Note

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

Quitting MonST2C

Ctrl-C — Terminate

This will issue a terminate trap to the *current* GEMDOS task. If a program has been loaded from within MonST2C it will be terminated and the message Program Terminated appear in the lower window. Another program can then be loaded, if required.

If no program has been loaded into MonST2C it will itself terminate when this command is used.

Note

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

Loading & Saving

Ctrl-L — Load Executable Program

This will prompt for an executable filename then a command line and will attempt to load the file ready for execution. If MonST2C has already loaded a program it is not possible to load another until the former has terminated.

The file to be loaded must be an executable file - attempting to load a non-executable file will normally result in TOS error 66 and further attempts to load executable files will normally fail as GEMDOS does not de-allocate the memory it allocated before trying to load the errant file. If this occurs terminate MonST2C, re-execute it and use the Load Binary File command.

B — Load Binary File

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

S — Save Binary File

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

A — Load ASCII File

This powerful command allows an ASCII file, normally of source code, to be loaded and viewed within MonST2C. Window 4 will be created if required then set up as a source-code window. Memory for the source code is taken from GEMDOS so sufficient free memory must be available. It is recommended that source-code be loaded *before* an executable program to ensure enough memory.

If the file currently being debugged contains Lattice debug information, MonST2C will use the line number information corresponding to the new source file that has been loaded. Thus, loading the source file will change the effect of the # operator.

Low Res

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

Note

If an ASCII file is loaded *after* an executable program the memory used will be owned by the *program itself*, not MonST2C. When such a program terminates, any displayed source-code window will be closed. This is also the case when the source text is automatically loaded by MonST2C.

Executing Programs

Ctrl-R — Return to program / Run

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

Ctrl-Z — Single-Step

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

Ctrl-Y — Single-Step

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

Ctrl-T — Interpret an Instruction (Trace)

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

Ctrl-S — Skip an Instruction

Ctrl-S increments the PC register by the size of the current instruction thus causing it to be skipped. Use this instead of Ctrl-Z when you know that this instruction is going to do something it shouldn't or that you don't like.

R — Run (various)

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

Run G Go

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

Run S Slowly

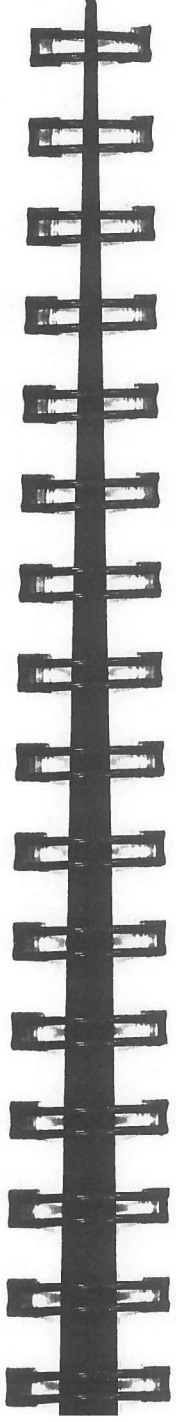
This will run the program at reduced speed, remembering every step in the history buffer.

Run I Instruction

This is similar to Run Slowly but allows a count to be entered, so that a particular number of instructions may be executed before MonST2C is entered.

Run U Until

You will be prompted for an expression which will be evaluated after every instruction. The program will then run, albeit at reduced speed, until the given expression evaluates to non-zero (true) when MonST2C will be entered. For example if single-stepping a DBF loop which used d6 in the ROM code you could say Run Until d6&ffff=ffff (waiting for the low word of d6 to be $FFFF) or, alternatively, PC=FC8B1A, or whatever.



Note

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

With all of these commands (except Run Go) you will then be asked Watch Y/N? If Y is selected then the MonST2C display will be shown after every instruction and you can watch registers and memory as they change, or interrupt execution by pressing both Shift keys simultaneously. If N is selected then execution will occur while showing your program's display and execution may be interrupted by pressing Shift-Alt-Help.

Note

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

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

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

Searching Memory

G — search memory (Get a sequence)

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

If you select B, W or L you will then be prompted to enter the sequence of numbers you wish to search for, each separated by commas. MonST2C is not normally fussy about word-alignment when searching, so it can find longs on odd boundaries, for example. If you wish to force a particular alignment, finish the list of items to search for with ',W'; for word boundaries or ',L' for longword boundaries.

If you select T you may search for any given text string, for which you will be prompted. You will also be asked whether you wish the search to be case sensitive; if you press Y then Test will match TEST or TeSt

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

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

N — find next

This can be used after the G command to find subsequent occurrences of the search data. With the B, W, L and T options you will always find at least one occurrence, which will be in the buffer within MonST2C that is used for storing the sequence. With the T option you may also find a copy in the system keyboard buffer. With these options, the Esc key is tested every 64k bytes and can be used to stop the search. With the I option, which is very much slower, the Esc key is tested every 2 bytes.

The search area of memory goes from 0 to the end of RAM, then via the system ROM area and cartridge area then back to 0. MonST2C will not search the cartridge area if the environment variable NOCARTRIDGE exists.

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

Searching Source-Code Windows

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

Miscellaneous

Ctrl-P — Preferences

This permits control over various options within MonST2C. The first three require Y/N answers, pressing Esc aborts and Return leaves them alone.

Screen Switching

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

Follow Traps

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

If you are interested in this sort of low-level hackery, you should consider purchasing DevpacST as it provides facilities from recovering from the after effects of interrupting the operating system code.

Relative Offsets

This option defaults to On and affects the disassembly of the address register indirect with offset addressing modes, i.e. xxx(An). With the option on, the current value of the given address register is added to the offset and then searched for in the symbol table. If found it is disassembled as symbol(An). This option is required to show the addresses of your global variables if they are accessed via an address register.

Ignore Case

This option defaults to Off. If it is set to On then if you enter fred in an expression the subsequent search will give the value of the first symbol that matches this, ignoring case, thus finding FRED, fred or Fred. This option is useful for lazy typists who use the same name with different casing.

Show Line Numbers in Source

MonST2C can either show line numbers in your source window in decimal (press D), hexadecimal (H) or not at all (press N). Using hexadecimal line numbers has the advantage that you can use them directly with the # line number operator. This if you can see that you want to execute your program until the line with number 001C then just type U (for run until) #1C. Remember how ever that A0 through A7 and D0 through D7 are register names and take priority over hexadecimal numbers. To enter line number A0 use #$A0.

Decimal line numbers are naturally more civilised but remember that you need to prefix any decimal number with \. If you want to find the address of line 28 decimal, use #\28 not #28.

Auto Load Source

Using the default settings, MonST2C will automatically load a C source file and run your program until the label _main, (i.e. the beginning of your function main), ready for you to set a breakpoint in the code. MonST2C loads the source file corresponding to the first module with debug information in the file that you are debugging. This would normally be your main program. You can disable this feature if you do not wish to load this source file or you wish to debug a program written in another language. Please see the sections at the end of this chapter concerning the use of MonST2C with multi-module programs.

Automatic Prefix Labels

Using the default setting MonST2C will try prefixing symbols by _ and @ if it cannot find a label, so that if you enter main and there is no label called main, the MonST2C will try _main or if this doesn't exist then it will try @main.

This facility is extremely useful since C functions normally have an _ added by the compiler. When using register passing @ is used as the prefix instead of _. Thus you can just use the C name without bothering about the prefix.

Should there be an assembly language name the same as a C name, say test and _test, then you will need to use the _ explicitly to get the C function rather than the assembly language one.

You can disable this option so that only exact matches of names are supported.

Symbols Option

This allows control over the use of symbols in expressions in MonST2C. It will firstly ask whether the case of symbols should be ignored, pressing Y will cause case independent searching to be used. It will then prompt for the maximum length of symbols, which is normally 22 but may be reduced to as low as 8.

Top Of RAM

This indicates to MonST2C which memory location should be considered the top of memory by the Search Memory (G) command. Normally you will not need to change this as it defaults to the system variable phys_top; but you may need to modify it if you are debugging software that lowers phys_top.

Save preferences

Reply Y to this command to save your current preferences to the file MONST2.INF in the current directory. When MonST2 loads it will read your current preferences from this file. MONST2.INF must be in the current directory when MonST2C is loaded.

I — Intelligent Copy

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

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

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

No checks at all are made on the validity of the move; copying to non-existent areas of memory is likely to crash MonST2C and corrupting system areas may well crash the machine.

L — List Labels

This opens up a large window and displays all loaded symbols. Any key displays the next page, pressing Esc aborts. The symbols will be displayed in the order they were found on disk.

W — Fill Memory With

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

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

The warning described under the I command about the lack of checks applies equally to this command.

M — Modify Address

Included for compatibility with MonST1, equivalent to Alt-A.

O — Show Other Bases

Included for compatibility with MonST1, equivalent to Alt-O.

D — Change Drive & Directory

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

Ctrl-Alt-Numeric Dot — Reset machine

Holding down Ctrl and Alt and then pressing the Dot (.) key on the numeric keypad will cause the machine to be reset. Great for Mega ST owners with 1.2 ROMs but without long arms!

Ctrl-E — Re-Install Exceptions

This command causes MonST2C to re-install the exception vectors; useful if you are debugging a high level language program whose runtime routines use the exceptions. Naturally, Lattice C 5 programs do not normally modify the exception vectors. This must be used *after* the user's program has modified the exceptions.

Command Summary

Window Commands

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

Screen Switching

V ..... View Other Screen
Ctrl-O ..... Other Screen Mode

Breakpoints

Alt-B ..... Set Breakpoint
Help ..... Show Help and Breakpoints
Ctrl-B ..... Set Breakpoint

U ..... Go Until
Ctrl-K ..... Kill Breakpoints
Ctrl-A ..... Set Breakpoint then Execute
Ctrl-D ..... GEMDOS Breakpoint

Loading and Saving

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

Executing Programs

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

Searching Memory

G ..... Search Memory (Get a sequence)
N ..... Find Next

Miscellaneous

Ctrl-Alt-Dot ..... Reset machine
Ctrl-C ..... Terminate
Ctrl-E ..... Re-install breakpoints
Ctrl-P ..... Preferences
D ..... Change Drive & Directory
H ..... Show History Buffer
I ..... Intelligent Copy
L ..... List Labels
M ..... Modify Address
O ..... Show Other Bases
W ..... Fill Memory With
Shift-Alt-Help ..... Interrupt Program

Debugging Stratagem

Hints & Tips

If you have interrupted a program using Shift-Alt-Help or by a Run Until command and have found yourself in the middle of the ROM, there is a way of returning to the exact point in your program which called the ROM. Firstly ensure the Follow Traps option is on, then do Run Until with an expression of sp=a7. This will re-enter MonST2C the moment user mode is restored which will be in your program.

When using Run Until knowing that it will take a quite a while for the condition to be satisfied, give MonST2C a hand by pre-computing as much of the expression as you can, for example

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

could be reduced to

```
a3>xxx
```

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

If you do use a label with Run Until then explicitly including any leading _ or @ will speed up the table search considerably.

Bug Hunting

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

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

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

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

```
{main_ptr}&1
```

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

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

Exceptions

MonST2C uses the 68000 processor exceptions to stop runaway programs and to single-step, so at this point it would be useful to explain them and what normally happens when they occur on an ST.

There are various types of exception that can occur, some deliberately, others accidentally. When one does occur the processor saves some information on the SSP, goes into supervisor mode and jumps to an exception handler. When MonST2C is active it re-directs some of these exceptions so it can take control when they occur. The various forms of exceptions, their usual results, and what happens when they occur with MonST2C active is shown in the following table:

Number	Exception	Usual effect	MonST2C active
2	bus error	bombs	trapped
3	address error	bombs	trapped
4	illegal instruction	bombs	trapped
5	zero divide	bombs	trapped
6	CHK instruction	bombs	trapped
7	TRAPV instruction	bombs	trapped
8	privilege violation	bombs	trapped
9	trace	bombs	used for single-stepping
10	line 1010 emulator	fast VDI interface	fast VDI interface
11	line 1111 emulator	internal TOS	internal TOS
32	TRAP #0	bombs	trapped
33	TRAP #1	GEMDOS call	GEMDOS call
34	TRAP #2	AES/VDI call	AES/VDI call
35-44	TRAP #3-#12	bombs	trapped
45	TRAP #13	XBIOS call	XBIOS call
46	TRAP #14	BIOS call	BIOS call
47	TRAP #15	bombs	trapped

Exceptions 2 to 8 are caused by a programmer error and are trapped by MonST2C.

Exception 9 can remotely be caused by programmer error and is used by MonST2C for single stepping.

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

The rest (i.e. the unused TRAP exceptions) are diverted into MonST2C, but can subsequently be re-defined to be exploited by programs if required.

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

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

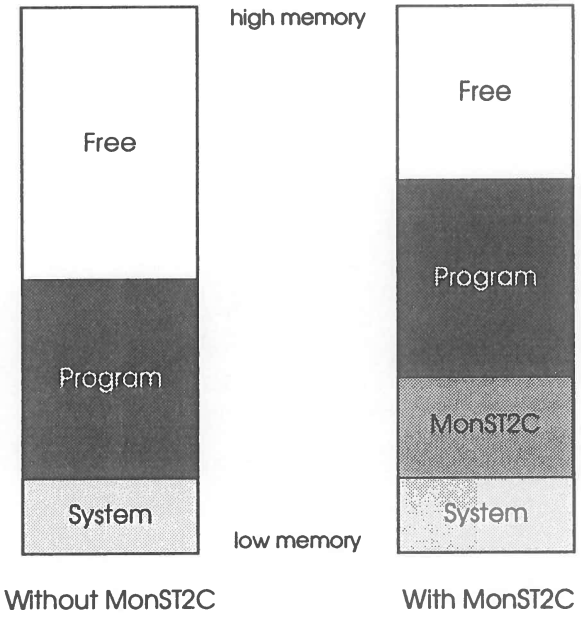
If the exception was caused by or resulted in important system variables being destroyed then the attempt may fail and the machine will not recover.

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

Memory Layout

The usual versions of MonST2C co-reside with programs being debugged; that is, they are loaded, ask for a filename, and load that file in together with any labels.

It is useful to examine the usual logical memory map both with and without MonST2C, shown below:



The actual code size of MonST2C is around 25k, but in addition it requires an additional 32k of workspace. This may seem large but it is required for the copy of the ST screen memory saved by MonST2C; this is a most useful feature of the debugger.

Exception Analysis

When an unexpected exception occurs, it can be useful to be able to work out where and why it occurred and, possibly, to resume execution. Often a quick curse and a look at your source code may get your program working quicker though!

Bus Error

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

Address Error

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

Illegal Instruction

If the PC is in very low memory, below around $30, it is probable that it was caused by a jump to location 0. If you use MonST2C to look here you will see a short branch together with, normally, various ORI instructions (really longword pointers) and eventually an illegal instruction.

Privilege Violation

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

Using MonST2C with other languages

A major feature of MonST2C is its ability to use symbols taken from the original program whilst debugging. MonST2C supports two formats for label information - the DRI standard, which allows up to 8 characters per symbol, and the HiSoft Extended Debug format, allowing up to 22 characters.

Most of HiSoft's language products support both formats (for example, DevpacST, HiSoft BASIC and FTL Modula-2) and many other vendors' compilers and linkers have an option to produce DRI-format debugging information.

The line number information format is, at the present time, specific to Lattice C and as such the line number operator, #, can only be used with programs compiled with one of the debugging options (-d1 to -d5).

Using MonST2C with multi-module programs

MonST2C will initially read the line number information and source file for the first module in the file that was compiled using -d. Thus, if you are only interested in debugging one module at a time then compile just this module with -d3: the appropriate source code will be loaded automatically.

If you wish to debug more than one file at once, then you can switch to another file by explicitly loading the appropriate source file using the A command.

For Devpac MonST2 Users

If you are used to the version of MonST that is part of Devpac ST version 2, here are the differences:

- Source line numbers can now be displayed in either hexadecimal or decimal. This is set using the Preferences command.
- Using the default settings, MonST2C will automatically run the program until the label _main and load the source file corresponding to the first debug information in the file.
- The operator # is used to give the address corresponding to a given C line number. To use this you need to use the compiler's -d option. The argument to # is a general MonST2C expression and so when using a number this should be in hexadecimal or prefixed by \ for decimal. Thus #10 and #\16 both give the address of line 16 of the program.
- The ASCII load command will change the action of the # operator if the appropriate debug information is available.
- MonST2C has no support for disassembly to disk. Dissassembly to the printer is only available via Alt-P.

ASM
The Assembler

The Lattice Macro Assembler supports the development of assembly language modules for use with C programs. Because the Lattice C Compiler generally produces very good machine code you seldom have to resort to assembly language programming. However, some intimate relations between hardware and software are best achieved in the assembly language environment. Also, assembly language is sometimes necessary when you want to get the best combination of code size and speed.

The assembler handles the complete set of Motorola 680x0 instruction mnemonics as well as an extensive set of assembler directives and a powerful macro facility. It can, therefore, be used to develop complete systems in assembly language. Nonetheless, it is provided primarily to supplement the C compiler and has not really been designed for large assembly language projects. For such tasks a full assembler package, such as DevpacST should be used giving more power for the assembly language programmer.

Basic Concepts

The assembler reads a source file and produces an object file in the Lattice object file format, along with an optional listing of the source and assembled code. The source file is assumed to have a .s extension and the object file is produced with a .o extension..

Source Format

Each assembly language source line has the following format:

```
label       operation operands  comment
```

White space (i.e. spaces and tabs) can appear before any field and must appear between the operation and operand.

The four fields of the source line are described below:

Label

The label field is optional. If it is present and is preceded by white space, it must be followed immediately by a colon. That is how the assembler determines that the field is a label and not an operation. If there is no white space before the label, then the colon may be omitted.

A label can normally be up to 63 characters long and can contain letters, digits, underscores, periods, at symbols (@) and dollar signs. It cannot start with a digit, and the case of letters *is* significant. For example, labels XYZ, xYZ, and XyZ are distinct.

Local labels are supported using the Motorola standard syntax of a decimal number followed by a dollar character. They may be used between two non-local labels and need only have unique names within that scope. Note that unlike GenST, starting a label with a period *does not* signify a local label.

Operation

The operation field contains the name of an instruction, assembly directive, or macro. This field may not begin a line; if no label is present, then the line must begin with white space. If a label is present but is not followed by a colon, then white space must separate the label and operation fields.

The case of this field is *not* significant. That is, operation MOVE is the same as move, this applies equally to macros.

Operands

The operands field contains zero or more expressions, depending on the particular operation. For some operations, the operands field is optional or never used. Expressions are composed of constants, variables, and operators.

A *constant* is a decimal, hexadecimal, octal, or binary number. The default number base is decimal, and the other bases are indicated by a prefix:

Number Representations

Number	Representation	Example
Decimal	a string of decimal digits	1234
Hexadecimal	$ followed by a string of hex digits	$89AB
Octal	@ followed by a string of octal digits	@743
Binary	% followed by zeros and ones	%10110111
ASCII Literal	Up to 4 ASCII characters within quotes	"AC9T"

A *variable* is a label name or a name defined via an assembler directive. The special variable, * (asterisk) can be used to signify the current program counter.

An *operator* is one of the following:

Order	Operator	Meaning
1	-	Unary minus
	~	Bitwise NOT
2	<<	Left shift
	>>	Right shift
3	&	Bitwise AND
	|	Bitwise OR
4	*	Multiply
	/	Divide
	%	Modulo
5	==	Equal to
	!=	Not equal to
	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
	+	Add
	-	Subtract
6	^	Bitwise Exclusive OR

The Order column indicates the order in which operators are processed. Operators of the same precedence are processed from left to right. For example, in the expression

```
ABC+DEF*-PDQ
```

the negation of PDQ is performed first, followed by the multiplication and then the addition, although this can be overridden by the use of parentheses as in,

```
(ABC+DEF)*-PDQ
```

Each expression represents a 32-bit value. An *absolute expression* is one that contains only constants (literal or equated), while a *relocatable expression* contains symbols whose value is determined during linking.

Comment

This field is any text appearing after an operation, associated operands and white space. A comment may also be specified after a label or on a blank line when prefixed with a semi-colon or asterisk.

Addressing modes

The addressing modes supported by the Lattice assembler are as follows:

Mode	Example
Dn	`add.w d1,d0`
An	`addq.w #1,a1`
(An)	`add.w (a1),d0`
(An)+	`add.w (a1)+,d0`
-(An)	`add.w -(a1),d0`
d16(An)	`add.w 10(a1),d0`
d8(An,Xn)	`add.w 10(a1,a2.l),d0`
bd(An,Xn)	`add.w $10000(a1,a2.l),d0` ('020 only)
([bd,An],Xn,od)	`add.w ([10,a1],a2.l,20),d0` ('020 only)
([bd,An,Xn],od)	`add.w ([10,a1,a2.l],20),d0` ('020 only)
(xxx).W	`add.w (100).w,d0`
(xxx).L	`add.l (100).l,d0`
#<data>	`add.l #100,d0`
d16(pc)	`add.w 10(pc),d0`
d8(pc,Xn)	`add.w 10(pc,a2.l),d0`
bd(pc,Xn)	`add.w $10000(pc,a2.l)` ('020 only)
([bd,pc],Xn,od)	`add.w ([10,pc],a2.l,20),d0` ('020 only)
([bd,pc,Xn],od)	`add.w ([10,pc,a2.l],20),d0` ('020 only)

where:

d8	8 bit number
d16	16 bit number
bd	32 bit byte displacement
od	32 bit outer displacement
An	Address register (a0-a7)
Dn	Data register (d0-d7)
Xn	Index register (d0-d7/a0-a7)

Note that all the operands of the addressing modes marked 68020 are optional.

Data for the 68881 floating point instructions may be specified using floating point notation, i.e.

```
...#2.1
...#2.1E+10
```

will be converted into the proper floating point formats according to the type of instruction. For example, in the following instruction:

```
fmove.s    #2.1,fp1
```

The 2.1 would be in single precision. Other sizes allowed are:

```
fmove.d    #2.1,fp1  ; double precision
fmove.x    #2.1,fp1  ; extended precision
```

Note that the packed data format is not converted for you. Also if you want to specify the bit pattern by hand you may use the following formats:

```
fmove.s    #$12345678,fp1                        ; 32 bit
fmove.d    #$123456781234568,fp1                 ; 64 bit
fmove.x    #$123456781234567812345678,fp1        ; 96 bit
```

You can also specify the constants in octal (i.e. `@123456712`) or binary (i.e. `%0110110100110101`).

Using the Assembler

The assembler can be run via the following command:

```
asm [>listfile] [options] filename
```

Optional fields are enclosed in brackets, and all fields are described below:

>listfile

Causes the listing and error message output of the assembler to be directed to the specified file.

options

Assembler options are specified as a minus sign followed by a single letter; in some cases, additional text may be appended. The letter may be in either upper or lower case. Each option must be specified separately, with a separate minus and letter. The options are:

-d This option has two uses. It activates the debugging mode (in the same way as the compiler -d1 option) or it defines symbols. When used to define symbols it may be used in the following ways.

-dsymbol

Causes symbol to be defined as if your source file had the statement:

```
symbol    EQU  1
```

-dsymbol=value

Causes symbol to be defined as if your source file had the statement:

```
symbol    EQU  value
```

-ipfx Specifies that INCLUDE files are to be searched for by prefixing the filename with the string pfx, unless the filename in the INCLUDE statement is already prefixed by a drive or directory specifier. Up to 4 different -I strings may be specified in the same command. No intervening blanks are permitted in the string following the -I. Note that if a directory name is to be specified as a prefix, a trailing backslash *must* be supplied.

When an unprefixed INCLUDE filename is encountered, the current directory is searched first; then file names are constructed and searched for, using prefixes specified in -I options, in the same left-to-right order as they were supplied on the command line.

-l(list) Causes a listing of the source file to be written to the standard output. The listing displays the appropriate program counter and code information alongside the assembly source. One or more of the following characters may be appended to the -l option, with the following effects:

- **i** List the source for text from INCLUDE files as well as the original source file.
- **m** List additional data generated for source lines which cannot be accommodated alongside the original source line (i.e. allows multiple listing lines for each source line).
- **x** List the expansion text for macros.

-m This option controls whether warnings are generated when 68020 code is encountered. The -m must be immediately followed by one of the letters from the following list:

- **0** Used for 68000 target. Provides warning flags if you attempt to use 68020 only instructions. This is the default case.
- **2** Used for 68020 target. Turns off the warnings supplied in the -m0 option.
- **3** Used for 68030 target.

-opfx Specifies that the output filename (the .o file). If a directory name is specified the output name is formed by prefixing the input filename (the .s file which is being assembled) with pfx. Any drive or directory prefixes originally attached to the input filename are discarded before the new prefix is added. No intervening blanks are permitted in the string following the -o. Note that if a directory name is to be specified as a prefix, a trailing backslash *must* be supplied.

-u This option automatically prefixes all external references with an underline (_). If references to C labels have already been prefixed with an underline, the option is not needed.

-w This option works like the option -dSHORTINT.

filename

Specifies the name of the source file to be assembled. This is the only required field on the command line. If the name does not have an extension .s is assumed. The object file will have the same name as the source file, except that the source file extension is replaced with .o.

For example, the following command causes the assembly language source file modn.s to be assembled, producing the object file modn.o. A listing of the source file, along with any error messages generated, will be written to the file modn.lst.

```
asm >modn.lst -l modn
```

Assembler Directives

The assembler handles all the 68000, 68020, and 68030 instructions using the standard Motorola syntax. Assembler directives are instructions to the assembler rather than instructions to be translated directly into object code.

Note that although the IDNT, PAGE, SPC and TTL directives are recognised, they are not supported and do not cause errors to be generated in order to provide compatibility with other assemblers. Also, as with instruction mnemonics, directives cannot begin in the first character of the source line.

CNOP offset,alignment

This directive aligns the program counter using the given byte alignment and offset. For example,

```
cnop 1,4
```

aligns the program counter one byte past the next long-word boundary relative to the start of the current section. Note that

```
cnop 0,2
```

is equivalent to the EVEN directive found in other assemblers and will ensure that the following data is aligned on an even address (i.e. a word boundary). This is normally only necessary when 68000 instructions follow byte-aligned data as the DC and DS directives word-align automatically.

CSECT name(,type,alignment,reltype,relsize)

Defines a program control section. Some form of section *must* be defined before any data can be generated. All parameters are optional except name and have the following functions:

name is the control section name, note that this is case sensitive.

type may be CODE (or 0) for instructions, DATA (or 1) for initialised data, or BSS (or 2) for uninitialised data sections; the default value is 0.

align specifies the alignment requirements of the control section as a power of 2; this parameter is currently ignored and all sections are longword aligned.

reltype specifies the relocation type, which determines the default addressing mode to be used for all symbol references and definitions from within the control section. The default value is 0.

relsize specifies the size, in bytes, of the relocation data for the section; the default value is 4. Legal type and size combinations for relocation information on the 68000 are summarised in the following table:

Type	Size (bytes)	Description
0	4	Absolute long addressing (default)
0	2	Absolute short addressing
1	2	PC-relative offset (PC)
2	2	Address-register-relative offset (A4)

A discussion of the use of CSECT directives which are compatible with the -b and -r options of the C compiler appears later.

(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 and any associated label will be aligned on a word boundary for DC.W and DC.L. You may also specify string expressions for DC.B within single or double quotes.

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 reserve uninitialised memory locations. Any label specified is set to the start of the area, which will lie on a word boundary for the DS.W and DS.L directives. If used within a BSS section, the reserved space is simply added to the section size and no object code is generated.

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

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

END

Signifies the end of program source.

ENDM

Terminates a macro definition. Must be used after a MACRO directive.

label EQU expression

This directive permanently assigns the value and type of a given label to be equivalent to the expression. If there is an error or forward reference in the expression, the assignment will not be made.

IDNT string

Currently ignored, provided for compatibility only.

INCBIN filename

Includes a binary file, verbatim, in the output file. Suggested uses include graphics data and ASCII files. You may specify a drive specifier and directory for INCBIN, otherwise it will default to searching the current directory.

INCLUDE filename

This directive will take source code from a file on disk and assemble it exactly as though it were present in the text. The directive must be followed by a filename in normal GEMDOS format. If a drive specifier or directory is included, the entire filename must be surrounded by quotes, e.g.

```
include "b:\constants\header.s"
```


In the absence of a drive specifier, the filename is taken to be relative to the current directory and any include directories specified on the command line are also searched.

Include directives may be nested up to 16 levels and if any error occurs when trying to open the file or read it, assembly will be aborted with a fatal error.

LIST

Turns on the assembly listing. 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.

(label) MACRO

This starts a macro definition causing all following lines to be copied into a macro buffer until a matching MEXIT directive is encountered. The presence of a label determines whether Motorola-style macros are to be used. Refer to the macro definition section for a more detailed explanation.

MEXIT

This can be used as part of a MACRO definition to stop the current macro expansion prematurely, usually as a result of a conditional.

NARG

This is not a directive but a reserved symbol. Its value is the number of parameters passed to the current macro. Note that \# may be used as a synonym for NARG.

NOLIST

Switches the assembly listing off.

OFFSET (expression)

The OFFSET directive switches code generation to a special dummy section for the generation of absolute labels. The optional expression sets the value for the first label, otherwise zero is used. No bytes are written to the disk and the only directive allowed is DS. This can be used to generate labels which represent offsets into a data structure. For example,

```
            offset    10
next        ds.l      1
title       ds.b      32
```

will assign the value of 10 to the label `next` and 14 to `title` (i.e. 1 longword after `next`). To return to ordinary code generation, use the CSECT or SECTION directive.

PAGE

Currently ignored, provided for compatibility only.

RORG expression

This directive changes the program counter to the specified number of bytes from the start of the current section. Note that the value specified *must be less than* the current PC.

SECTION name(,type)

Define a program section. There are no restrictions on name and the optional type may be one of the following (in upper or lower case):

CODE	code section (instructions)
DATA	data section (initialised data)
BSS	BSS section (uninitialised data)

The default type is CODE. Note that the SECTION directive is a subset of the CSECT directive which is explained in greater detail elsewhere.

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.

TTL string

Currently ignored, provided for compatibility only.

XDEF symbol(,symbol...)

Defined symbols may be exported using XDEF; the symbol type (relocatable or absolute) will also be exported.

XREF symbol(,symbol...)

This defines labels to be imported from other programs or modules. If any of the labels specified are already defined an error will occur, although importing a label more than once is accepted. Note that the symbol will inherit the relocation type of the control section in which it appears.

Conditional Assembly

Conditional assembly allows the programmer to write a comprehensive source program that can cover many conditions. 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 a corresponding ENDC directive.

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

These directives evaluate the expression, compare it with zero and then conditionally assemble depending on the result. The conditions correspond exactly to the 68000 condition codes with the exception of the IF directive, which is identical to IFNE.

IFD label
IFND label

These directives allow 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.

IFC 'string1','string2'
IFNC 'string1','string2'

Primarily for use within macros, these directives perform a case-sensitive comparison of two strings, both of which must be enclosed within quotes. IFC will only assemble the block if the strings match exactly, whereas IFNC does *not* assemble if the strings match.

ELSE

Toggles conditional assembly on or off. If the preceding conditional block was assembled, ELSE will cause assembly to stop until a matching ENDC is encountered, and vice-versa.

ENDC

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

Macro Definition

Asm supports two styles of macro definition. Motorola standard macros are defined via the following sequence:

```
name    MACRO
        ...
        ENDM
```

The definition must begin with the macro name followed by the directive MACRO. This is followed by the lines that comprise the macro itself, terminated by the ENDM directive. The MEXIT directive may also be used within the macro to terminate the macro early. Using this method of definition, macro parameters are referenced by a backslash and a number, for example

```
        move.w \2,(a0)
```

which would substitute the second macro parameter for \2. Alternatively, you may wish to use the second form of macro definition which is more flexible although non-standard:

```
        MACRO
        name    [arglist]
        ...
        ENDM
```

With this system the MACRO directive must appear first, followed by a line showing a model of how the macro will be called. The arglist is a comma-separated list of argument strings which provide macro parameter names and default values in the following format:

```
        arg[=default]
```

where arg is an identifier which can be used within the macro to refer to the corresponding argument text in the macro invocation and default is a string that will be associated with arg when that argument is not provided by a particular macro invocation. Note that default must be enclosed in single or double quotes if it contains any white space characters.

Both formats of macro definition support the NARG reserved word - and its alternative syntax of \# - which will be substituted with the number of macro arguments. Also, quoted strings may be passed as macro parameters.

In order to define labels within a macro you should use the special symbol \@. This causes the assembler to generate a unique number each time the macro is used, preventing multiple definitions of the same label.

The following example illustrates macro definition using the second style:

```
        MACRO
        MINWORD   source=#100,dest
        cmp.w     source,dest
        blt.b     min\@
        move.w    source,dest
min\@
        ENDM
```

The macro name is MINWORD and it could be invoked in the following way:

```
        MINWORD   ,d2
        rts
```

resulting in the instructions,

```
        cmp.w     #100,d2
        blt.b     min.0
        move.w    #100,d2
min.0
        rts
```

Note that the default value of #100 was substituted because the first parameter was omitted and that \@ was replaced by .0 (calling the macro a second time would use .1 etc.).

Interfacing C with Assembly Language

The aim of this section is to discuss the conventions which a program must follow when interfacing to C. Attention is given to features of the Lattice assembler, Asm, which assist in writing such code and some of the pitfalls which can occur. Full examples of both C calling an assembly language routine and assembly calling a C function are given towards the end of the section.

The following list covers the main points which you should bear in mind when writing assembly code for use with C. Each of these is covered in greater detail with examples later in the section.

- Separate control sections containing definitions or external references should be defined for code, initialised data and uninitialised data (BSS) via the CSECT or SECTION directives.
- Code references (including function calls) may use PC-relative addressing or branch instructions if the function is within a 32K range, otherwise you should use absolute addressing (i.e. a JSR instruction).
- Data references for near data should use register A4 as a base pointer whereas far data must use absolute addressing.

- Near data must be defined in the named section __MERGED.
- Standard argument passing functions are prefixed by an underscore (_) and use values pushed onto the stack.
- Register passing functions have a prefix of @ and place *some* arguments in registers with the remainder on the stack.
- The __asm specifier can be used to determine which register each function argument is passed in, with certain limitations.
- The size of type int may vary between word and long. Also, type char may be signed or unsigned depending upon compiler options.
- Return values appear in D0 with D1 also being used for double values. Note that the condition codes after a function call *cannot* be relied upon.
- A function may only corrupt registers D0-D1/A0-A1, all others *must* be preserved, including 68881 floating point registers (except for FP0/FP1) if used.

Control Sections

In order for an assembly language program to link correctly with C object files you must use named control sections. The Lattice assembler provides this facility through the SECTION and CSECT directives. The latter of these provides more powerful options concerning automatic conversion of addressing modes, although in many cases you can simply use SECTION. A summary of both options can be found in the assembly directives section.

Programs should be divided into *code* (assembly language instructions and routines), *data* (initialised data and constants) and *BSS* (uninitialised data) sections. Each of these is described in greater detail below.

Code Sections

All assembly language instructions should appear within code sections. The two simplest form of directives you can use to specify a code section are:

```
SECTION   name
CSECT     name
```

where name is the control section name. The compiler uses the default section name of text for all code generation although you may wish to use different names to identify program modules.

Any functions defined within a code section can be called from the same module with a branch or jump to subroutine instruction which you may wish to make PC-relative. However, in order to make a function visible to other modules when the program is linked you must define it as an external definition, for example,

```
XDEF      newtable
```

would make the function newtable callable from any other module. You should take into account that the C compiler automatically prefixes all external references with an underscore character (_). The XREF directive may be used to access an external reference which is defined in another module.

The CSECT directive may also be used to specify additional information about the control section; its general format is:

```
CSECT     name,type,align,reltype,relsize
```

Only the name parameter must be present; it specifies the name of the control section. The type parameter describes the type of section; code, data or BSS (the values 0, 1 and 2 may also be used). The align parameter specifies the alignment requirements of the control section. The last two parameters, reltype and relsize, specify the type and size of relocation information associated with symbols declared within the control section.

For example, the section directives described previously are equivalent to:

```
CSECT     name,code,4,0,4
```

which is interpreted as a named code section, aligned on a longword boundary, defaulting to absolute longword addressing for symbols. The final two parameters can be used in code sections to automatically convert absolute long addressing to PC-relative for more compact code, as in

```
CSECT     text,0,,1,2
XREF      _function
JSR       _function
```

Note that we have used the number 0 rather than code and the alignment parameter has been omitted as all sections are longword aligned. The JSR instruction will actually be assembled as

```
JSR       _function(PC)
```

because we have specified a relocation type of PC-relative. To override this you may move the XREF out of the PC-relative section. It is also possible to use several code sections with different relocation types, the assembler will only use PC-relative addressing for symbols declared in the correct sections.

The advantage of using CSECT to provide PC-relative instructions is that changing a single CSECT directive gives you the ability to transform all external references. This provides you with an equivalent mechanism to that provided by the -r option on lc.

To call a C function from an assembly language module, you must always include an XREF declaration for the function. Before calling the function (via JSR or BSR), you must supply any expected arguments in the proper order either on the stack or in registers, depending upon the style of parameter passing employed by the function. After control returns from the called function, the stack pointer must be adjusted to account for any pushed arguments.

```
        XREF    _cfunc

        MOVE.L  D0,-(A7)  ;push argument
        MOVE.L  D1,-(A7)
        JSR     _cfunc          ;call function
        ADDQ.W  #8,A7           ;restore stack pointer
```

This code fragment illustrates stack parameter passing, more details can be found in the relevant section. Remember to prefix function names with an underscore (_) or @ symbol accordingly.

Data Sections

There are two types of control sections in which program data can be held; *data* and *BSS* sections (described later). The first of these is for initialised data and constants and may be defined with either of the following directives,

```
SECTION   name,data
CSECT     name,data
```

where name is the control section name. The compiler uses two names for data sections; data for far data (this is accessed with absolute long addressing) and __MERGED (the program's near data, accessed as a base-relative offset from register A4). Examples of instructions used to access each type of data are

```
move.w    fardata,d0
move.w    neardata(a4),d1
```


When defining global data in assembly which is accessed by a C program you must declare the symbol as an external with an XDEF directive. The C source must also include an extern declaration of the correct type. For example, this assembly program *defines* a global variable:

```
            CSECT     asmdata,data
            XDEF      _entrynum

_entrynum   DC.W      15
            END
```

Note that data is always prefixed with an underscore. This can be done automatically via the -u option. The corresponding C code to declare the variable is as follows,

```
extern unsigned short far entrynum;
```

The Lattice assembler provides a way of specifying a near data section, i.e. where all the data lies within a 32K range which is accessed off A4. All absolute longword references to symbols declared within such a control section will automatically be converted to the address-register-relative addressing mode. This is done through the CSECT directive:

```
CSECT     __MERGED,data,,2,2
```

where the case of the section name *is* important. In practice, this gives you a direct equivalent to the -b option of lc, allowing you to change the arrangement and thus the access mode for any data by simply placing it in an appropriate control section. Consider the following code:

```
            SECTION   text
            move.w    globl,d0
            move.l    _otherdata,d1
            rts

            CSECT     __MERGED,1,,2,2
            XREF      _otherdata
globl       DC.W      42
```

The move instructions will actually be assembled as

```
            move.w    global(a4),d0
            move.l    _otherdata(a4),d1
```

because the symbols were declared in a near data section.

BSS and Offset Sections

The second form of data section is the *BSS* or uninitialised data section. It behaves in exactly the same way as a regular data section except that the only directive allowed is the DS directive. By placing all data which you require to be initialised to zero in the BSS section you can save considerable file space because no data is actually written, the *size* of the section is merely remembered.

The directives to start a BSS section are identical to data sections in every respect other than the section type. The special section name of __MERGED is also recognised for near data in a similar way to that described previously.

Although visibly very similar to a BSS section, an *offset* section describes merely the layout of data and not actually a specific instance of it. The primary use of the OFFSET directive is to provide a simple way to declare offsets into data structures. For example, here is a structure described in C:

```
struct NameNode {
          struct NameNode *next;
          struct NameNode *prev;
          int uses;
          unsigned char name[16];
};
```

In order to use this structure from an assembly language program, we must use numerical offsets into the structure. To aid readability and maintainability we wish to use symbols which refer to each element. The following description provides just that:

```
          OFFSET
nn_next   DS.L      1
nn_prev   DS.L      1
          IFD       SHORTINT
nn_uses   DS.W      1
          ELSE
nn_uses   DS.L      1
          ENDC
nn_name   DS.B      16
sizeof_nn DS.B      0
```

This does not generate any code, simply offset values. The symbols nn_next, nn_prev and nn_uses will be set to the absolute values of 0, 4 and 8 respectively. The prefix of nn_ has been added to avoid possible name clashes with other symbols and the dummy entry sizeof_nn provides a convenient way of referring to the size of the entire structure.

A conditional block has been used around the integer field because the length of an integer may vary between word and longword. Using this method, re-assembling the source with the -w flag for short integers will automatically generate the correct offsets. Some code which accesses this structure might look like the following:

```
lea       firstnode(a4),a0
subq.w    #1,nn_uses(a0)
move.l    nn_next(a0),a0
rts
```

Function Entry Rules

There are several rules which the compiler enforces to provide a mechanism for calling functions. These rules must also be followed by assembly programmers wishing to interface with C.

Regardless of how the function was called, register A7 (the stack pointer) always points to a return address. Register A4 points into a program's near data to allow base-relative addressing as discussed in the previous section.

Depending upon the style of parameter passing employed by a particular function, parameters may either be found on the stack, in registers or a combination of both. Arguments are always passed by value. An explanation of the three methods of parameter passing follows.

Standard Arguments

This is the default method of parameter passing where all function arguments are placed on the stack immediately before the return address. The __stdargs keyword may also be used in a function prototype or definition to force stack parameters. Note that functions which take a variable number of parameters *always* use standard argument passing.

Register A7 is the stack pointer which points to the 4-byte return address followed by the arguments in left-to-right order. Arguments can then be accessed as an offset from the stack pointer. The exact location of the parameters on the stack depends on the argument types and the current flags. Considering the default long integer mode, for the function call:

```
char ccc;
double ddd;
int iii;
func(ccc,ddd,iii);
```

The compiler generates code to extend each of the parameters to the size of an int if it is smaller and then push the arguments onto the stack in *reverse* order. For example,

```
move.l    d0,-(sp)
movem.l   d2-d3,-(sp)
ext.w     d1
ext.l     d1
move.l    d1,-(sp)
```

This results in a stack organised in the following way:

Location	Size	Contents
(A7)	4	Return address
4(A7)	4	Argument ccc
8(A7)	8	Argument ddd
16(A7)	4	Argument iii

By comparison, in default short integer mode (option -w) the compiler would generate code to push the arguments ccc, ddd, and iii onto the stack using *two* bytes, eight bytes and *two* bytes, respectively:

```
move.w    d0,-(sp)
movem.l   d2-d3,-(sp)
ext.w     d1
move.w    d1,-(sp)
```

Location	Size	Contents
(A7)	4	Return address
4(A7)	2	Argument ccc
6(A7)	8	Argument ddd
14(A7)	2	Argument iii

Note that due to the widening of char types to the size of an int, the actual parameter is in the *low byte* of the int although the full integer value may be used. Also remember that char may be signed (the default) or unsigned depending upon compiler options.

If a structure or union is passed by value to a function, then the contents of the aggregate are copied onto the stack with the last element pushed first. In effect you receive a complete copy of the aggregate on the stack followed by a single byte for alignment if necessary.

Stack space occupied by function arguments may be used by the function as temporary workspace once the values are no longer needed.

Register Arguments

If a function is explicitly declared __regargs or is called from a module compiled with the -rr option, some arguments are passed in registers instead of on the stack. Note that functions which accept a variable number of parameters always use the previous style of parameter passing.

With register parameters, the first two pointer arguments will appear in A0 and A1, and the first two integral arguments will be in D0/D1 and widened to an int if necessary as previously described. Structures, unions and double precision floating point numbers, along with any parameters not placed in registers are passed via the stack in the usual way.

Obviously, the function needs to know whether it is being called with some arguments in registers or with all arguments on the stack. The compiler helps make this distinction by placing the character @ in front of function names that are called with register arguments, replacing the underscore that the compiler normally supplies as a function prefix.

The __asm Keyword

Providing much greater control over register passing, the __asm keyword allows you to specify exactly which registers parameters are to be passed in. It can be used in both function definitions and declarations:

```
int __asm mymax(register __d0 int,register __d1 int);

int __asm myfun( i,p )
          register __d0 int i;
          register __a1 char *p;
```

In order for the register specifier sequence to be used, you must have the __asm keyword specified on the function. If you do use the __asm keyword, you *must* specify a register for each parameter and not re-use the same register for any two parameters. If you need to pass some parameters on the stack then you should use the __regargs keyword instead. Note that currently the compiler is restricted to returning only basic types like long, double, etc.

In order to permit the most flexibility in register passing, the compiler does not limit what registers may be passed. However this can lead to situations in which it is impossible to generate code that works in the presence of aliased variables. To ensure that such situations are not encountered, you should avoid utilising registers that would normally be assigned as register variables and instead only use the registers:

```
__d0        __a0
__d1        __a1
__d2        __a2
__d3
```

The best advice is to be careful when using this feature and if you are uncomfortable with it, use the -rr option of lc1 (or __regargs).

Another mechanism which may be used to achieve similar effect to __asm is the #pragma inline statement described in detail elsewhere in this manual. When no instruction stream is present, this will generate a function call which may use any register or the stack for parameters and may use any register for the return value.

Function Exit Rules

Function return values are passed back in one or more registers, depending on the data type declared for the function. The conventions are:

Return Data	Bits	Asm Syntax	Meaning
char	8	D0.B	Low byte of D0
short	16	D0.W	Low word of D0
long	32	D0.L	All of D0
float	32	D0.L	All of D0
double	64	D0.L, D1.L	High bits in D0
pointer	32	D0.L	All of D0

Note that the above table does not mention int. An assembly language function should return its value as a short, if in default short integer mode (-w) or as a long if not in that mode, i.e. D0.W or D0.L.

If the function returns a structure or union, it must define a static work area (i.e. not on the stack) to temporarily hold the returned object. Then the function must return in D0 a pointer to this temporary copy, and the calling function will immediately move the data to the appropriate place. This approach implies that functions returning structures or unions are not re-entrant, although they are serially re-usable. Such functions *can* be recursive if designed very carefully with this in mind.

The registers D2 through D7 and A2 through A6 must be saved if they are used by the function, similarly if a 68881 maths co-processor is present (only possible on 68020 or 68030 systems) and any of the floating point registers FP2 through FP7 is used, they must also be saved.

After setting up the return value, a function exits with the RTS instruction. Note that the calling function removes the arguments from the stack.

Calling Assembly from C

To illustrate how the rules governing C functions affect an assembly language routine we have chosen a short example which can be implemented either as C calling assembly, or assembly calling C (the C and assembly object modules must be linked with the startup code and appropriate libraries). It illustrates many of the points made previously and can be used as a basis for your own function calls.

The function returns a hash value calculated by adding together the ASCII codes of each character in the supplied string up to a specified length. This value is then divided by the number held in the global variable maxhash and the remainder (or modulo) is returned.

The calling program simply defines and initialises the variable maxhash and calls the hash function with a sample string. Implemented in C, this is as follows:

```
unsigned short maxhash;  /* definition */
/* declaration (prototype) */
unsigned int hash(unsigned int length, const char *string);

void main(void)
{
        unsigned int result;

        maxhash = 101;
        result = hash(4,"Banana");
}
```

The hash function coded in assembly language for default addressing modes, parameter passing and types:

```
            CSECT     text,code       control section
            XDEF      _hash,@hash     declarations
            XREF      _maxhash        imported global

_hash                                 ; stdargs entry point
            movem.l   4(sp),d0/a0     get the parameters
@hash                                 ; regargs entry point
            move.l    d2,-(sp)        preserve register
            moveq     #0,d2
            bra.s     1$
2$          move.b    (a0)+,d1
            ext.w     d1
            ext.l     d1
            add.l     d1,d2
1$          subq.l    #1,d0
            bcc.s     2$
            divu      _maxhash(a4),d2
            clr.w     d2              make result 32-bit
            swap      d2              get remainder
            move.l    d2,d0           return value in D0
            move.l    (sp)+,d2        restore register
            rts

            END
```

Any labels available to the C program are prefixed by an underscore character (_) or @. Note that for this function, it is easy to provide an entry point for register parameter calling by simply bypassing the code which loads arguments from the stack into registers for use by the body of the function. If you are using register parameters as default, you may leave out this code entirely.

The global variable is accessed as a base-relative offset from A4 because we are using default near data. The function must also save D2 on the stack because it is used as a temporary register and must be restored.

Compiling the program with default short integers, unsigned char and far data does not change the C source although it causes many changes to the assembly language. The function must now be changed to:

```
_hash
            move.w    4(sp),d0        length is now a word
            move.l    6(sp),a0        changing stack offsets
@hash
            move.l    d2,-(sp)
            moveq     #0,d1           can't sign extend char
            moveq     #0,d2
            bra.s     1$

2$          move.b    (a0)+,d1
            add.l     d1,d2
1$          dbra      d0,2$           optimised loop
            divu      _maxhash,d2
            swap      d2              don't clear high word

            move.w    d2,d0
            move.l    (sp)+,d2
            rts
```

Note that the parameters now have different offsets on the stack, characters can no longer be sign extended and global data must be accessed using absolute long addressing.

It becomes apparent that changes in compiler options such as -b or -r can dramatically alter the appearance of assembly code. The Lattice compiler provides some ways of insulating the programmer from these factors, as illustrated in the next section.

Calling C from Assembly

This time, we will write the same program but as a C function called from assembly language. In order to provide the greatest flexibility whilst preserving code clarity, we will make use of the CSECT directive. This is the calling program for register arguments only:

```
            CSECT     text,code,,1,2
            XDEF      @main           PC-relative
            XREF      @hash

@main                                 ; regargs version
            move.w    #101,maxhash
            moveq     #4,d0
            lea       string,a0
            jsr       @hash           returns D0
            rts

            CSECT     __MERGED,data,,2,2
string      DC.B      'Radish'        data access off A4

            CSECT     __MERGED,bss,,2,2
            XDEF      maxhash
maxhash     DS.W      1
            END
```

Firstly, you may notice that there are no longer underscores before external labels. This is because the assembler can be called with the -u option which automatically prefixes an underscore to all externally visible labels whilst being overridden by the presence of an @ symbol.