HiSoft DevpacST

Assembler/Editor/Debugger

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

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 inconvenience. 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

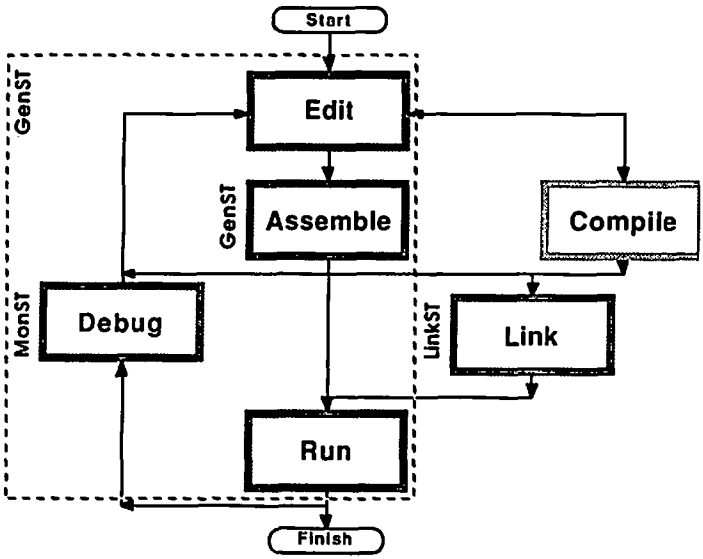
Enclosed with this manual is a registration card which you should fill in and return to us after reading the licence statement. Without it you will not be entitled to technical support or upgrades. Be sure to fill in all the details especially the serial number and version number. Also supplied is a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

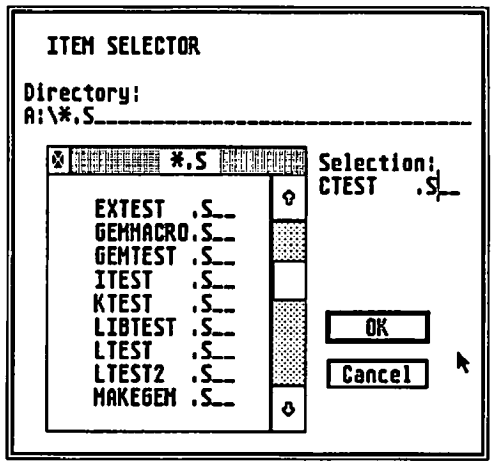


Figure 2.1 - the GEM File Selector

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.

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`.

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

It 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.

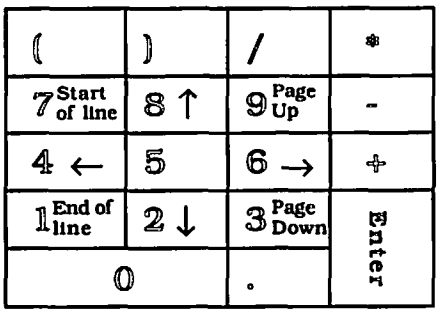


Figure 2.2 Numeric Keypad

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.

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.

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.

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.

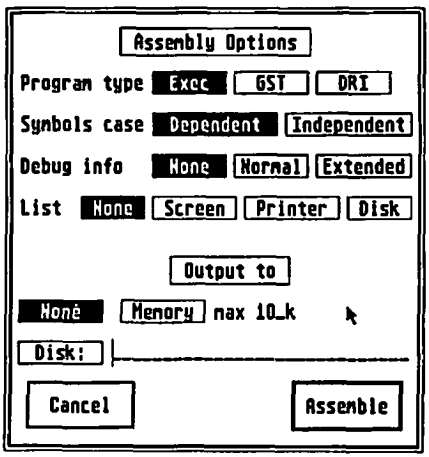


Figure 3.1 - the Assembly Options dialog box

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

Addressing Modes

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
```

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.

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
```