. Note that the disassembly window will always re-display from the PC after you single-step, because it is *locked* to the PC. The locking of windows is detailed in the **Reference** section.

To **quit** MonST press `Ctrl-C`. Strange as it may sound this will not always work - what `Ctrl-C` does is terminate the *current program*, which may be MonST or, more likely, the program you are debugging. You know when you have terminated the program under investigation because it will say so in the lower window. Once your program has been terminated, pressing `Ctrl-C` will terminate MonST. If you used the Debug option from the editor then `Ctrl-C` will always terminate MonST as well as your program.

We hope this overview has given you a good idea of the most common features of MonST to let you get on with the complex process of writing and debugging assembly language programs. When you feel more confident you should try and read the **Reference** section, probably best taken, like all medicine, in small doses.

MonST Reference

Numeric Expressions

MonST has a full expression evaluator, based on that in GenST, including operator precedence. The main differences are that the default base is hexadecimal (decimal may be denoted with a \ sign), there is no concept of *types* of expressions (relative or absolute), * is used *only* for multiplication and there is a not-equals operator, <>.

Symbols may be referred to and are normally case-sensitive and significant to either 8 or 22 characters (depending on the form of debug used), though this can be changed with Preferences.

Registers may be referred to simply by name, such as A3 or D7 (case insensitive), but this clashes with hex numbers. To obtain such hex numbers precede them with either a leading zero or a $ sign. A7 refers to the *user* stack pointer.

There are several reserved symbols which are case insensitive, namely TEXT, DATA, BSS, END, SP, SR, and SSP. END refers to one byte past the end of the BSS section and SP refers to either the user- or supervisor-stack, depending on the current value of the status register.

In addition there are 10 memories numbered M0 through M9, which are treated in a similar way to registers and can be assigned to using the Register Set command. Memories 2 through 5 inclusive refer to the current start address of the relevant window and assigning to them will change the start address of that window.

The MonST expression evaluator also supports indirection using the { and } symbols. Indirection may be performed on a byte, word or long basis, by following the } with a period then the required size, which defaults to long. If the pointer is invalid, either because the memory is unreadable or even (if word or longword indirection is used) then the expression will not be valid.

For example, the expression

```
        {data_start+10}.w
```

will return the word contents of location `data_start+10`, assuming `data_start` is even. Indirection may be nested in a similar way to ordinary parenthesis.

Window Types

There are four window types and the exact contents of these windows and how they are displayed is detailed below. The allowed types of windows is shown in the table below.

Window	**Allowed Types**
1	Register
2	Disassembly
3	Memory
4	Disassembly, Memory or Source-code
5	Memory

Register Window Display

The data registers are shown in hex, together with the ASCII display of their low byte and then a hex display of the eight bytes they point to in memory. The address registers are also shown in hex, together with a hex display of 12 bytes. As with all hex displays in MonST this is word-aligned, with non-readable memory displayed as `**`.

The status register is shown in hex and in flag form, additionally with `U` or `S` denoting user- or supervisor-modes. `A7'` denotes the supervisor stack pointer, displayed in a similar way to the other address registers.

The PC value is shown together with a disassembly of the current instruction. Where this involves one or more effective addresses these are shown in hex, together with a suitably-sized display of the memory they point to.

For example, the display

```
        TST.W $12A(A3) ;00001FAE 0F01
```

signifies that the value of $12A plus register A3 is $1FAE, and that the word memory pointed to by this is $0F01. A more complex example is the display

```
  MOVE.W $12A(A3),-(SP) ;00001FAE 0F01 ⇒0002AC08 FFFF
```

The source addressing mode is as before but the destination address is $2AC08, presently containing $FFFF. Note that this display is always of a suitable size (MOVEM data being displayed as a quad-word) and when pre-decrement addressing is used this is included in the address calculations.

Low Res No hex data is shown for the data registers and the address register data area is reduced to 4 bytes. In addition the disassembly line may not be long enough to display complex addressing modes such as the second example above.

Disassembly Window Display

Disassembly windows display memory as disassembled instructions to the standard described below. On the left the hex address is shown, followed by any symbol, then the disassembly itself. The current value of the PC is denoted with ⇒.

If the instruction has a breakpoint placed on it this is shown using square brackets ([ ]) afterwards, the contents of which depend on the type of breakpoint. For stop breakpoints this will be the number of times left for this instruction to execute, for conditional breakpoints this will be a ? followed by the beginning of the conditional expression, for count breakpoints this will be a = sign followed by the current count, and for permanent breakpoints a * is shown.

The exact format of the disassembled op-codes is Motorola standard, as GenST accepts. All output is upper-case (except lower-case labels) and all numeric output is hex, except Trap numbers. Leading zeroes are suppressed and the `$` hex delimiter is not shown on numbers less than 10. Where relevant numerics are shown signed. The only deviation from Motorola standard is the register lists shown in `MOVEM` instructions - in order to save display space the type of the second register in a range is abbreviated, for example

```
        MOVEM.L d0-d3/a0-a2,-(sp)
```

will be disassembled as

```
        MOVEM.L d0-3/a0-2,-(sp)
```

Low Res Any displayed symbols replace the hex address display, limited to a maximum of 8 characters.

Memory Window Display

Memory windows display memory in the form of a hex address, word-aligned hex display and ASCII. Unreadable memory locations are denoted by **. The number of bytes shown is calculated from the window width, up to a maximum of 16 bytes per line.

Source-code Window Display

The source-code window displays ASCII files in a similar way to a screen editor. The default tab setting is 8 though this can be toggled to 4 with the Edit Window command.

Window Commands

The `Alt` key is generally used for controlling windows, and when used apply to the *current window*. This is denoted by having an inverse title and can be changed by pressing `Tab` or `Alt` plus the window number.

Most window commands work in any window, zoomed or not, though when it does not make sense to do something the command is ignored.

Alt-A Set Address

This sets the starting address of a memory or disassembly window.

Alt-B Set Breakpoint

Allows the setting of any type of breakpoint, described later under **Breakpoints**.

Alt-E Edit Window

On a memory window this lets you edit memory in hex or ASCII. Hex editing can be accomplished using keys `1-9`, `A-F`, together with the cursor keys. Pressing `Tab` switches between hex & ASCII, ASCII editing takes each keypress and writes it to memory. The cursor keys can be used to move about memory. To leave edit mode press the `Esc` key.

On a register window this is the same as `Alt-R`, Register Set, described shortly.

On a source-code window this toggles the tab setting between 4 and 8.

Alt-F Font size

This changes the font size in a window. In high resolution 16 and 8 pixel high fonts are used, in colour 8 and 6 pixel high fonts are used. This allows a greater number of lines to be displayed, assuming your monitor can cope.

Changing the font size on the register window causes the position of windows 2 and 3 to be re-calculated to fill the available space.

Alt-L Lock Window

This allows disassembly and register windows to be locked to a particular register. After any exception the start address of the window is re-calculated, depending on the locked register.

To unlock simply enter a blank string. By default window 2 is locked to the PC. You can lock windows to each other by specifying a lock to a memory window, such as `M2`.

Alt-O — Show Other

This prompts for an expression and displays it in hex, decimal and as a symbol if relevant.

Alt-P — Printer Dump

Dumps the current window contents onto the printer. It can be aborted by pressing `Esc`.

Alt-R — Register Set

Allows any register to be set to a value, by specifying the register, an equals sign, then its new value. It can also be used to set the value of memories. For example the line

```
a3=a2+4
```

sets register `A3` to be `A2` plus `4`. You can also use this to set the start address of windows when in zoom mode so that on exit from zoom mode the relevant window starts at the required address.

Note Do not assign to `M4` if window 4 is currently a source-code window.

Alt-S — Split windows

This either splits window 2 into 2 and 4, or splits window 3 into 3 and 5. Each new window is independent from its creator. Pressing `Alt-S` again will unsplit the window.

Low Res This command has no effect.

Alt-T Change Type

This only works on window 4 (created either by splitting window 2 or by loading a source file). It changes the type of the window between disassembly, memory and source-code (if a file has been loaded).

Alt-Z Zoom Window

This zooms the current window to be full size. Other `Alt` commands are still available and normal size can be achieved by pressing `Esc` or `Alt-Z` again.

Note Zooming the register window is unlikely to be useful.

Cursor Keys

The cursor keys can be used on the current window, the action of which depends on the window type.

On a memory window all four cursor keys change the current address, and `Shift` ↑ and `Shift` ↓ move a page in either direction.

On a disassembly window ↑ and ↓ change the start address on an instruction basis, ← and → change the address on a word basis.

On a source-code window ↑ and ↓ change the display on a line basis, and `Shift` ↑ and `Shift` ↓ on a page basis.

Screen Switching

MonST uses its own screen display and drivers to prevent interference with a program's own screen output. To prevent flicker caused by excessive screen switching when single-stepping the screen display is only switched to the program's after 20 milliseconds, producing a flicker-free display while in the debugger. In addition the debugger display can have a different screen resolution to your program's if using a colour monitor.

V — View Other Screen

This flips the screen to that of the programs, any key returns to the MonST display.

Ctrl-O — Other Screen Mode

This changes the screen mode of MonST's display between low and medium resolution. It re-initialises window font sizes and positions to the initial display. This will not effect the screen mode of the program being debugged.

This command is ignored on a monochrome monitor.

As MonST has its own idea of where the screen is, what mode it is in and what palettes to use you can use MonST to actually look at the screen memory in use by your program, ideal for low-level graphics programs.

Note If your program changes screen position or resolution, via the XBIOS or the hardware registers, it is important that you temporarily disable screen switching using Preferences while executing such code else MonST will not notice the new attributes of your program's screen.

When a disk is accessed, when loading or saving, the screen display will probably switch to the program's during the operation. This is in case a disk error occurs, such as write-protected or read errors, as it allows any GEM alert boxes to be seen and acted upon.

Breaking into Programs

Shift-Alt-Help — Interrupt Program

While a program is running it can be interrupted by pressing this key combination, which will cause a trace exception at the current value of the PC. With computationally-intense program sections this will be within the program itself but with a program making extensive use of the ROM, such as the BDOS or AES, the interruption will normally be in the ROM itself, or the line-F handler stored in low-memory. If this is the case it is recommended that a breakpoint be placed in your actual program area then a Return to Program command (`Ctrl-R`) issued.

Pressing `Alt-Help` without the `Shift` key will normally produce a screen dump to the printer - if you press this accidentally it should be pressed again to cancel the dump.

It is possible for this key combination to be ignored when pressed - if this occurs press it again when it should work. Pressing it when in MonST itself will produce no effect.

Note A program should never be terminated (using `Ctrl-C`) if it has just been interrupted in the middle of a ROM routine. This is likely to cause a system crash.

Breakpoints

Breakpoints allow you to stop the execution of your program at specified points within it. MonST allows up to eight simultaneous breakpoints, each of which may be one of five types. When a breakpoint is hit MonST is entered and then decides whether or not to halt execution of your program, entering the front panel display, or continue; this decision is based on the type of the breakpoint and the state of your program's variables.

Simple Breakpoints

These are one-off breakpoints which, when executed, are cleared and cause MonST to be entered.

Stop Breakpoints

These are breakpoints that cause program execution to stop after a particular instruction has been executed a particular number of times. In fact a simple breakpoint is really a stop breakpoint with a count of one.

Count Breakpoints

Merely counters; each time such a breakpoint is reached a counter associated with it is incremented, and the program will resume.

Permanent Breakpoints

These are similar to simple breakpoints except that they are never cleared - every time execution reaches a permanent breakpoint MonST will be entered.

Conditional Breakpoints

The most powerful type of breakpoint and these allow program execution to stop at a particular address only if an arbitrarily complex set of conditions apply. Each conditional breakpoint has associated with it an expression (conforming to the rules already described). Every time the breakpoint is reached this expression is evaluated, and if it is non-zero (i.e. true) then the program will be stopped, otherwise it will resume.

Alt-B Set Breakpoint

This is a window command allowing the setting or clearing of breakpoints at any time. The line entered should be one of the following forms, depending on the type of breakpoint required:

<address>

will set a simple breakpoint.

<address>,<expression>

will set a stop breakpoint at the given address, after it has executed <expression> times.

<address>,=

will set a count breakpoint. The initial value of the count will be zero.

<address>,*

will set a permanent breakpoint.

<address>,?<expression>

will set a conditional breakpoint, using the given expression.

<address>,-

will clear any breakpoint at the given address.

Breakpoints cannot be set on addresses which are odd or unreadable, or in ROM, though ROM breakpoints may be emulated using the Run Until command.

Every time a breakpoint is reached, regardless of whether the program is interrupted or resumed, the program state is remembered in the History buffer, described later.

Help — Show Help and Breakpoints

This displays the text, data and BSS segment addresses and lengths, together with every current breakpoint. Alt-commands are available within this display.

Ctrl-B — Set Breakpoint

Included mainly for compatibility with MonST 1, this sets a simple breakpoint at the start address of the current window, so long as it is a disassembly window. If a breakpoint is already there then it will be cleared.

U — Go Until

This prompts for an address, at which a simple breakpoint will be placed then program execution resumed.

Ctrl-K Kill Breakpoints

This clears all set breakpoints.

Ctrl-A Set Breakpoint then Execute

A command that places a simple breakpoint at the instruction *after* that at the PC and resumes execution from the PC. This is particularly for DBF-type loops if you don't want to go through the loop, but just want to see the result after the loop is over.

Ctrl-D BDOS Breakpoint

This allows a breakpoint to be set on specific BDOS calls. The required BDOS number should be entered, or a blank line if any existing BDOS breakpoint needs to be cleared.

History

MonST has a *history buffer* in which the machine status is remembered for later investigation.

The most common way of entering data into the history buffer is when you single-step, but in addition every breakpoint reached and every exception caused enters the machine state into the buffer. Various forms of the Run command also cause entries to be made into this buffer.

Note The history buffer has room for five entries - when it fills the oldest entry is removed to make room for the newest entry.

H Show History Buffer

This opens a large window displaying the contents of the history buffer. All register values are shown including the PC as well as a disassembly of the next instruction to be executed.

Note If a disassembly in the History display includes an instruction which has a breakpoint placed on it the `[ ]`s will show the *current* values for that breakpoint, not the values at the time of the entry into the history buffer.

Quitting MonST

Ctrl-C Terminate

This will issue a terminate trap *to the current GEMDOS task*. If a program has been loaded from within MonST it will be terminated and the message `Program Terminated` appear in the lower window. Another program can then be loaded, if required.

If no program has been loaded into MonST it will itself terminate when this command is used.

If the Debug option has been used from the GenST editor then MonST will terminate automatically when the program it is debugging has terminated.

Note Terminating some GEM programs prematurely, before they have closed workstations or restored window control properly can seriously confuse the AES and VDI. This may not be noticeable immediately but often causes crashes when a subsequent program is executed.

Loading & Saving

Ctrl-L Load Executable Program

This will prompt for an executable filename then a command line and will attempt to load the file ready for execution. If MonST has already loaded a program it is not possible to load another until the former has terminated.

The file to be loaded must be an executable file - attempting to load a non-executable file will normally result in TOS error 66 and further attempts to load executable files will normally fail as GEMDOS does not de-allocate the memory it allocated before trying to load the errant file. If this occurs terminate MonST then re-execute it and use the Load Binary File command.

Note This command is not available in the auto-resident version of MonST or in MonST invoked using Debug from the editor.

B Load Binary File

This will prompt for a filename and optional load address (separated by a comma) and will then load the file where specified. If no load address is given then memory will be allocated from GEMDOS and used. `M0` will be set to the start address and `M1` to the end address.

S Save Binary File

This will prompt for a filename, a start address and an (inclusive) end address. To re-save a file recently loaded with the above command `<filename>,M0,M1` may be specified, assuming of course that `M0` and `M1` have not been re-assigned.

A Load ASCII File

This powerful command allows an ASCII file, normally of source code, to be loaded and viewed within MonST. Window 4 will be created if required then set up as a source-code window. Memory for the source code is taken from GEMDOS so sufficient free memory must be available. It is recommended that source-code be loaded *before* an executable program to ensure enough memory.

Low Res Window 4 is not though an ASCII file may may be loaded in low-res then viewed after switching to medium resolution using `Ctrl-O` and pressing `Alt-S`, `Alt-T`, `Alt-T`.

Note If an ASCII file is loaded *after* an executable program the memory used will be owned by the *program itself*, not MonST. When such a program terminates, any displayed source-code window will be closed. The auto-resident version of the debugger cannot detect this so care should be taken if loading source code into it.

Executing Programs

Ctrl-R Return to program / Run

This runs the current program with the given register values at full speed and is the normal way to resume execution after entry via a breakpoint.

Ctrl-Z Single-Step

This single-steps the instruction at the PC with the current register values. Single-stepping a Trap, Line-A or Line-F opcode will, by default, be treated as a single instruction. This can be changed using Preferences.

Ctrl-Y Single-Step

Identical to `Ctrl-Z` above but included for the convenience of German users.

Ctrl-T Interpret an Instruction (Trace)

This interprets the instruction at the PC using the displayed register values. It is similar to `Ctrl-Z` but *skips over* BSRs, JSRs, Traps, Line-A and Line-F calls, re-entering the debugger on return from them to save stepping all the way through the routine or trap. It works on instructions in ROM or RAM.

R Run (various)

This is a general Run command and prompts for the type of the Run to be done, selected by pressing a particular key.

Run G Go

This is identical to `Ctrl-R`, Run, and resumes the program at full speed.

Run S Slowly

This will run the program at reduced speed, remembering every step in the history buffer.

Run I Instruction

This is similar to Run Slowly but allows a count to be entered, so that a particular number of instructions may be executed before MonST is entered.

Run U Until

You will be prompted for an expression which will be evaluated after every instruction. The program will then run, albeit at reduced speed, until the given expression evaluates to non-zero (true) when MonST will be entered. For example if single-stepping a `DBF` loop which used d6 in the ROM code you could say `Run Until d6&ffff=ffff` (waiting for the low word of `d6` to be `$FFFF`) or, alternatively, `PC=FC8B1A`, or whatever.

Note This should not be confused with the Until command, which takes an address, places a breakpoint there then resumes execution.

With all of these commands (except Run Go) you will then be asked `Watch Y/N?` If `Y` is selected then the MonST display will be shown after every instruction and you can watch registers and memory as they change, or interrupt execution by pressing both `Shift` keys simultaneously. If `N` is selected then execution will occur while showing your program's display and execution may be interrupted by pressing `Shift-Alt-Help`.

Note Selecting Watch mode with screen switching turned off is likely to result in a great deal of eye strain as the display will be flipped after each and every instruction, particularly alarming with colour monitors.

With any of these Run modes (except Go) all information after every instruction will be remembered in the history buffer. In addition Traps will be treated as single-instructions, unless changed with Preferences, though see the warnings under that command about tracing all the way through ROM routines.

When a program is running with one of the above modes a couple of pixels near the top left of the display will flicker, to denote that something is happening, as it is possible to think the machine has hung when, in fact, it is simply taking a while to Run through the code an instruction at a time.

Searching Memory

G search memory (Get a sequence)

This will prompt `Search for B/W/L/T/I?`, standing for Bytes, Words, Longs, Text and Instructions.

If you select `B`, `W` or `L` you will then be prompted to enter the sequence of numbers you wish to search for, each separated by commas. MonST is not fussy about word-alignment when searching, so it can find longs on odd boundaries, for example.

If you select `T` you may search for any given text string, which you will be prompted for. The search will be case-dependent.

If you select `I` you can search for part or all of the mnemonic of an instruction, for example if you searched for `$14(A` you would find an instruction like `MOVE.L D2,$14(A0)`. The case of the string you enter *is* important (unlike MonST version 1), but you should bear in mind the format the disassembler produces, e.g. always use hex numbers, refer to `A7` rather than `SP` and so on.

Having selected the search type and parameters, the search begins, control passing to the Next command, described below.

This can be used after the G command to find subsequent occurrences of the search data. With the B, W, L and T options you will always find at least one occurrence, which will be in the buffer within MonST that is used for storing the sequence. With the T option you may also find a copy in the system keyboard buffer. With these options, the Esc key is tested every 64k bytes and can be used to stop the search. With the I option, which is very much slower, the Esc key is tested every 2 bytes.

The search area of memory goes from 0 to the end of RAM, then from $FA0000 to $FEFFFF (the cartridge and system ROM area), then back to 0.

The search will start just past the start address of the current window (except register windows) and if an occurrence is found re-display the window at the given address.

Searching Source-Code Windows

If the G command is used on a source-code window the T sub-command is automatically chosen and if the text is found the window will re-display the line containing it.

Miscellaneous

Ctrl-P Preferences

This permits control over various options within MonST. The first three require Y/N answers, pressing Esc aborts or Return leaves them alone.

Screen Switching

Defaulting to On, this causes the display to switch to your program's only after 20 milliseconds. It should be switched off when a program is about to change a screen's address or resolution, then turned back on afterwards.

Follow Traps

By default single-stepping and the various forms of the Run command treat Traps, Line-A and Line-F calls as single instructions. However by turning this option On the relevant routines will be entered allowing ROM code to be investigated.

Note **Important:** this option should be used with care. Certain time critical routines, such as the floppy- or hard-disk drivers have portions of code designed to be atomic, i.e. not interruptable, and being traced will cause malfunctions within such code and possible loss of data. On the other hand it can be fun to watch the AES as it draws pull-down menus or opens windows.

If you have let ROM execute for a while you can interrupt it by pressing `Shift-Alt-Help`, then resume at normal speed by pressing `Ctrl-R`. However the AES and VDI both use Line-A and Line-F calls and it is very likely that there are pending stack frames left with the Trace bit set, so having resumed a traced program it is likely that seemingly spurious trace exceptions will be generated. Pressing `Ctrl-R` will resume at normal speed, though a few more such exceptions are likely until program flow reaches the *lowest level*, i.e. your program.

There is a side effect of this that can cause machine to crash though: if you have traced through any AES event-type calls then stack frames can be created *in desk accessories* with the Trace bit set. If your program terminates before the accessory has a chance to respond to its own event call, a trace exception will occur *after* MonST terminates and returns to the Desktop or GenST, causing a system crash, unless an auto-resident MonST is installed or the `NOTRACE.PRG` program is used.

NOTRACE Program

This is a very small program intended to be added to the `AUTO` folder of your boot disk which causes trace exceptions to be ignored, instead of producing a large number of bombs as it will do by default. The source code is also supplied.

Relative Offsets

This option defaults to On and affects the disassembly of the *address register indirect with offset* addressing modes, i.e. `xxx(An)`. With the option on the current value of the given address register is added to the offset then searched for in the symbol table. If found it is disassembled as `symbol(An)`. This option is very useful for certain styles of assembly language programming as well as high level languages which use a base register as a major offset, such as **HiSoft BASIC** which uses `A3` as a pointer to the run-time system.

Symbols Option

This allows control over the use of symbols in expressions in MonST. It will firstly ask whether the case of symbols should be ignored, pressing `Y` will cause case independent searching to be used. It will then prompt for the maximum length of symbols, which is normally 22 but may be reduced to as low as 8.

I Intelligent Copy

This copies a block of memory to another area. The addresses should be entered in the form

```
<start>,<inclusive_end>,<destination>
```

The copy is intelligent in that the block of memory may be copied to a location which overlaps its previous location.

Note No checks at all are made on the validity of the move; copying to non-existent areas of memory is likely to crash MonST and corrupting system areas may well crash the machine.

L List Labels

This opens up a large window and displays all loaded symbols. Any key displays the next page, pressing `Esc` aborts. The symbols will be displayed in the order they were found on the disk (or in memory if using the Debug option from the editor).

W — Fill Memory With

This fills a section of memory with a particular byte. The range should be entered in the form

```
<start>,<inclusive_end>,<fillbyte>
```

The warning described previously about no checks applies equally to this command.

P — Disassemble to Printer/Disk

This command allows the disassembly of an area of memory to printer or disk, complete with original labels and, optionally, an automatic list of labels created by MonST, based on cross-references. The first line should be entered as

```
<start_address>,<end_address>
```

The next line prompts for the area of memory used to build the cross-reference list, which should be left blank if no automatic labels are required else should be of the form

```
<buffer_start>,<buffer_end>
```

Next is the prompt for data areas which will be disassembled as `DC` instructions, of the form

```
<data_start>,<data_end>[,<size>]
```

The optional size field should be `B`, `W` or `L`, defaulting to `L`, determining the size of the data. When all data areas have been defined, a blank line should be entered.

Finally a filename prompt will appear; if this is blank all output will be to the printer, else it will be assumed to be a disk file.

If automatic labels were specified there may be a delay at this point while the table is generated. Automatic labels are of the form `Lxxxxx` where `xxxxx` is the actual hex address.

Printer Output

This is of the form of an 8 digit hex number, then up to 10 words of hex data, 12 characters of any symbol, then the disassembly itself. Printer output may be aborted by pressing `Esc`.

Disk Output

This is in a form directly loadable by GenST, consisting of any symbol, a tab, then the disassembly itself, with a tab separating any operand from the op-code. If you are disassembling an area of memory without loaded symbols then the XREF option should be used else no symbols will appear at all in the output file. Pressing `Esc` or a disk error will abort the disassembly.

M Modify Address

Included for compatibility with MonST 1, equivalent to `Alt-A`.

O Show Other Bases

Included for compatibility with MonST 1, equivalent to `Alt-O`.

D Change Drive & Directory

This allows the current drive and sub-directory to be changed.

Auto-Resident MonST

The additional version of MonST called `AMONST2.PRG` will now be described. When placed in the `AUTO` folder on a boot disk, it will be loaded and initialised automatically on boot-up.

Once booted, this version of MonST lies *dormant*, ready to be invoked when any exception occurs in the machine, such as an address error. It is intended primarily for programmers writing and debugging desk accessories or other AUTO-type applications, as if there is a problem in the code which gets called as the machine boots, it hangs before you get a chance to use the normal MonST. If required you can deliberately put an illegal opcode, such as `ILLEGAL`, at the start of your auto program so that MonST will be invoked and then use it to investigate any problems your code has.

The auto-resident version may be double-clicked from the Desktop and will initialise itself in the same way as from the `AUTO` folder, unless a version of MonST is already resident.

Once invoked the auto-resident version is very similar in use to the other versions except that programs or labels cannot be loaded and the base page variables are unknown and so set to 0. The other difference is that when the program being debugged exits or `Ctrl-C` is pressed within MonST, MonST itself stays active in memory.

In addition any program may be interrupted by pressing the `Shift-Alt-Help` key combination when a resident version of MonST is installed.

The resident version of MonST cannot be reclaimed from memory except by resetting the machine and booting with a disk which does not contain MonST in the `AUTO` folder.

When an auto-resident version of MonST is loaded, the usual versions can still be used as normal, memory permitting, and the resident version will be ignored until the non-resident version exits, when it will become active once again.

Note Do *not* invoke an auto-resident MonST from within a program other than the Desktop, such as using Run Other from within GenST, as large areas of system memory will become locked away and unusable until a machine reset.

If both `shift` keys are held down during the installation of the auto-resident MonST, the debugger is itself entered, allowing the editing of memory or setting of BDOS breakpoints. When entered via this method the debugger should be left using `Ctrl-C` when the debugger will remain resident.

Command Summary

Window Commands

Alt-A	Set Address
Alt-B	Set Breakpoint
Alt-E	Edit Window
Alt-F	Font Size
Alt-L	Lock Window
Alt-O	Show Other
Alt-P	Printer Dump
Alt-R	Register Set
Alt-S	Split Windows
Alt-T	Change Type
Alt-Z	Zoom Window

Screen Switching

V	View Other Screen
Ctrl-O	Other Screen Mode

Breakpoints

Alt-B	Set Breakpoint
Help	Show Help and Breakpoints
Ctrl-B	Set Breakpoint
U	Go Until
Ctrl-K	Kill Breakpoints
Ctrl-A	Set Breakpoint then Execute
Ctrl-D	BDOS Breakpoint

Loading and Saving

Ctrl-L	Load Executable Program
B	Load Binary File
S	Save Binary File
A	Load ASCII File

Executing Programs

Ctrl-R	Return to program / Run
Ctrl-Z	Single-Step
Ctrl-Y	Single-Step
Ctrl-T	Interpret an Instruction (Trace)
R	Run (various)

Searching Memory

G	Search Memory (Get a sequence)
N	Find Next

Miscellaneous

Ctrl-C	Terminate
Ctrl-P	Preferences
I	Intelligent Copy
W	Fill Memory With
L	List Labels

`P`	Disassemble to Printer/Disk
`M`	Modify Address
`O`	Show Other Bases
`D`	Change Drive & Directory
`Shift-Alt-Help` .	Interrupt Program
`H`	Show History Buffer

Debugging Stratagem

Hints & Tips

If you have interrupted a program using `Shift-Alt-Help` or by a Run Until command and have found yourself in the middle of the ROM, there is a way of returning to the exact point in your program which called the ROM. Firstly ensure the Follow Traps option is on, then do Run Until with an expression of `sp=a7`. This will re-enter MonST the moment user mode is restored which will be in your program.

If you are in a subroutine which doesn't interest you and want to let it run but return to MonST the easiest way is to use Until (not Run Until) then specify the expression `{sp}` - this sets a breakpoint at the return address. If the subroutine has placed something on the stack, or uses a local stack frame (normally the case for compiled programs) then try Run Until `{pc}.w=4e75` which will run slowly until the instruction `RTS` is reached. This won't work if the subroutine in question calls another, so it may require a further condition, such as `({pc}.w=4e75)&(sp>xxx)` where `xxx` is one less than the current value.

When using Run Until and you know it will take a quite a while for the condition to be satisfied, give MonST a hand by pre-computing as much of the expression as you can, for example

```
(a3>(3A400-\100+M1))
```

could be reduced to

```
a3>xxx
```

where `xxx` has been calculated by you using the `Alt-O` command.

MonST Command Line

If you use a CLI-type program you can pass a command line to MonST, consiting of the program you wish to load and, optionally, a command line to pass on to it.

Bug Hunting

There are probably as many strategies for finding bugs as there are programmers; there is really no substitute for learning the hard way, by experience. However, here are some hints which we have learnt, the hard way!

Firstly, a very good way of finding bugs is to look at the source code and think. The disadvantage of reaching first for the debugger, then second for the source code, is that it gets you into bad habits. You may switch to a machine or programming environment that does not offer low-level debugging, or at least not one as powerful you are used to.

If a program fails in a very detectable way, such as causing an exception, debugging is normally easier than if, say, a program sometimes doesn't quite work exactly as it should.

Many bugs are caused by a particular memory location being stepped on. Where the offending memory location is detectable, by producing a bus error, for example, a conditional breakpoint placed at one or more main subroutines can help greatly. For example, suppose the global variable `main_ptr` is somehow becoming odd during execution, the conditional expression could be set up as

```
        (main_ptr)&1
```

If this method fails, and the global variable is being corrupted somewhere un-detectable, the remaining solution is to Run Until that expression, which could take a considerable time. Even then it may not find it, for example if the bug is caused by an interrupt happening at a certain time when the stack is in a particular place.

Count breakpoints are a good way of tracking down bugs *before* they occur. For example, suppose a particular subroutine is known to eventually fail but you cannot see why, then you should set a count breakpoint on it, then let the program run. At the point where the program stops, because of an exception say, look at the value of the count breakpoint (using Help). Terminate the program, re-load it, then set a stop breakpoint on the subroutine for that particular value or one before it. Let it run, then you can follow through the sub-routine on the very call that is fails on, to try and work out why.

Good luck!

AUTO-folder programs

If these crash during initialisation then use `AMONST` (which must be before your program in the directory) to catch the exception. Including a deliberate `ILLEGAL` instruction at its beginning will let you single-step the initialisation.

Desk Accessories

If an accessory is mis-behaving during normal execution then use `AMONST`. To find a desk accessory in memory, enter the debugger by pressing `Shift-Alt-Help` then start looking from location 0 for the upper-cased name of your `.ACC` file, padded to eight characters with spaces. Ignore occurrences within directory buffers (these will be followed by `.ACC`) and in MonST's own buffer (these will be preceded by an ASCII T character). The correct occurrence will have a longword 12 bytes after the start of the name. This will point to the basepage of your accessory and $100 bytes after that will be the start of your program. From looking at this you should be able to find your main loop and set a suitable breakpoint. Normal execution should be resumed with `Ctrl-R` then MonST will be re-entered when your breakpoint is reached.

If an accessory is misbehaving during its initialisation then you have to stop it at the very beginning before it has a chance to do anything. The recommended way is to re-assemble the accessory with an `ILLEGAL` instruction at the beginning and let `AMONST` catch it, but this is sometimes not possible. There follows a method that works on current ST ROMs to stop the AES just before it executes your program, but please note the method is complicated and not recommended for beginners

Firstly hold down both shift keys to enter AMONST during the boot sequence then set a BDOS Breakpoint on the Open call, $3D, then press Ctrl-C to let the boot sequence resume.

MonST will be re-entered every time something tries to Open a file, so make window 3 the current window and after every BDOS breakpoint is hit set its address to {sp+2} - if the name is *not* your accessory then Ctrl-Z, to execute the Open call, set another BDOS breakpoint on $3D then Ctrl-R, and try again. If the name *is* your accessory then set a BDOS breakpoint on $4B, then Ctrl-R. MonST will then be entered just before it loads the accessory, so Ctrl-Z to do the GEMDOS call, then Alt-B and enter d0+100 which sets a breakpoint on the very first instruction. Now Ctrl-R and the next time MonST appears it will be on the first instruction of the accessory. This method takes a while but it's often the only way of finding bugs in accessories.

Exception Analysis

When an unexpected exception occurs, it's very useful to be able to work out where and why it occurred and, possibly, to resume execution.

Bus Error

If the PC is in some non-existent area of memory then look at the relevant stack to try and find a return address to give a clue as to the cause, probably an unbalanced stack. If the PC is in a correct area of your program then the bus error must have been caused by a memory access to non-existent or protected memory. Recovering from bus errors and resuming execution is generally not possible.

Address Error

If the PC is somewhere strange the method above should be used, otherwise the error must have been caused by a program access to an odd address. Correcting a register value may be enough to resume execution, at least temporarily.

Illegal Instruction

If the PC is in very low memory, below around $30, it is probable that it was caused by a jump to location 0. If you use MonST to look here you will see a short branch together with, normally, various ORI instructions (really longword pointers) and eventually an illegal instruction.

Privilege Violation

This is caused by executing a privileged instruction in user mode, normally meaning your program has gone horribly wrong. Bumping the PC past the offending instruction is unlikely to be much help in resuming the program.

CHAPTER 5
Linker

Introduction

A linker's job is to accept one or more input files generated with GenST or a high-level language compiler and create a single executable file from it. One of the most powerful features is library searching, which means that the linker will only use the parts of a library of modules that are required by other sections of the program, resulting in much smaller output files.

There are unfortunately two different linker file formats on the Atari ST, known as DRI- and GST-formats. While GenST can generate both formats, only the GST-format is supported in the LinkST linker. To link DRI code you need either the Atari `ALN` program or the Digital Research `LINK68` and `RELMOD` programs.

Note

LinkST will only link GST format files.

Invoking the Linker

The simplest way to run the linker from the Desktop is to double-click on the `LINKST.TTP` icon, and enter a suitable command line. There is another way to invoke the linker, using a control file which contains the required options.

The command line contains the necessary information for the linker to read all the relevant files, and generate an output file.

Command Line

The command line should be of the form:

```
<filename> <-options>   [filename]  [-options]
```

Options are denoted by a - sign then an alphabetic character, allowed options being:

B	generate a true BSS section for any such named sections
D	debug - include all symbols in the binary file using DR standard 8 character format (for MonST or other debuggers)
F	force pass 2 of the linker, useful if you want to see all errors (as any pass 1 errors will by default stop the link before the second pass)
L	specify that all following filenames are library filenames
M	dump a map file showing the order of the sections and labels, will be the main filename with an extension of `.MAP`
O	specify object code filename, may be followed by white space before filename
Q	'quiet' mode, which disables the pause after the link
S	dump a symbol table listing, will be the main filename with an extension of `.SYM`
X	extended debug, using the *HiSoft Extended Debug* format
W	specify control file filename, defaults to `.LNK` extension

Normally any filenames given are taken to be input files, defaulting to the extension of `.BIN`, though if a `.LNK` extension is specified it will be taken to be a control file, or after a `-L` option filenames are all assumed to be libraries.

The output filename can be specified with the `-o` option on the command line, or using the `OUTPUT` directive in the control file. If there is more than one of these, the last one is used. If there is none, then the first input filename specified in the command line or control file is used with an extension of `.PRG`.

```
PART1 PART2 -d
```

Reads `PART1.BIN` and `PART2.BIN` as input files, and generates `PART1.PRG` as an output file complete with debugging information.

```
PART1 PART2 -o TEST.PRG
```

Reads `PART1.BIN` and `PART2.BIN` as input files, and generates `TEST.PRG` as an output file.

```
-o TEST.TOS START -l MYLIB -s
```

Reads `START.BIN` as an input file, selectively reads `MYLIB.BIN` as a library, and generates the output file `TEST.TOS` and the symbol listing file `TEST.SYM`.

LinkST Running

LinkST has two passes - during pass 1 it builds up a symbol table of all sections and modules, and during pass 2 it actually creates the output file. When it starts it prints a logon message, then reports on which files it is reading or scanning during both passes. This gives you some idea of what takes time to do, as well as exactly where errors have occurred.

If there is enough free memory at the end of pass 1 LinkST will use a cache to store the output file, which greatly speeds up the process. If it uses the cache it will write to the disk at the end of pass 2, and report the number of errors.

When the link finishes you will be prompted to press a key before quitting. This is to give you an opportunity to read any warning or error messages before returning to the Desktop. You can disable this pause by using the `-q` option, useful if you are using a CLI or batch file program.

LinkST was especially optimised for speed, though the speed of the ST floppies is still a restricting factor. If you can't afford a hard disk we recommend the use of a RAM disk which can make great improvements, **but leave enough memory free for the linker to cache your output file**. If you are limited in what you can fit on your RAM disk we recommend you put many small library or input files on there.

Error and warning messages are directed to the screen - if you want to pause output you can press `Ctrl-S`, and `Ctrl-Q` will resume it. Pressing `Ctrl-C` will abort the linker immediately. You can re-direct screen output to a disk file by starting the command line with

```
        >FILENAME.TXT
```

or you can re-direct it to a printer by starting the command line with

`>PRN:` (parallel port) or `>AUX:` (serial port)

If you do re-direct output in this way you should use the `-q` option as you won't be able to see the prompt at the end of the linking.

Control Files

The alternate way to run the linker is to have a control file for the programs which you are linking together.

If you require a lot of options which won't fit on the command line or you get bored of typing them you can use a control file, which is a text file containing commands and filenames for the linker. The default extension is `.LNK`, and the text file can be generated with GenST (though don't try and assemble it !). The control filename is specified on the command line with the `-w` (for With) option, and each line can be one of the following:

INPUT <filename>

This specifies a filename to be read as an input file. The default extension is `.BIN` if none is given.

OUTPUT <filename>

This specifies the filename to be used for the output file. There is no default extension - you should specify it explicitly.

LIBRARY <filename>

This specifies a filename to be scanned as a library. The default extension is `.BIN` if none is given.

SECTION <sectionname>

This allows specific section ordering to be forced.

DEBUG

All symbol names included in the link are put in the output file so that debugging programs such as MonST can use them when the program is running.

XDEBUG

Similar to debug option but uses *HiSoft Extended Debug* format for up to 22 character significance.

DATA size(K)

The BSS segment size is set accordingly. The size can be given either as a number of bytes or as a number of K-bytes (units of 1024). This option is particularly useful for compilers like **Prospero Pascal** which store their variables in the BSS segment. Blank lines in the control file are ignored, and comments can be included by making the first character in the line a *, a ; or a !.

BSS <sectionname>

Specifics that the named section should lie in the GEMDOS BSS section area. This can save valuable disk space, but will generate errors if the section contains any non-zero data. This should not be used at the same time as the DATA statement.

With the INPUT or OUTPUT directive if the filename is specified as * it is substituted for the first filename on the command line. This can be useful for having a generic control file for linking C programs, for example.

An example control file is:

```
* control file for linking C program
INPUT STARTUP
INPUT *
XDEBUG
LIBRARY CLIB
```

Assuming this control file is called `CPROG.LNK`, the LinkST command line

```
TEST -w CPROG
```

will read as input files `STARTUP.BIN` and `TEST.BIN`, and scan the library `CLIB.BIN`. The object code, including extended debug information, will be written to `TEST.PRG`, as none was explicitly specified.

If you do not specify a drive name in the control file or on the command line, the default drive will be assumed. If you run LinkST from the Desktop, the default drive will always be the same as the file on which you double-clicked; though if you run it from a CLI or from the GenST editor this will not necessarily be so.

Automatic Double-Clicking

It is possible to install LinkST so that you can double-click on a `.LNK` file from the Desktop to invoke the linker, by using the Install Application option from the Desktop. This is a similar process to that described for GenST, except the type should be left as TOS Take Parameters and the extension should be `.LNK`

LinkST Warnings

Warnings are messages indicating that something might be wrong, but it's nothing too serious.

```
duplicate definition of value for symbol "x"
```

The symbol was defined twice. This can happen if you replace a subroutine in a module with one of your own, for example. The linker will use the first definition it comes across, and give this warning on the second.

`module name is too long`

Module names can only be 80 characters long.

`comment is too long`

Comment directives are only allowed to be 80 characters long (don't ask us why, we don't know!).

`absolute sections overlap`

Two absolute sections clash with each other.

`SECTION "x" is neither COMMON nor SECTION`

A section name was specified without defining its type.

LinkST Errors

LinkST errors divide into four areas: general errors, I/O errors, binary file errors, and linker bugs. In some error messages a string is included, denoted by "x" below. In others a number may be output, denoted by 99 below.

General Errors

`unresolved symbol "x" in file "x"`

The symbol was referred to but not defined in the file. There may also be other files which refer to the symbol, but this gives you a start in your search!

`XREF value truncated`

A value was too large to fit into the space allocated for it, for example a `BSR` to an external may be out of range.

`bad control line "x"`

An illegal line was found in a control file.

`non-zero data in BSS section`

A section wanted as a true BSS section contained non-zero data.

Input/Output (I/O) Errors

```
file "x" not found
Can't open output file "x"
Can't open map file "x"
Can't open symbol file "x"
Can't open input file "x"
i/o error on input file
disk write failed
filename "x" was too long
```

Binary File Errors

These are errors in the internal syntax of the input file, and should not occur. If they do it probably means the compiler or assembler produced incorrect code.

```
missing SOURCE directive
```

Can occur if a file is not in GST format, for example a DRI file.

```
runtime relocation is only available for LONGs
attempt to redefine id of symbol "x"
attempt to DEFINE "x" with <id> of zero
bad operator code 0x99 in XREF directive
bad truncation rule in XREF
wrongly placed SOURCE directive
bad directive 99
<id> 99 not DEFINEd as a SECTION but used as one
attempted re-use of <id> 99 as SECTION id
attempted re-use of "x" as SECTION name
section is COMMON but being used as though it's not
SECTION is being misused as COMMON
unexpected end of input file
```

'Linker Bug' Messages

These can be produced as a result of internal checks by the linker. If you get one please send us copies of the files you are trying to link!

Appendix A
GEMDOS error codes

This appendix details the numeric GEMDOS errors and their meanings. The error numbers shown are those displayed by MonST and GenST; when calling GEMDOS from your own programs these values will be negative.

0	OK (no error)
1	Fundamental error
2	Drive not ready
3	Unknown command
4	CRC error
5	Bad request
6	Seek error
7	Unknown medium
8	Sector not found
9	No paper
10	Write fault
11	Read fault
12	General error
13	Write protect
14	Medium change
15	Unknown device
16	Bad sectors on format
17	Insert other disk
32	Invalid function number
33	File not found
34	Path not found
35	Too many files open
36	Access denied
37	Invalid handle
39	Insufficient memory
40	Invalid memory block address
46	Invalid drive
49	No more files
50	Disk full (not a GEMDOS error; produced by GenST)
64	Range error
65	Internal error
66	Invalid program load format
67	Setblock failure due to growth restrictions

Appendix B
GenST error messages

GenST can produce a large number of error messages, most of which are pretty well self explanatory. This appendix lists them all in alphabetic order, with clarifications for those which require them.

Please note that GenST is continually being improved and this list may not agree exactly with the version you have, there may be additional messages not documented here.

Errors

If you get a message beginning with `INTERNAL` please tell us - you should never see these.

`.W or .L expected as index size`

`absolute expression MUST evaluate`

`absolute not allowed`

`additional symbol on pass 2`

> somchow a symbol has appeared during pass 2 that did not appear during pass 1

`address register expected`

`addressing mode not allowed`

`addressing mode not recognised`

`BSS or OFFSET section cannot contain data`

> OFFSET sections and non-GST BSS sections can only contain DS directives

`cannot create binary file`

> could be a bad filename, or a write-protected disk, etc.

cannot export symbol

cannot import symbol

cannot nest MACRO definitions or define in REPTs

macro definitions may not be nested or defined within repeat loops

cannot nest repeat loops

comma expected

data register expected

data too large

division by zero

duplicate MODULE name

module names must be unique

error during listing output

listing will be stopped at this point

error during writing binary file

normally disk full.

executable code only

only executable code may be assembled to memory

expression mismatch

normally a syntax error within an expression

fatally bad conditional

there were more ENDCs in a macro than IFs

file not found

forward reference

garbage following instruction

`illegal BSR.S`

a BSR.S to the following instruction is not allowed - change it to BSR

`illegal type combination`

`immediate data expected`

`imported label not allowed`

`include file read error`

`instruction not recognised`

`invalid FORMAT parameter`

`invalid IF expression, ignored`

`invalid MOVEP addressing mode`

`invalid number`

`invalid numeric expansion`

the symbol is not defined or relative or a syntax error

`invalid option`

`invalid printer parameter`

`invalid register list`

`invalid section name, TEXT assumed`

`invalid size`

`line malformed`

`linker format restriction`

the DRI format is restrictive about where it allows imports

`local not allowed`

`missing close bracket`

missing ENDC

there were more IFs then ENDCs

missing quote

misuse of label

not yet implemented

number too large

odd address

option must be at start

ORG not allowed

out of memory

phasing error

should never happen, look investigate immediately before first such error

program buffer full

change the program buffer size when assembling to memory

register expected

relative not allowed

relocation not allowed

repeated include file

each include file may only be included once on each pass

source expired prematurely

within an IF, MACRO or REPT and the source ran out

spurious ENDC

spurious ENDM or MEXIT

spurious ENDR

`symbol defined twice`

`symbol expected`

`undefined symbol`

`user error`

caused by a FAIL directive

`wrong processor`

`XREF's not allowed within brackets`

Warnings

`68010 instruction, converted to MOVE SR`

MOVE CCR, is *not* a 68000 instruction

`branch made short`

by optimising

`directive ignored`

`invalid LINK displacement`

if negative or odd

`offset removed`

xx(An) form reduce to (An) by optimising

`relative cannot be relocated`

`short branch converted to NOP`

`sign extended operand`

data in MOVEQ needed sign extension to fit

`size should be .W`

Appendix C
ST Memory Map

This Appendix details certain information about the ST memory map:

1. Processor Dump area

2. Base Page layout

3. Hardware memory map

Processor Dump Area

When the ST crashes with an exception (i.e. mushrooms or bombs) it stores a copy of the processor's state in an area of memory which is not destroyed by a RESET. Thus after such a crash you can load MonST and investigate the relevant area of memory to try to ascertain what exactly went wrong. If this happens a lot you should the auto-resident version of MonST so you will have a much better idea of the cause of the problem.

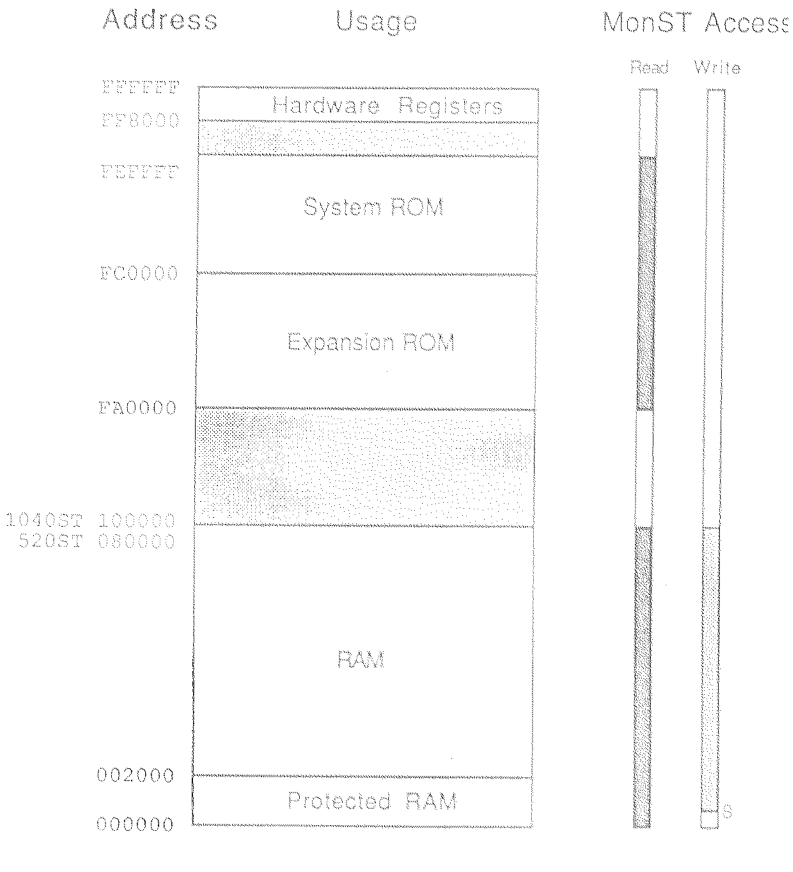
\$380	long	contains \$12345678 if valid
\$384	8 longs	saved values of D0-D7
\$3A4	8 longs	saved values of A0-A7 (SSP)
\$3C4	byte	exception number
\$3C8	long	saved USP
\$3CC	16 words	copied from the SSP

Base Page Layout

Every program that runs under GEMDOS has a base page area which contains certain information. It is $100 bytes long.

Offset	Name	Contents
$00	p_lowtpa	base address of the TPA (i.e. here)
$04	p_hitpa	pointer to end of TPA +1
$08	p_tbase	pointer to start of TEXT area
$0C	p_tlen	length of TEXT area
$10	p_dbase	pointer to start of DATA area
$14	p_dlen	length of DATA area
$18	p_bbase	pointer to start of BSS area
$1C	p_blen	length of BSS area
$20	p_dta	pointer to DTA address
$24	p_parent	pointer to parent's base page (0 if desk acc.)
$28		(reserved)
$2C	p_env	pointer to environment string
$80	p_cmdlin	command line: length byte then string, which is not guaranteed to be null-terminated
$100		your program starts here

Hardware Memory Map



Not to scale

Appendix D Calling the Operating System

The operating system of the ST is large and complex and consists of various levels. To help in your own program development, this appendix describes the calling mechanisms and routines available, but it is not intended to be definitive. It also details the various example programs and include files supplied with DevpacST. The various levels of the operating system are:

GEM AES	window and event manager
GEM VDI	device-independent graphics routines
GEMDOS	disk and screen I/O, similar to MS-DOS
BIOS	low level I/O
XBIOS	extended low level I/O

Each of these will now be described in varying degrees of detail.

GEMDOS - Disk and Screen I/O

GEMDOS was converted from CP/M 68k and is similar in many ways to generic CP/M but with extra facilities (e.g. sub-directories) taken from MS-DOS. It is responsible for disk I/O and character I/O via the screen, keyboard, serial and parallel ports. It is also responsible for memory management.

GEMDOS was designed to be called directly from C, so all parameters are put onto the stack and have to be removed afterwards. The calling sequence from assembler is of this general form:

```
        move     ??,-(a7)        put parameters on stack
        move.w   #??,-(a7)       the function number
        trap     #1              call GEMDOS
        add.l    #??,a7          restore the stack
```

After the call the stack has to be corrected; while an ADD.L can be used as above, it is slow and takes six bytes. If the stack needs correction by 8 bytes or less, the best way is to use

```
        add.l   #??,a7
```

which takes two bytes. If it has to be corrected by more than 8 bytes, the best way is

```
        lea     ??(a7),a7
```

which takes four bytes. Both methods are smaller and faster than the first method. Incidentally, a major source of bugs when starting programming with GEMDOS is forgetting to correct the stack, or correcting it by the wrong value.

Program Startup and Termination

When a GEMDOS program starts up it owns all free memory - that is the memory from the end of the program through to the end of usable RAM (normally just before the screen) is owned by the program, which is just as well as the stack is at the very end of this area.

If any memory management calls (such as m_alloc) are required, you wish to execute other programs within yours, or if you are writing a GEM program, it is important to give back some of this memory. If you don't there will be no free memory for these uses. This is normally done during at the beginning of programs using the m_shrink call, utilising the fact that a pointer to the programs basepage is 4 bytes down on the stack, like this:

```
        move.l  4(a7),a3            basepage
        move.l  $C(a3),d0           text length
        add.l   $14(a3),d0          data length
        add.l   $1c(a3),d0          BSS length
        add.l   #extra,d0           any additional memory
        add.l   #$100,d0            basepage length
        move.l  #mystack,a7         before shrinking
        move.l  d0,-(a7)
        move.l  a3,-(a7)
        clr.w   -(a7)
        move.w  #$4a,-(a7)
        trap    #1                  do the shrink
        lea     12(sp),sp
```

The extra bytes may be required for your programs storage. Note that you should move the stack to a safe area *before* the shrink, otherwise the stack will be in memory that is not owned by your program and liable to corruption.

A GEMDOS program can terminate in one of three ways: p_term0, which is not recommended, p_term, the normal way to finish a program, and p_termres, for system patches and the like. For normal termination use this code:

```
        clr.w    -(a7)              no return code
        move.w   #$4c,-(a7)         p_term
        trap     #1
```

When a program terminates all memory it owns is freed and any open files are closed.

GEMDOS Summary

The calls will now be described in numeric order by giving the size of the parameters, in the order they should be placed on the stack, and the stack correction number. For example, using function call 2, c_conout, to print the character x, the code would be

```
        move.w   #'X',-(a7)         the character
        move.w   #2,-(a7)           the function number
        trap     #1                 call it
        addq.l   #4,sp              then correct
```

At present GEMDOS calls corrupt registers d0 and a0 only, but this is not documented. We therefore recommended programmers to assume that registers d0-d2/a0-a2 are corrupted, in a similar way to the other operating system calls.

0 - Terminate Process (old form), p_term0

Parameters:	None
Result:	None
Stack:	2

This terminates the current program, with a return code of 0. It is recommended that p_term (function $4c) should be used in preference to this call. As control never returns after the call, no stack correction is actually required.

1 - Read character from keyboard, c_conin

Parameters:	None
Result:	D0.L=key code
Stack:	2

This waits for a key to be struck, echoes it to the screen, and returns its value. The long result has the ASCII value in the lowest 8 bits, and a physical key number is returned in bits 16-23. All other bits are set to 0.

2 - Write character to screen, c_conout

Parameters:	word: character
Result:	None
Stack:	4

This writes the given character to the screen. A 16-bit parameter is supported for future expansion, so bytes should always be ANDed with $FF before the call, though currently the upper 8 bits are ignored.

3 - Read character from serial port, c_auxin

Parameters:	None
Result:	D0.B=character read
Stack:	2

This waits for a byte to be received from the auxiliary device, which is the serial port.

4 - Write character to serial port, c_auxout

Parameters:	word:character
Result:	None
Stack:	4

This sends the character out via the serial port. As with function 2, the upper 8 bits of the word should be 0 for upward compatibility.

5 - Write character to printer, c_prnout

Parameters:	word: character
Result:	D0.W=0 if failed, -1 if OK
Stack:	4

This sends the character out via the parallel printer port. As with functions 2 and 4 above, bits 8-15 of the word should be 0.

6 - Raw I/O to standard I/O, c_rawio

Parameters:	word: character for output, or $00FF to read
Result:	D0.W if $00FF passed
Stack:	4

If the character is passed as $00FF then the keyboard is scanned and a result returned in D0.W (or 0 if no key available). If the character is not $00FF, then it is printed on the screen.

7 - Raw input from keyboard, c_rawcin

Parameters:	None
Result:	D0.L=character read
Stack:	2

This waits for a key to be pressed and returns its value. It does not echo it to the screen.

8 - Read character from keyboard, no echo, c_necin

Parameters:	None
Result:	D0.L=character read
Stack:	2

This waits for a key to be pressed and returns its value. It does not echo, but the control keys `Ctrl-C`, `Ctrl-S` and `Ctrl-Q` are interpreted in their usual way - `Ctrl-C` will abort the program, `Ctrl-S` will pause output and `Ctrl-Q` will resume it.

9 - Write string to screen, c_conws

Parameters:	long: address of string
Result:	None
Stack:	6

This writes the given null-terminated string to the screen.

$A - Read edited string from keyboard, c_conrs

Parameters:	long: address of input buffer
Result:	None
Stack:	6

Before calling this, the first byte of the buffer should be set to the size of the data portion of the buffer. On return, the second byte in the buffer will be set to the length of the string, and the string itself starts at the third byte. No CR or null is stored in the returned string and pressing `Ctrl-C` will terminate the entire program.

$B - Check status of keyboard, c_conis

Parameters:	None
Result:	D0.L=-1 if character available, 0 if none
Stack:	2

This returns the status of the keyboard. The key itself should be read with another call.

$E - Set default drive, d_setdrv

Parameters:	word: drive number
Result:	D0.L=bit map of drives in the system
Stack:	4

This sets the default drive; a word of 0 denotes A:, 1 denotes B:, etc. The returned value has a bit set for each installed drive, bit 0=A:, bit 1=B:, etc.

$10 - Check status of standard output, c_conos

Parameters:	None
Result:	D0.L=-1 if ready, 0 if not
Stack:	2

This tests to see if the console device is ready for output.

$11 - Check status of printer, c_prnos

Parameters:	None
Result:	D0.L=-1 if ready, 0 if not
Stack:	2

This tests the status of the printer port. If the printer is ready to receive a character it returns -1, else it returns 0.

$12 - Check status of serial port input, c_auxis

Parameters:	None
Result:	D0.L=-1 if character waiting, 0 if not
Stack:	2

This tests the serial port and returns -1 if there is a character waiting to be read.

$13 - Check status of serial port output, c_auxos

Parameters:	None
Result:	D0.L=-1 if ready, 0 if not
Stack:	2

This tests the serial port and returns -1 if it is ready to receive a character.

$19 - Get default drive, d_getdrv

Parameters:	None
Result:	D0.W=drive number
Stack:	2

This returns the number of the current drive, with A:=0, B:=1 etc.

$1A - Set disk transfer address, f_setdta

Parameters:	long: pointer to disk transfer address
Result:	None
Stack:	6

This sets the address of a 44-byte buffer used for searching for filenames. It must be word-aligned.

$20 - Get into Supervisor/User Mode, s_super

Parameters:	long: value for stack, or 0 or 1
Result:	D0.L=(depends on parameter)
Stack:	6

This has two functions - it can tell you if the program is in User or Supervisor mode and it can switch from one mode to another. To find which mode the processor is in, call this routine with a parameter of 1. The return value will be 0 for user mode, and -1 for supervisor. To switch modes you have to supply a new stack pointer, or pass 0 if you want the stack to remain unchanged. For example, if you are in user mode and want to switch to supervisor mode using a SSP at address myssp, the code would be:

```
        move.l  #myssp,-(a7)
        move.w  #$20,-(a7)
        trap    #1
        addq.l  #6,a7
```

When switching to supervisor mode the old value of the SSP is returned in d0.l. If you only want to go temporarily into Supervisor mode to hack protected memory, for example, XBIOS call supexec is a lot easier.

$2A - Get date, t_getdate

Parameters:	None
Result:	D0.W
Stack:	2

This reads the date, with the result in this format:

Day:	bits 0-4
Month:	bits 5-8
Year:	bits 9-15 (since 1980).

$2B - Set date, t_setdate

Parameters:	word: date
Result:	None
Stack:	4

This sets the date, using the same word format as the previous function.

$2C - Get time, t_gettime

Parameters:	None
Result:	D0.W
Stack:	2

This returns the time of the day, with the result in this format:

Seconds/2:	bits 0-4
Minutes:	bits 5-10
Hours:	bits 11-15

$2D - Set time, t_settime

Parameters:	word: time
Result:	None
Stack:	4

This sets the current time of day, in the same word format as the previous function.

$2F - Get disk transfer address, f_getdta

Parameters:	None
Result:	D0.L=pointer to disk transfer address
Stack:	2

This returns the current disk transfer address, and should always be even.

$30 - Get version number, s_version

Parameters:	None
Result:	D0.W=version number
Stack:	2

This returns the GEMDOS version number, with the major number in the low byte, and the minor number in the high byte. Known releases at this time are:

$0D00	version 0.13 (obsolete disk-based)
$1300	version 0.19 (ROM-based)

$31 - Terminate and stay resident, p_termres

Parameters:	word: exit code, long: bytes to keep
Result:	None
Stack:	8

This allows a program to terminate while keeping part or all of it in memory. It is useful for programs which extend the system, such as RAM disc drivers; if they terminated normally the memory they lie in would get destroyed when the next program loaded. The memory that can be retained is that starting at the base page, and the length parameter should include the $100 of the base page, the required program length, data and stack space if relevant.

$36 - Get drive free space, d_free

Parameters: word: drive code, long: pointer to buffer
Result: None
Stack: 8

This returns various bits of information about a particular disk drive. The drive code should be 0 for the default drive, 1 for A:, 2 for B:, etc. The buffer should be 16 bytes long, and word aligned. On return, it will contain 4 longs of information: free space, number of available clusters, sector size (in bytes), and cluster size (in sectors).

$39 - Create a sub-directory, d_create

Parameters: long: address of pathname
Result: D0.W=0 if OK, else error code
Stack: 6

This creates a new directory, according to the null-terminated string.

$3A - Delete a sub-directory, d_delete

Parameters: long: address of pathname
Result: D0.W=0 if OK, else error code
Stack: 6

This deletes a directory, so long as it has no files or other directories in it.

$3B - Set current directory, d_setpath

Paramcters: long: address of pathname
Result: D0.W=0 if OK, else error code
Stack: 6

This sets the current directory, according to the null-terminated string. Note that drive specifiers are *not* allowed - you should set the current drive then its directory.

$3C - Create a file, f_create

Parameters: word: attributes, long: pointer to string
Result: D0.W=file handle if successful, else error (and longword negative)
Stack: 8

This will attempt to create the given file and if successful will return a file handle that can be used in other file GEMDOS calls. The attribute word can be these values:

01	read only
02	hidden file
04	hidden system file
08	filename contains volume name in first 11 bytes

File handle numbers returned by this call and the following one start are words normally starting at 6 and go upwards. Handles 0 to 5 are standard handles which are already open when a program starts. They correspond to the following devices:

0 - console input
1 - console output
2 - serial port
3 - parallel port

There are three system device names, called CON:, AUX: and PRN: which can be used with this and the following call. They return negative words, so to distinguish these from error returns always TST.L / BMI for the error case.

$3D - Open a file, f_open

Parameters:	word: mode, long: pointer to filename
Result:	D0.W=file handle if successful, else error (and longword negative)
Stack:	8

This will open an existing file for reading, writing, or both. The mode word must be one of the following:

0	open to read
1	open to write
2	open for both reading and writing

If successful this will return a handle which can be used subsequently, else an error number.

$3E - Close file, f_close

Parameters:	word: handle
Result:	D0.W=0 if OK, else an error number
Stack:	4

Given a file handle this will close the file. Do not close a standard handle.

Note This call, along with all the others that require handles, do not do very extensive checks on the validity of the handle. If you pass an invalid one you may get an error return, or the machine may crash!

$3F - Read file, f_read

Parameters:	long: load address, long: number of bytes to read, word: handle
Result:	D0.L=number of bytes read, or an error code
Stack:	12

This will attempt to read bytes from the given file. If an error occurs `D0.L` will be negative. If the end of file is reached during the read operation an error code is not returned - if you wish to check for this you have to compare the number of bytes you asked for with the result - if they are different then you tried to read past the end of file.

$40- Write file, f_write

Parameters:	long: start address, long: number of bytes to write, word: handle
Result:	D0.L=number of bytes written, or an error code
Stack:	12

This will attempt to write bytes to the given file. If an error occurs D0.L will be negative. If the disk becomes full an error code will not be issued, but the value returned will not be the same as the value passed to it as the number of bytes to write.

Note If you pass a negative length parameter GEMDOS will crash very badly

$41 - Delete File, f_delete

Parameters:	long: pointer to filename
Result:	D0.W=0 if successful, else error code
Stack:	6

This will attempt to delete the given file.

$42 - Seek file pointer, f_seek

Parameters:	word: mode, word: file handle, long: position
Result:	D0.L=absolute position in file after seek
Stack:	10

This will move the file pointer to a given position in the file. The mode word should be one of the following:

0	move to N bytes from the start of the file
1	move to N bytes from the current location
2	move to N bytes from the end of the file

If you try and move past either end of the file you will get a result of 0 (for the start) or the actual length of the file.

$43 - Get/Set file attributes, f_attrib

Parameters:	word: attributes, word: get/set, long: pointer to filename
Result:	D0.W=new attributes, or an error code
Stack:	10

This can be used to get or set the attributes for a given file. The attribute's word can be:

01	read only
02	hidden file
04	hidden system file
08	filename is actually the volume label in first 11 bytes
$10	sub-directory
$20	file is written and closed

The other word should be 0 to Get the attribute, or 1 to Set it.

$45 - Duplicate File Handle, f_dup

Parameters:	word: standard handle
Result:	D0.W=new handle, or error code
Stack:	4

Given a handle to a standard device (0-5), this function returns another handle that can be used to address the same device. It can also be closed without affecting the standard device handle.

$46 - Force file handle, f_force

Parameters:	word: non-standard handle, word: standard handle
Result:	D0.W=0 if OK, else error code
Stack:	6

This forces the standard handle to point to the same device or file as the non-standard one, and can be used, for example, to re-direct screen output to a disk file.

$47 - Get Current Directory, d_getpath

Parameters:	word: drive number, long: pointer to buffer
Result:	D0.W=0 if OK, else error code
Stack:	8

Given a drive number (default drive=0, A:=1, B:=2 etc.) this will return the current directory in the given buffer, in null-terminated form. The buffer should be 64 bytes long.

$48 - Allocate Memory, m_alloc

Parameters:	long: number of bytes required
Result:	D0.L=address of memory allocated, or 0 if failed
Stack:	6

This allocates the given amount of memory from the system pool, if available. When a program terminates all its memory allocations are cleaned up. This call can also be used to find the amount of free memory, if -1 is passed.

Note When GEMDOS itself uses this call it always ensures the number of bytes required is even, so we recommend this out of paranoia. This call can occasionally return an odd value for the start of the allocated memory under TOS 1.3.

$49 - Free Allocated Memory, m_free

Parameters:	long: address of area to free
Result:	D0.W=0 if OK, else an error code
Stack:	6

This frees a block of memory allocated with m_alloc above.

$4A - Shrink Allocated Memory, m_shrink

Parameters:	long: length to keep, long: start address to keep, word: 0
Result:	D0.W=0 if OK, else an error code
Stack:	12

This is normally used when a program starts up and releases part of the allocated memory back to GEMDOS.

$4B - Load or Execute a Program, p_exec

Parameters:	long: pointer to environment string, long: pointer to command line, long: pointer to filename, word: mode
Result:	D0.L=(depends on mode)
Stack:	16

This call can be used for loading and chaining programs. The mode word can be one of:

0	load and execute
3	load but do not execute
4	execute base page
5	create base page

For load and execute, the return value is either an error code, or the value returned when the child program exited.

For load but don't execute the return value is either an error code, or a pointer to the base page of the loaded program.

A discussion of using modes 4 and 5 is beyond the scope of this document.

The command line should be of the form of a length byte followed by the line itself.

The environment string may be passed as 0 to inherit the programs parents basepage, or as a pointer to a list of null-terminated environment strings, ending in a double-null. The normal environment looks like this:

```
        dc.b      'PATH=',0,'A:\',0,0
```

$4C - Terminate Program, p_term

Parameters:	word: return value
Result:	N/A as doesn't return
Stack:	N/A

This terminates the current program, returning control to the calling program. The word value returned should be an error code, or 0 for no error. Returned error codes should be positive, to avoid confusion with system error codes, which are negative.

$4E - Search for First, f_sfirst

Parameters:	word: attributes, long: pointer to filespec
Result:	D0.W=0 if found, else -33 not found
Stack:	8

This trap can be used to scan a directory using wild-cards to find all the files. This should be called to find the first one, then f_snext should be called for the rest. When a file is found the parameters of the file are returned in the DTA buffer area. The attribute word determines which file types are to be included in the search, and may be one of:

00	normal files
01	read only files
02	hidden files
04	system files
08	return volume name only
$10	sub-directories
$20	files that have been written to and closed

The returned values in the DTA buffer are:

0-20	reserved for internal use
21	file attributes
22-23	file time stamp
24-25	file date stamp
26-29	file size (long)
30-43	name and extension of file, null terminated

The address of the DTA buffer can be set with function $1A, and read with function $2F.

$4F - Search for Next Occurrence, f_snext

Parameters:	None
Result:	D0.W=0 if found, else -33 not found
Stack:	2

After calling f_sfirst to find the first occurrence of a filespec, this call is used to find subsequent files. When a file is found the DTA buffer is filled as described previously. For it to work the first 20 bytes of the DTA must remain untouched between calls.

$56 - Rename a file, f_rename

Parameters:	long: pointer to new name, long: pointer to old name, word: 0
Result:	D0.W=0 if OK, else error code
Stack:	12

This will attempt to rename the file to the new name. A file with the new name must not already exist.

$57 - Get/Set File Date & Time Stamp

Parameters:	word: 0 for Get / 1 for Set, word: file handle, long: pointer to buffer
Result:	None
Stack:	10

This can be used to get or set the time and date stamp on an open file. The buffer should contain two words, the first being the time, and the second the date, in the format already described.

The ST BIOS is intended for low-level access to the screen, keyboard and disk drives. It is accessed using the stack for parameters as described previously, but using TRAP #13 to invoke it. Programmers who require access to the BIOS are likely to need much more detail than we could provide, so only one BIOS call is described here. For greater BIOS detail see the books in the bibliography. The BIOS handler preserves registers D3-D7/A3-A7 - all others may be corrupted by a call.

BIOS 5 - Set Exception Vector, setexc

This is a very useful trap and sets certain system vectors to point to your own routines. It can set both exception vectors and system vectors. The calling sequence is:

```
move.l   #myroutine,-(a7)   address of new handler
move.w   #vectornum,-(a7)   vector number
move.w   #5,-(a7)           BIOS function number
trap     #13                do it
addq.l   #8,a7              restore stack
move.l   d0,oldroutine      store old one
```

The vector number should be the exception number (2 for bus error, 3 for address error etc.), or one of the following system vectors:

$45	200Hz list
$100	system timer interrupt
$101	critical error handler
$102	process terminate hook

On return from the trap, D0.L contains the previous value. If a program modifies any vectors it should always restore them to their original values before terminating.

If you pass an address of -1 it will not be changed, but the current value will be returned in D0.L.

XBIOS - Extended BIOS

The XBIOS consists of 40 functions for a wide variety of functions including hardware access, screen control, and keyboard mapping. Again we leave most of the description to the books in the bibliography, with the exception of five XBIOS calls. The XBIOS handler preserves registers D3-D7/A3-A7 - all others can be corrupted by a call. The calling sequence is the usual one: put parameters on the stack, put a function word on the stack, do a TRAP #14, then restore the stack. The XBIOS functions are:

XBIOS 2 - Get Physical Screen Address, _physbase

Parameters:	None
Result:	D0.L=start of screen
Stack:	2

This will return the physical address of the screen, which always occupies 32000 bytes and is aligned on a 256-byte boundary.

XBIOS 3 - Get Logical Screen Address, _logbase

Parameters:	None
Result:	D0.L=start of screen
Stack:	2

This will return the logical address of the screen.

XBIOS 4 - Get Screen Resolution, _getRez

Parameters:	None
Result:	D0.W=0 low, 1 medium, 2 high
Stack:	2

This will return the current screen resolution.

XBIOS 5 - Set Screen Address & Mode, _setScreen

Parameters:	word: mode, long: physical address, long: logical address
Result:	None
Stack:	12

This lets you change the screen resolution and addresses. If any parameter is specified as -1 the it is left alone. Changing the screen mode will clear the screen.

?

XBIOS $26 - Call Supervisor Routine, supexec

Paramcters:	long: address of routine
Result:	None
Stack:	6

This will call the given routine in supervisor mode. The routine should not make any BIOS, XBIOS or GEMDOS calls.

GEM Libraries

GEM itself consists of two components; the VDI and the AES.

The GEM VDI (for Virtual Device Interface) is the main part of the operating system that draws graphics and text on the screen.

The GEM AES (for Application Environment Services) is the part of the operating system that provides the user-interface facilities of GEM such as windows, menus and dialog boxes.

This section is intended to give details of the supplied library files and calling conventions used. It does not attempt to describe either the VDI or the AES in great detail - the books in the Bibliography should be referred to for this. However, details are given of information that we feel is badly documented or not documented at all.

GEM AES Library

The calling sequence to the AES is based on various arrays of words and longwords. These arrays are defined using DS directives and are:

`control`	words
`int_in`	words
`addr_in`	longwords
`int_out`	words
`addr_out`	words
`aes_params`	longwords
`global`	words

For example the C program segment

```
        val=int_out(2)+int_out(3)
```

could be converted into this assembly language:

```
        move.w    int_out+4,d0
        add.w     int_out+6,d0
```

Note the way that the array index is doubled before adding to the start of the array, as it is an array of words. For an array of longs the index should be quadrupled.

A macro file, called `GEMMACRO.S` should be used which defines various macros and, if generating executable code, the file `AESLIB.S` should be included at the end of assembly.

The macros take a varying number of parameters and place them in the required places in the AES arrays, before making a call to the general AES routine. If passing a constant to a macro be sure to precede it with a # sign. for example passing the parameters `3,myptr` to a macro could generate the code

```
        move.w   3,int_in
        move.l   myptr,addr_in
```

The first line will cause a run-time error, the parameter should have been `#3`. There are a few AES macros which do not take all the required parameters - additional information may have to be placed in other arrays. On return from an AES macro D0.W (and the flags) reflect the contents of the array int_out(0), normally useful. Various return values can often be found in the int_out array.

The following descriptions assume all parameters to be word sized, unless shown with a .L suffix, denoting a longword parameter.

Application Library

appl_init

Should be called at the start of any AES program.

appl_read *id,length,buffer.L*

appl_write *id,length,buffer.L*

appl_find *name.L*

Find a named program, normally a desk accessory

appl_tplay *memory.L,number,scale*

appl_trecord *memory.L,count*

appl_exit

Should be just before an AES program terminates. It sends AC_CLOSE type messages to all desk accessories.

Event Library

evnt_keybd

evnt_button *clicks,mask,state*

The return value is the number of times the button entered the desired state. Array elements 1-4 of int_out contain the X co-ordinate, the Y co-ordinate, the button state and the keyboard state at the time of the event in that order.

evnt_mouse *flags,x,y,w,h*

The return values are as described for the previous call.

evnt_mesag *buffer.L*

evnt_timer *count.L*

evnt_multi *flags,clicks,mask,m1flags,m1x,m1y,m1w,m1h,*
& *m2flags,m2x,m2y,m2w,m2h,count.L*

All parameters except the first are optional, specifying a null parameter means nothing is placed in the relevant element of int_in. It is shown above with the syntax of a multi-line macro call but this is not obligatory. The int_out array contains which event, mouse X, mouse Y, button, keyboard state, keyboard code and button value, respectively.

evnt_dclick *new,getset*

Menu Library

menu_bar *tree.L,show*

menu_icheck *tree.L,item,check*

menu_ienable *tree.L,item,enable*

menu_tnormal *tree.L,title,normal*

menu_text *tree.L,item.text.L*

menu_register *id,string*

Note Normally a menu tree is generated by a resource editor though they can be constructed, with a great deal of care, by hand. Another alternative is to use the MENU2ASM compiler, detailed later in this section.

Object Library

Object trees are normally constructed with a resource editor, though they can be constructed by hand if required. Dialog boxes are the easiest type of object tree to construct by hand and menus the most difficult.

objc_add *tree.L,parent,child*

objc_delete *tree.L,object*

objc_draw *tree.L,startob,depth,x,y,w,h*

objc_find *tree.L,startob,depth,x,y*

objc_offset *tree.L,object*

Elements 1 and 2 of int_out contain the returned X and Y co-ordinates.

objc_order *tree.L,object,newpos*

objc_edit *tree.L,object,char,idx,kind*

int_out(1) contains the new idx.

objc_change *tree.L,object,x,y,w,h,new,redraw*

Form Library

form_do *tree.L,startob*

Never pass startob as -1 as often documented, use 0 instead.

form_dial *flag,x1,y1,w1,h1,x2,y2,w2,h2*

form_alert *button,string.L*

form_error *errnum*

Error numbers should be positive and less than 64.

form_center *tree.L*

Graphics Library

graf_rubberbox *x,y,w,h*

int_out(1) contains the finish width, int_out(2) the height.

graf_dragbox *w,h,x,y,bx,by,bw,bh*

int_out(1) contains the finish X co-ordinate, int_out(2) the Y.

graf_movebox *w,h,x,y,dx,dy*

graf_growbox *x,y,w,h,fx,fy,fw,fh*

graf_shrinkbox *x,y,w,h,sx,sy,sw,sh*

graf_watchbox *tree.L,object,instate,outstate*

graf_slidebox *tree.L,parent,obj,vh*

graf_handle

The int_out array will contain the VDI handle, character cell width, then height, system font width, then height.

graf_mouse *number,address.L*

The address parameter is optional, only required if defining you own shape.

graf_mkstate

The int_out array will contain a reserved value, mouse X and Y position, mouse button state and keyboard state.

Scrap Library

scrp_read *buffer.L*

scrp_write *buffer.L*

File Selector Library

fsel_input *path.L,filename.L*

The path parameter should point to a buffer containing the null-terminated path, such as `A:\*.S`, and the new path will be returned in it, so be sure it is large enough. The filename buffer should be 13 bytes, with a maximum of 12 used for the filename, for example `TEST.S`. If D0.W is non-zero on return then it means there was not enough free memory to invoke the selector, else int_out(1) will contain 0 if Cancelled.

Window Library

wind_create *kind,x,y,w,h*

wind_open *handle,x,y,w,h*

wind_close *handle*

wind_delete *handle*

wind_get*handle,field*

wind_set*handle,field*

wind_find *x,y*

wind_update *begend*

wind_calc *type,kind,inx,iny,inw,inh*

Resource Library

rsrc_load *filename.L*

rsrc_free

rsrc_gaddr *type,index*

The result address may be found in addr_out.

rsrc_saddr *type,index,saddr.L*

rsrc_obfix *tree.L,object*

shel_read *command.L,shell.L*

shel_write *doex,sgr,scr,cmd.L,shell.L*

We have never managed to get this call to work reliably.

shel_find *buffer.L*

The buffer should be a minimum of 80 bytes.

shel_envrn *value.L,string.L*

Debugging AES Calls

Unlike the calls to the VDI, calls to the AES are not immediately obvious when viewed from MonST as they are of the form

```
        moveq     #??,d0            AES function number
        bsr       CALL_AES
```

As an aid to decoding these, here is a table listing all the AES calls and their hex function numbers:

A	appl_init	B	appl_read	C	appl_write
D	appl_find	E	appl_tplay	F	appl_trecord
13	appl_exit	14	evnt_keybd	15	evnt_button
16	evnt_mouse	17	evnt_mesag	18	evnt_timer
19	evnt_multi	1A	evnt_dclick	1E	menu_bar
1F	menu_icheck	20	menu_ienable	21	menu_tnormal
22	menu_text	23	menu_register	28	objc_add
29	objc_delete	2A	objc_draw	2B	objc_find
2C	objc_offset	2D	objc_order	2E	objc_edit
2F	objc_change	32	form_do	33	form_dial
34	form_alert	35	form_error	36	form_center
46	graf_rubberbox	47	graf_dragbox	48	graf_movebox
49	graf_growbox	4A	graf_shrinkbox	4B	graf_watchbox
4C	graf_slidebox	4D	graf_handle	4E	graf_mouse
4F	graf_mkstate	50	scrp_read	51	scrp_write
5A	fsel_input	64	wind_create	65	wind_open
66	wind_close	67	wind_delete	68	wind_get
69	wind_set	6A	wind_find	6B	wind_update
6C	wind_calc	6E	rsrc_load	6F	rsrc_free
70	rsrc_gaddr	71	rsrc_saddr	72	rsrc_obfix
78	shel_read	79	shel_write	7A	shel_find
7B	shel_envrn				

The calling sequence itself to the VDI is, like the AES, based on various arrays of words and longwords. These arrays are defined using DS directives and are:

`contrl`	words
`intin`	words
`ptsin`	words
`intout`	words
`ptsout`	words
`vdi_params`	longwords

All (but one) VDI calls require a VDI handle, which by tradition is a parameter to every call. However, the majority of programs only use one handle, to a virtual workstation (the screen), so the supplied VDI libraries use a word called *current_handle* as the handle to pass on to the VDI itself. This saves an appreciable amount of code and is the same way the **HiSoft BASIC** libraries work. As the source to the library is supplied you could change this, if required.

The macro file `GEMMACRO.S` should be used which defines various macros and, if generating executable code, the file `VDILIB.S` should be included at the end of assembly.

The macros take a varying number of parameters and place them in the required places in the VDI arrays, before making a call to a VDI library routine. The warning about # signs in parameters described previously applies to the VDI too. There are a number of VDI macros which do not take all the required parameters - additional information may have to be placed in other arrays. On return, various return values can often be found in the *intout* and *ptsout* arrays.

The following descriptions assume all parameters to be word sized, unless shown with a .L suffix, denoting a longword parameter.

Control Functions

v_opnwk — Open Workstation

This should not be used unless GDOS is installed. The intin array should be suitably initialised, current_handle will be set to the result of this call.

v_clswk — Close Workstation

v_opnvwk — Open Virtual workstation

This uses current_handle to open another workstation and sets current_handle to the result. intin is normally filled with 10 words of 1 and one word of 2 (denoting RC co-ordinates).

v_clsvwk — Close Virtual Workstation

v_clrwrk — Clear Workstation

v_updwk — Update Workstation

vst_load_fonts — Load Fonts

Do not attempt this unless GDOS is loaded.

vst_unload_fonts — Unload Fonts

Fonts *must* be unloaded before a workstation is closed.

vs_clip *flag,x1,y1,x2,y2* — Set Clipping Rectangle

Output Functions

v_pline *count* — Polyline

The input co-ordinates should be copied to intin before the call.

v_pmarker *count* — Polymarker

The input co-ordinates should be copied to intin before the call.

v_gtext *x,y,string.L* — Text

The string should be in the from of null-terminated bytes.

v_fillarea *count*	Filled Area

The input co-ordinates should be copied to intin before the call.

v_contourfill *x,y,index*	Contour Fill
vr_recfl *x1,y1,x2,y2*	Fill Rectangle
v_bar *x1,y1,x2,y2*	Bar
v_arc *x,y,radius,start,end*	Arc
v_pieslice *x,y,radius,start,end*	Pie
v_circle *x,y,radius*	Circle
v_ellarc *x,y,xradius,yradius,start,end*	Elliptical Arc
v_ellpie *x,y,xradius,yradius,start,end*	Elliptical Pie
v_ellipse *x,y,xradius,yradius*	Ellipse
v_rbox*x1,y1,x2,y2*	Rounded Rectangle
v_rfbox *x1,y1,x2,y2*	Filled Rounded Rectangle
v_justified *x,y,string.L,length,ws,cs*	Justified Graphics Text

The string should be null-terminated.

Attribute Functions

vswr_mode *mode*	Set Writing Mode
vs_color *index,red,green,blue*	Set Colour Representation
vsl_type *style*	Set Polyline Line Type
vsl_udsty *pattern*	Set User Defined Line Style Pattern
vsl_width *width*	Set Polyline Line Width
vsl_color *index*	Set Polyline Colour Index
vsl_ends *begin,end*	Set Polyline End Styles
vsm_type *symbol*	Set Polymarker Type

vsm_height *height*	Set Polymarker Height
vsm_color *index*	Set Polymarker Colour Index
vst_height *height*	Set Character Height, Absolute Mode

The ptsout array will contain the selected size.

vst_point *point*	Set Character Height, Points Mode

The ptsout array will contain the selected size.

vst_rotation *angle*	Set Character Baseline Vector
vst_font *font*	Set Text Face
vst_color *index*	Set Graphic Text Colour Index
vst_effects *effect*	Set Graphic Text Special Effects
vst_alignment *horizontal,vertical*	Set Graphic Text Alignment
vsf_interior *style*	Set Fill Interior Style
vsf_style *index*	Set Fill Style Index
vsf_color *index*	Set Fill Colour Index
vsf_perimeter *vis*	Set Fill Perimeter Visibility
vsf_updat	Set User Defined Fill Pattern

The intin array should be filled with the pattern and contrl(3) set suitably.

Raster Operations

vro_cpyfm *mode,source.L,dest.L*	Copy Raster, Opaque

This is the general *blit* call, most often used for scrolling the screen. The source and destination parameters should point to a memory form definition block (MFDB) which describes the format of the memory to blit. An MFDB consists of ten words:

0	high word of address
2	low word of address
4	width in pixels

6	height in pixels
8	width in words
10	form flag, normally 1
12	number of planes
14-18	reserved, set to 0

The address in the first two words is normally either the screen address or the address of a buffer being used for the blit. The width and height fields should be those suitable for the screen size and the number of planes can be found from a vq_extnd 1 call in intout(4). When scrolling the screen the source and destination parameters may point to the same MFDB.

The source and destination rectangles should be placed in the ptsin array, each in the form x1,y1,x2,y2. A mode of 3 means replace.

vrt_cpyfm *mode,source.L,dest.L,i1,i2*	Copy Raster, Transparent
vr_trnfm *source.L,destination.L*	Transform Form
v_get_pixel *x,y*	Get Pixel

Input Functions

vex_timv *newtimer*	Exchange Timer Interrupt Vector
v_show_c *reset*	Show Cursor
v_hide_c	Hide Cursor
vq_mouse	Sample Mouse Button State
vex_butv *newxbut*	Exchange Button Change Vector
vex_motv *newmotv*	Exchange Mouse Movement Vector
vex_curv *newcursor*	Exchange Cursor Change Vector
vq_key_s	Sample Keyboard State Information

Inquire Functions

vq_extnd *flag*	Extended Inquire
vq_color *index,flag*	Inquire Colour Representation

vql_attributes	Inquire Polyline Attributes
vqm_attributes	Inquire Polymarker Attributes
vqf_attributes	Inquire Fill Area Attributes
vqt_attributes	Inquire Graphic Text Attributes
vqt_extent *string.L*	Inquire Text Extent

The string should be null-terminated, the results will be found in ptsout.

vqt_width *char*	Inquire Character Cell Width
vqt_name *number*	Inquire Face Name & Index
vqt_fontinfo	Inquire Current Face Information

AES & VDI Program Skeleton

The general structure of a GEM-type program is as follows:

```
     shrink memory call
     call appl_init
     set current_handle to the result from graf_handle
     open a virtual workstation using this handle
     open a window, perhaps
main wait for events & act on them as required
quit close any window
     close virtual workstation
     call appl_exit
     finally p_term
```

Desk Accessories

A desk accessory is an executable file with the extension .ACC loaded during AES initialisation. We have never seen any official documentation on desk accessories, and the following information has been learnt the hard way, mainly when writing our **Saved!** program.

The first thing to be wary of is that it is *not* a normal GEMDOS program. When it starts up all registers *including A7* are 0, with the exception of A0 which points to the basepage. An accessory must include all the memory it requires within itself, the BSS segment being a good place. An accessory must *not* do a GEMDOS shrink call or attempt to terminate.

The main loop of an accessory is like any other AES program, consisting of an event loop, but note that most documentation details incorrect message numbers - AC_OPEN is really 40 and AC_CLOSE is 41.

Other programmers have reported problems using the VDI from within an accessory. The recommended method is to open a virtual workstation only when you have to (i.e. before creating a window) and always close it (when you close your window or, failing that, when receiving an AC_CLOSE message. The example accessory supplied, like our **Saved!** program, does not use the VDI at all - paranoia rules!

If your accessory responds to timer events ensure that no GEMDOS calls (Trap #1s) are made unless your window is the front one, otherwise time bombs will be set and a crash is highly likely.

The file DESKACC.S contains the source to an example accessory, which simply displays the system free memory in an alert box. It has a label called RUNNER which can be set to 1 to produce a stand-alone application instead of an accessory. This can be invaluable during program development as you can symbolically debug a stand-alone program, while an accessory has to be debugged using AMONST without the benefit of symbols.

Linking with AES & VDI Libraries

The supplied macro file `GEMMACRO.S` is designed to be used in executable or linkable programs. The files `AESLIB.S` and `VDILIB.S` contain the actual code and should be included at the end of programs when generating executable code, but if generating linkable code they should not. If you look at `GEMTEST.S` you can see how a conditional is used to make this automatic.

When developing a program using these libraries we recommend executable code as it greatly reduces development time. However the file size can be reduced by using the selective library feature of the linker and using the `GEMLIB.BIN` file. For example, if `GEMTEST` is linked to GST-linkable code, producing `GEMTEST.BIN`, it can be linked with this library by passing LinkST the command line

```
gemtest -wgemlib
```

The `GEMLIB.LNK` control file will do the rest. If you want to reduce your program to the absolute minimum then you can change the libraries as you require, which is why we supply the source code.

Menu Compiler

For those who wish to use menus without using a resource editor we supply the program `MENU2ASM.TTP` which converts a menu definition file into assembly language source statement for inclusion in your program. We use this method ourselves in the GenST editor.

The menu specification should be created in a text file with the extension `.MDF` and an example follows:

```
[  Desk |  About Program ]
[ File | New \ Load \(--------\ Quit ]
[ Search | Find ]
```

and so on. Line breaks are ignored. Each menu title and its items are enclosed in square brackets `[` and `]`. There is a vertical bar (`|`) after each title and the individual items separated by back-slashes (`\`). For grey items precede the text with an open parentheses `(`. The first menu is always the desk title (normally `Desk`); the currently loaded desk accessories will be added by the AES. (It is no coincidence that this is the same syntax as that accepted by our BASIC compilers).

We recommend that you precede each menu item with two spaces and have at least one space after the item. Menu titles should have one space before and after them.

To compile a file double-click on `MENU2ASM.TTP` and enter the filename, without an extension. It will produce a file with an extension of `.MNU` which may be included in your program.

The file `MENUTEST.MDF` contains an example definition of a menu and `MENUTEST.S` the source code to a program illustrating its use, as well as showing other AES features.

Old GenST AES & VDI Libraries

The folder `OLDGEN` contains updated versions of the source files supplied with version 1 of **DevpacST**. These use different calling conventions and are supplied for users who have upgraded.

VT52 Screen Codes

When writing to the screen via the BDOS or BIOS calls, the screen driver emulates VT52 protocols. The control codes are sent via *escape* sequences, which means an escape character is sent (27 decimal, or $1B) followed by one or more other characters.

`ESC A`	cursor up; no effect if at the top line
`ESC B`	cursor down; no effect if at the bottom line
`ESC C`	cursor right; no effect if on the right hand side
`ESC D`	cursor left; no effect if on left hand side
`ESC E`	clear screen and home cursor
`ESC H`	home cursor
`ESC I`	move cursor up one line; if at top scrolls the screen down a line
`ESC J`	erase to end of screen, from the cursor position onwards
`ESK K`	clear to end of line
`ESC L`	insert a line by moving all following lines down. Cursor is positioned at start of the new line
`ESC M`	delete a line by moving all following lines up
`ESC Y`	position cursor; should be followed by two characters, the first being the Y position, the second the X. Row and column numbering starts at (32, 32) which is the top left
`ESC b`	foreground colour; should be followed by a character to determine the colour, of which the four lowest bits are used
`ESC c`	background colour; similar to above
`ESC d`	erase from beginning of display to the cursor position
`ESC e`	enable cursor
`ESC f`	disable cursor
`ESC j`	save the current cursor position
`ESC k`	restore a cursor position saved using `ESC j`
`ESC l`	erase a line and put cursor at start of line
`ESC o`	erase from start of line to cursor position
`ESC p`	inverse video on
`ESC q`	inverse video off
`ESC v`	wrap around at end of line on
`ESC w`	wrap around at end of line off

Appendix E Converting from other Assemblers

Most 68000 assemblers for the ST follow, to one degree or another, the Motorola standard. While the instructions themselves are thankfully standard, the syntax rules for labels, comments and directives can, and do, vary. This Appendix covers the changes most likely to be made when converting programs from another assembler, whether they are your old source files or a program listed in a magazine. It does not attempt to detail the differences in user interfaces or options between the different assemblers.

Atari MADMAC

GenST does not require colons after labels or comments to be delimited with semi-colons, but it does not allow instructions or directives to start in the label field.

The syntax and rules for local labels are the same, though $ and ? are not valid in GenST symbols. The use of \ in quoted strings may have to be changed, and some arithmetic operators and priorities are different.

MADMAC allows directives to start with dot, if these are removed most directives are the same as GenST. Those that differ, and their GenST equivalents, are:

BSS=SECTION BSS, DATA=SECTION DATA, TEXT=SECTION TEXT, ABS=OFFSET, ELSE=ELSEIF, ENDIF=ENDC, EXITM=MEXIT, GLOBL and EXTERN=XREF or XDEF, EJECT=PAGE, TITLE=TTL, NLIST=NOLIST.

INIT can be converted to DC or DCB statements and CARGS can be replaced with suitable RS directives.

MADMACs macro syntax is unique and its named parameters will need conversion, equivalents for its parameters are \~=\@ and \#=NARG, \? can be emulated using IFC or IFNC. The 6502 options of MADMAC are not supported.

GST-ASM

GST labels are significant only to the first 8 characters and are case insensitive so OPT C8- may be required. Its rules for expression evaluation are very similar though $ is not allowed within a GenST symbol.

Most directives are the same, those requiring name changes are PAGEWID=LLEN and PAGELEN=PLEN. Macro definitions will require conversion as will GSTs unique form of local symbols.

Built-in functions and structure statements are not supported.

MCC Assembler

Very few changes are required, only the syntax for local labels and add .L to XREF directives of absolutes.

K-Seka

Colons are not required after labels in GenST though instructions or directives that start in the label field will need a tab added before them. Several Seka directives default to Byte instead of Word sizes for some reason. Equivalent directives names are:

D=DC, BLK=DS, CODE=SECTION CODE, DATA=SECTION DATA,, IF=IFNE, ELSE=ELSEIF, ENDIF=ENDC.

Macro syntax requires ?s to be changed to \s, except ?0 which should be replaced with \@.

Fast ASM

The syntax of Fast ASM was designed around GenST 1.2 so few changes are required. Tokenised source files will need conversion to ASCII (using the Clipboard) before attempting to load them into the GenST editor. The main change involves comment delimiters - Fast ASM lines starting with \ should be changed to start with * or ; - \s used after instructions will not require any changes.

The floating point facilities in Fast ASM, left over from its BASIC interpreter origins, are not supported in GenST.

Appendix F
Bibliography

This bibliography contains our suggestions for further reading on the subject of the 68000, 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.

68000 Programming

M68000 Programmer's Reference Manual
Published by Prentice-Hall

The definitive guide to the instruction set produce by Motorola. The supplied Pocket Guide is a subset of this book. Be sure to get the latest version - at the time of writing the Fifth Edition is the latest.

68000 Assembly Language Programming by Kane, Hawkins & Leventhal, published by Osborne/McGraw-Hill

This is large (and expensive) but good, containing lots of examples. Be sure to get the second edition. Not for complete beginners to microprocessors.

68000 Tricks and Traps by Mike Morton
BYTE magazine, September 1986 issue

By far the best article on 68000 programming we have ever seen. We wish there was a book like this.

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 and lack of 68000 details. 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 Atari ST internals. 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.

M68000 Cross Macro Assembler Reference Manual
Published by Motorola (M68KXASM)

The official definition of 68000 assembly-language syntax on which GenST is based.

M68000 Resident Structured Assembler Reference Manual
Published by Motorola (M68KMASM)

This details the more advance aspects of the Motorola standard including extended macros and 68010/20/881 processors.

Appendix G Technical Support & Upgrades

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 three 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 more old fashioned methods; it's cheaper for everyone.

Whichever method you use please *always quote your serial number* (from your master disk) and the version number of the program. We reserve the right to refuse technical support if you do not supply this information.

For bug reports, please run the `CHECKST.PRG` program supplied and quote the information given by it, as well as details of any desk accessories and auto-folder programs in use. 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, DevpacST 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.

DevpacST Developer Version

For those that require maximum power from their 68000 assembler we have available the Developer version of DevpacST. Features over and above this version include:

GDOS is supplied together with documentation, sample program and calling sequences; Motorola **S record** hex output and multiple-ORG statements for users cross-developing; **Amiga** executable & linkable file formats; 68010/20/30/881/882 instructions; **Dual machine** debugging; Detailed notes on GST & DRI **file formats** including special dump programs for both formats, source (in **HiSoft BASIC**) included; **Free upgrades** for a year, despatched automatically.

DevpacST Developer is available as an upgrade.

Appendix H
Revision history

Product History

DevpacST 0.50 was first released in late 1985, but with various restrictions to do with the editor and the lack of linkable code. The next major version was 0.91 which was much improved in many respects, followed by 0.99f, the last version which didn't produce linkable code. Version 1.0 was released in April 1986, and underwent a few minor changes before the release of version 1.22 in June 1986, the first version supplied in a ring-bound manual. After various small revisions the greatly improved version 2.0 was released in April 1988.

Development Technique

DevpacST was originally based on **DevpacQL**, our Assembler Development suite for the Sinclair QL. Both GenST and MonST were written in assembler on the QL then uploaded via the serial port into the ST, and LinkST was written using Lattice C on the ST. Development moved across to the ST around version 1.24 which had minor changes made, reaching version 1.26. Special internal versions were written to experiment with things like linkable code and extended debug and reached version 1.57 before both GenST and MonST were completely re-written for version 2. The editor was extensively altered, originally for **HiSoft BASIC**, then **Power BASIC**, then GenST. The editor is written entirely in assembly language.

Summary of Version 2 Improvements

This section is intended as a quick guide to the main additional features for users who have upgraded from version 1.2 of **DevpacST**. It gives an overview of the new features, for further details you should consult the relevant sections of the new manual.

The Editor

This has been greatly enhanced, with an overall improvement in display speed being the most obvious. The editor supports lines up to 240 characters in length, sideways scrolling as required. It also works in low-resolution. There is now a horizontal scroll bar and the workings of the vertical scroll bar is now more "standard". By default the numeric pad is configured as an IBM-style cursor cluster and the text editor workspace size can now be changed from within the editor. This, combined with the saving of preferences, means the installation program in version 1 is redundant. Other programs can be run from within the editor using the Run Other facility.

Block Delete has changed to `Shift-F5` from `Shift-F3` (as fast left-handed typists can generate the `Shift-F3` scan-code in ordinary typing) and now remembers the block, if possible, allowing it to be pasted. A deleted line may be recalled as many times as required. A block may be copied to the block buffer, and marked blocks are now show on screen. Our **Saved!** desk accessory may be invoked at the press of a key, there is a keyboard shortcut for Save, and the editor will now run in low-resolution.

The Assembler

Symbols are now significant to the first 127 characters and local labels are supported. The INCBIN directive takes a straight binary file and copies it into the output file, particularly useful for screen data. Speed - include files are read only once, memory permitting, and the binary file is buffered for as long as possible. The absolute maximum speed has over doubled to 75,000 lines per minute though for real programs 35,000 lines is the norm. The output file is also extensively buffered if possible producing spectacular improvements during floppy-to-floppy assemblies in particular.

General improvements in symbol table searching and hashing have also increased overall speed. Extended Debug - a HiSoft extended version of the DR symbol table, allowing debugging with up to 22 character significance. Macro calls and Includes may be nested as deeply as memory allows, IFs can be 64k levels deep. TEXT, DATA & BSS segments are properly supported when generating executable code.

There is much greater control over output filenames. Multiple Modules & Sections - the GST linker format is more fully supported allowing multiple modules and multiple sections. Externals may be used in expressions, with each other if required. DRI linkable code can now be generated.

Optimising can now be performed by the assembler on things such as short branches. Macros now support up to 36 parameters multi-line calls and numeric substitution. The macro buffer is now dynamic, the free space in the editor workspace is no longer used for this. REPEAT loops are now allowed and the expression evaluator now includes comparisons. The REG directive allows symbolic register lists. There are a considerable number of extensions to the OPT directive. Octal numbers are supported. Registers may also be called R0 through R15. A stand-alone assembler is supplied for those who use alternate editors, CLIs or batch files.

Compatibility Issues

Most source files should assemble with little or no changes. The differences to be careful of are: BSS sections - neither RSBSS or DSBSS are supported, the code should be converted to switch to SECTION BSS then use DS statements. Symbols now default to case-dependent and are significant to the first 127 characters. OPT N for narrow listings has been superseded with the FORMAT directive allowing much greater flexibility and general listing control has been improved. The ORG statement has changed which will effect any programs that use it.

HiSoft BASIC users - if creating libraries please note that GenST 2 output is not accepted by BUILDLIBs prior to version 1.4.

BRA.W wasn't accepted by version 1, forcing the use of BRA.L - this is, strictly speaking, a 68020 instruction so now generates a warning. BRA.W (and BRA.B) are now accepted. Various minor changes have been made to the parsing of instructions allowing a greater degree of flexibility. If you used (expression)\W (denoting short-word addressing) within a macro this will be be ignored as \W and \L refer to macro parameters and will probably be replaced with nulls. Use instead (expression).W, which is the Motorola standard.

The use of \W or \L after register equates used as index registers is no longer required, for example if `buf` is a register equate then `move.b d0,0(a6,buf.l)` is now allowed. Register equates are now allowed in MOVEMs. The priority of the equality operator (=) has been changed.

The GEM example program has been changed to use a true BSS section and to fix a bug preventing correct operation under GDOS - this and its include file can be found in the `OLDGEM` folder. Many new example files are supplied including a desk accessory and completely new AES and VDI libraries.

The assembler now reports syntactic errors on pass 1 and will not start pass 2 if errors have been found. It now uses GEMDOS character output routines so can be paused with `Ctrl-S`, resumed with `Ctrl-Q` and aborted with `Ctrl-C`.

There are several new directives which could potentially clash with macro names in GenST 1 source files. These are: COMMENT, DCB, ELSEIF, ENDR, FORMAT, IIF, INCBIN, OFFSET, OUTPUT, REG, REPT, RSSET, SUBTTL.

There are three reserved symbols which could theoretically clash with your own, all starting with two underlines: __LK, __RS and __G2.

Debugger

MonST supports a great number of new features including multiple windows and, as a result, has a changed user interface. It is strongly recommended that you read the **Reference** section of the **Chapter 4** before trying any serious work with the new version. The main new features are: Multi-window display; Timed screen switching removing flicker; Full expression evaluator including indirection; Supports up to 22 significant characters in symbols; Multi-resolution allows you to debug a low-res program in medium res (or vice versa); Allows the viewing of source files within the debugger; Disassemble to printer with automatic label generation or to a disk file in GenST format; Conditional breakpoints; History Buffer; Interrupt running programs.

Both GEM and TOS versions of the debugger are now the same except for the file extension and only one auto-resident version needs to be supplied.

Integration

Probably the greatest improvement to the package as a whole is the integration between its various parts. The assembler is available at the press of a key from the editor, as it was, but so is the debugger. The assembler can assemble directly into memory, then the code can be run from the editor without any disk accesses. If required debugging information can also be included in assembled-to-memory programs so they can be debugged at the press of a key, directly from the editor. A program assembled to memory is a true GEMDOS task so no code changes are required.

Assembly warnings and errors are remembered by the editor and can be stepped through by pressing `Alt-J`. Errors are no longer lost when the number of lines is changed, though they are not re-calculated. After an assembly which had an error the editor will automatically place the cursor on the line of the first error.

Linker

This now supports the HiSoft Extended Debug format and is faster than its predecessors. It also allows explicit section ordering and true BSS sections. Note that LinkST only supports the GST format - if you wish to link DRI format code you need to use the Atari `ALN` or the Digital Research `LINK68` linkers.

Notes