SNASM 68000

Development System

User's Manual

Version 2.01

The information contained in this document is subject to change without notice.

SNASM2 68000 User's Manual

First edition, November 1993
Revised, January 1994
Revised, February 1994

Assembler Quick Reference

Switch	Description
?	Brings up on-line help describing the syntax for switches, options and optimisations.
b *Size*	Set the *Size* of the input buffers in KBytes from 1-64, the default being 16K. Note that there must be at least one space character between b and *size*.
coff	Change between big endian and little endian COFF output file.
d	Debug mode only; assemble the code but do not run it.
e *Symbol*=[*Val*|"*Str*"]	Equate a symbol to a value or a string. The symbol will be set to 1 if no value or string is specified. Multiple equates are separated by semicolons (';').
emax *NumErrors*	Abort the assembly after the number of errors exceeds that specified by *NumErrors*. The default value of *NumErrors* is zero which will not abort the assembly.
g	Write non-global symbols to linker object file.
hex *Number*	Set the width of hexadecimal output in the listing file from 2-8 words, the default being 4.
i	Display information window during assembly.
j *Dir* [;*Dir*]...	Specify the search directory for INCLUDE file. If an INCLUDE filename does not specify a path, by default the assembler first looks for files in the current directory. If it cannot be found there the assembler looks in the directories built up using the 'j' switch.
k	Enable additional conditional assembly structures. These are implemented via macros and are described on page 111.
l	Produce linkable output file.

Switch	Description
`lnos`	Show source code line numbers in the listing file.
`m`	List external symbols to map file (linking only).
`o` *Options*	Set assembler options and optimisations. Note that there must be at least one space character between the switch and the parameter.
`p`	Produce pure binary output file.
`q` *Quirks*	Enable quirks. Quirks are special options that enable certain features specific to SNASM68K version 1.x.
`sdb`	Add source debug information to COFF file.
`w`	Write equates to symbol table.
`z`	Include line number information in linker (.COF) file.

Quirk	Description
`fl-/+`	*Functions in lower case* Specify names of functions and pre-defined constants in lower case if the case sensitivity option is enabled.
`mc-/+`	*Macro continuation character.* Allows the use of '\' as a line continuation character in macro calls and on the first line of a macro definition.
`mp-/+`	*Macro parameter lower case.* Sets unquoted macro parameters to lower case if the assembler is set to be case insensitive.
`regs-/+`	*Registers.* Causes REGS statements to be added to the default group instead of the currently opened section.
`sa-/+`	*Section alignment.* Aligns a section re-opened without a size modifier to the previously defined alignment for that section. This applies to both the SECTION and POPS directives.

Option	Default	Description
`ae+/-`	On	*Automatic even.* This forces the program counter to the next word boundary before assembling the word and long forms of DC, DCB, DS and RS.
`an-/+`	Off	*Alternate numeric.* Allows the use of character suffixes H, D, Q and B to denote Hexadecimal, Decimal, Octal and Binary constants respectively.
`bin-/+`	Off	*Show binary.* Show full binary in the listing file.
`c-/+`	Off	*Case sensitivity.* By default all symbols are case insensitive, for example `Main` and `main` are treated as the same label. Enable this option to make labels case sensitive so that `Main` and `main` would be two distinct labels.
`d-/+`	Off	*Descope local labels.* When used outside of modules the EQU and SET directives do not affect the scope of local labels. Set this option if you want these directives to de-scope local labels.
`l-/+`	Off	*Local label character.* Toggle between '.' (`l+`) and '@' (`l-`) as the local label character.
`l`*Value*		Define the local label character where *Value* is the ASCII code for the character. Valid local label characters are '@', '.', ':', '?', '|' and '!' *only.* The default local label character is '@'.
`lf-/+`	Off	*List failed.* This lists instructions not assembled due to conditional assembly statements.
`m-/+`	Off	*List macros.* Lists macro expansions in listing file.
`mc-/+`	Off	*Show macro calls.* In the listing file, shows each macro invocation including nested macros and the nesting level.

Option	Default	Description
s-/+	Off	*Equated symbols as labels.* Treat equated symbols as labels.
t-/+	Off	*Truncate.* Disables expression overflow checking.
v-/+	Off	Write local labels to MAP file.
w+/-	On	*Print warnings.* Warnings are not errors but unusual occurrences that can be reported.
ws-/+	Off	*Allow white space.* This allows space and tab characters in operands to increase code readability. Normally a blank terminates the operand field and begins the comment field. With the ws option enabled the comment field must begin with a semi colon (;) and white space in the operand field is ignored.
x-/+	Off	*External symbols.* Assume external symbols are in the section they are declared in.

Optimisation	Description
op+/-	*PC Relative.* Changes absolute long addressing to PC relative addressing if possible and legal.
os+/-	*Short Branch.* Forces forward references in relative branches to use the short form of the instruction.
ow+/-	*Absolute Word.* Forces absolute word addresses to short word addressing if in range.
oz+/-	*Zero Displacement.* Changes address register indirect with displacement to address register indirect if the displacement evaluates to zero.
oaq+/-	*Quick ADD.* Changes the ADD instruction to the shorter ADDQ.
osq+/-	*Quick SUB.* Changes the SUB instruction to the shorter SUBQ.
omq+/-	*Quick Move.* Changes the MOVE.L instruction to the shorter MOVEQ. MOVE.W is not changed as MOVEQ is defined as long.

Debugger Quick Menu Reference

Main Window Menus

File

Load Binary & Debug Info...	Shift+Ctrl+C
Load Debug Info Only...	Ctrl+C
Send Binary...	Shift+S
Get Binary...	Shift+G
Disassemble to File...	Shift+D
Hex Dump to File...	Shift+H
Shell to DOS	Ctrl+Z
Prompt and Exit	F3
Save and Exit...	Ctrl+X
Quit (No Save)	Ctrl+Q

Session

Load...	F4
Save...	Ctrl+F3

Target

Select...	Shift+0..7
Discard...	Alt+0..7
Update Rate...	Shift+U
Monitoring	Ctrl+M
Memory Fill...	Shift+F
Memory Copy...	Shift+C

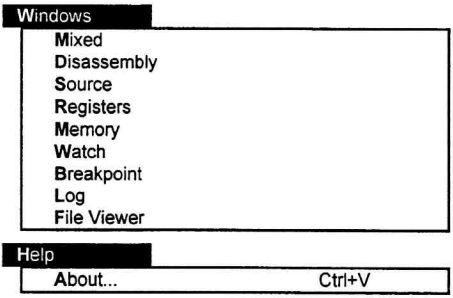
Execution

Run from PC	F9
Run to Address...	Shift+F9
Single Step	F7
Step Into	Shift+F7
Step Over	F8
Unstep	Ctrl+F7
Stop	Esc
Stop (DMA & Interrupts)	Shift+Esc
Reset Processor	Shift+Ctrl+R
Save Registers	Ctrl+S
Retrieve Registers	Ctrl+R

Breakpoints

Remove All	Shift+F5
Reset All Counts	Shift+F6
Disable All	
Enable All	

Main Window Menus (continued)



Mixed Window Menus

Display

Update Rate...	Shift+U
Zoom	Ctrl+Y
Switch Active	Space
Centre on Trace	
Centre on Bpoint	
Centre on Instr. Error	
Centre on Scroll	

Origin

Goto...	Ctrl+G
Go to Cursor	Home
Toggle Window Lock	Ctrl+L

Format

Show Line Nos.	
Set Tab Width	
Show Symbols	
Show Upper-Case	
Show T-States	Ctrl+T
Show Instr. Code	Ctrl+I
Decimal	Ctrl+D
Hex	Ctrl+H

Execution

Run from PC	F9
Run to Address...	Shift+F9
Run to Cursor	F6
Single Step	F7
Step Into	Shift+F7
Step Over	F8
Unstep	Ctrl+F7
Stop	Esc
Stop (DMA & Interrupts)	Shift+Esc
Reset Processor	Ctrl+Shift+R
Save Registers	Ctrl+S
Retrieve Registers	Ctrl+R

Breakpoints

Toggle at Cursor	F5
Configure...	Ctrl+F5
Remove All	Shift+F5
Reset All Counts	Shift+F6
Disable all	
Enable All	

Utils

Expression Calculator...	Ctrl+E
Find...	Ctrl+F

Disassembly Window Menus

Display

Update Rate...	Shift+U
Zoom	Ctrl+Y
Centre on Trace	
Centre on Bpoint	
Centre on Instr. Error	
Centre on Scroll	

Origin

Goto...	Ctrl+G
Go to Cursor	Home
Toggle Window Lock	Ctrl+L

Format

Show Symbols	
Show Upper-case	
Show T-states	Ctrl+T
Show Instr. Code	Ctrl+I
Decimal	Ctrl+D
Hex	Ctrl+H

Execution

Run from PC	F9
Run to Address...	Shift+F9
Run to Cursor	F6
Single Step	F7
Step Into	Shift+F7
Step Over	F8
Unstep	Ctrl+F7
Stop	Esc
Stop (DMA and Interrupts)	Shift+Esc
Reset Processor	Ctrl+Shift+R
Save Registers	Ctrl+S
Retrieve Registers	Ctrl+R

Breakpoints

Toggle at Cursor	F5
Configure...	Ctrl+F5
Remove All	Shift+F5
Reset All Counts	Shift+F6
Disable all	
Enable All	

Utils

Expression Calculator...	Ctrl+E
Find...	Ctrl+F

Source Window Menus

Display

Update Rate...	Shift+U
Zoom	Ctrl+Y
Centre on Trace	
Centre on Bpoint	
Centre on Instr. Error	
Centre on Scroll	

Origin

Goto...	Ctrl+G
Go to Cursor	Home
Toggle Window Lock	Ctrl+L

Format

Show Line Nos.	
Set Tab Width	

Execution

Run from PC	F9
Run to Address...	Shift+F9
Run to Cursor	F6
Single Step	F7
Step Into	Shift+F7
Step Over	F8
Unstep	Ctrl+F7
Stop	Esc
Stop (DMA and Interrupts)	Shift+Esc
Reset Processor	Ctrl+Shift+R
Save Registers	Ctrl+S
Retrieve Registers	Ctrl+R

Breakpoints

Toggle at Cursor	F5
Configure...	Ctrl+F5
Remove All	Shift+F5
Reset All Counts	Shift+F6
Disable all	
Enable All	

Utils

Expression Calculator...	Ctrl+E
Find	Ctrl+F

Registers Window Menus

Display

Update Rate...	Shift+U
Zoom	Ctrl+Y

Edit

Increment	+
Decrement	-
Inc/Dec Amount...	
Expression...	Enter
Reset Processor	Ctrl+Shift+R
Save Registers	Ctrl+S
Retrieve Registers	Ctrl+R

Utils

Expression Calculator...	Ctrl+E
Find...	Ctrl+F

Memory Window Menus

Display

Update Rate...	Shift+U
Zoom	Ctrl+Y

Origin

Goto Address...	Ctrl+G
Goto Cursor	Home
Goto Pointer at Cursor	Ctrl+P
Toggle Window Lock	Ctrl+L

Format

Show ASCII	Ctrl+A
Bytes Per Line...	Ctrl+B
Show Bytes	Shift+B
Show Words	Shift+W
Show Longs	Shift+L

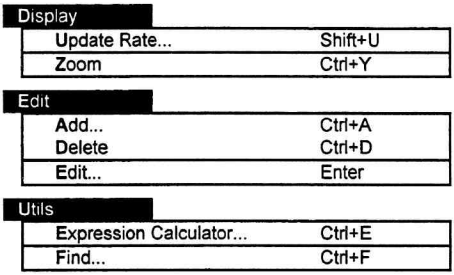
Edit

Increment	+
Decrement	-
Inc/Dec Amount...	
Expression	Enter

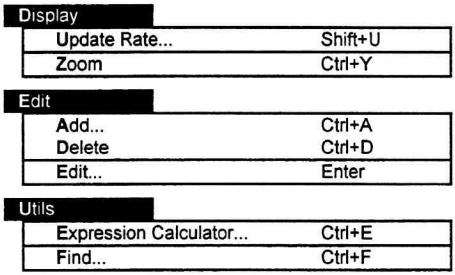
Utils

Expression Calculator...	Ctrl+E
Find...	Ctrl+F

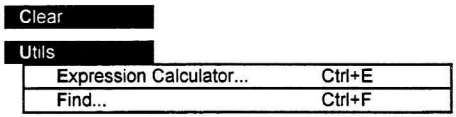
Watch Window Menus



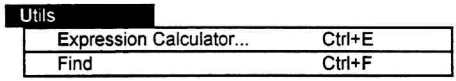
Breakpoint Window Menus



Log Window Menus



File Viewer Window Menus



Debugger Quick Key Reference

All Windows

	Short-cut Key	
Windows	F2	Select Main window
	Alt+Esc	Toggle Between Main Window and Last Selected Window
	Ctrl+N	New window for the current target
	Ctrl+W	Select Window for current target from a list of open windows
	Ctrl+0..9	Select window for current target
	Ctrl+↑ or →	Next Window for Current Target
	Ctrl+↓ or ←	Previous window for Current Target
	F10	Select first menu from current window
	Alt+Space	Selects the System Menu for the current Window
	Ctrl+L	Toggle Window Lock
	Shift+P	Prints the window contents
	Ctrl+Y	Zoom
	Alt+F4	Closes Current Window
Display	PgUp, PgDn	Scrolls display by pages
	Shift+← or →	Scrolls Page Left or Right
	Shift+↓ or ↑	Scrolls display Up or Down
Files	Shift+Ctrl+C	Load Binary and Debug Info
	Ctrl+C	Load Debug Info only
	Shift+S	Send Binary
	Shift+G	Get Binary
	Shift+D	Disassemble to File
	Shift+H	Hex Dump to File
	Ctrl+Z	Shell to DOS
	F3	Prompt and Exit
	Ctrl+X	Exit and save session file
	Ctrl+Q	Quit without saving
Session	F4	Load Session File
	Ctrl+F3	Save Session File
Target	Shift+0..7	Selects a target
	Alt+↓	Selects next available target
	Alt+↑	Selects previously available Target
	Alt+0..7	Discards a target
	Shift+U	Window Update Rate
	Ctrl+U	Toggle Continuous Update Rate for all windows and all targets.
	Ctrl+M	Toggle Monitoring for current target
	Shift+F	Memory Fill
	Shift+C	Memory Copy
	Ctrl+F1	Move Debugging link workspace

All Windows (continued)

	Short-cut Key	
Execution	F9	Run from PC
	Shift+F9	Run to Address
	F7	Single Step
	Shift+F7	Step Into
	F8	Step Over
	Ctrl+F7	Unstep
	Esc	Stop Program
	Shift+Esc	Stop (DMA and Interrupts)
	Shift+Ctrl+R	Reset Processor
	Ctrl+S	Save Registers
	Ctrl+R	Retrieve registers
Breakpoints	Shift+F5	Remove all Breakpoints
	Shift+F6	Reset All Counts
Utils	Ctrl+E	Expression Calculator
	Ctrl+F	Find
Help	F1	Help
	Ctrl+V	About Box

Main window

	Short-cut Key	
Target	0..7	Selects (initialises) a (new) target
	Enter	Displays Target Configuration dialog
	Shift+N	Name a target
Display	←,→	Scrolls display left or right
	↑, ↓	Selects between targets
	Home	Selects first available target
	End	Selects lasts available target
	Tab	Ensures status line for current target is visible

All Debug Windows

	Short-cut Key	
Windows	←,→	Moves cursor
	↑, ↓	Moves cursor
	Home	Displays window contents from cursor position*
	End	Displays window contents to cursor position*
	Tab	Displays window contents from current value of PC
	Ctrl+Home	Forces cursor to start of display
	Ctrl+End	Forces window to end of display
	Shift+Home	Forces the cursor to the start of line
	Shift+End	Forces the cursor to the end of line
	Shift+Tab	Forces value PC to current cursor position
	Shift+N	Name a window
	Space	Toggle Focus in Mixed Window
Edit	Enter	Edit to result of expression
	0..9	Edits value at cursor
	A..F	Edits value at cursor
	+	Increments value at cursor
	-	Decrements value at cursor
	Ctrl+G	Goto result of expression or line number

* Except for the Watch window

Memory Window

Short-cut Key	
Ctrl+A	Toggle ASCII Display
Ctrl+B	Bytes per line
Alt+→	Move cursor to next Byte, Word or Long
Alt+←	Move Cursor to previous Byte, Word or Long
Shift+B	Shows Bytes
Shift+W	Shows Words
Shift+L	Shows Longs

Disassembly window

	Short-cut Key	
Display	←,→	Shift Window contents by default alignment
	Alt+→	Move cursor forward by default alignment
	Alt+←	Move cursor back by default alignment
	Ctrl+D	Show Decimal
	Ctrl+H	Show Hex
	Ctrl+I	Show Instruction Code
	Ctrl+T	Show T-states
Execution	F6	Run to Cursor
Breakpoints	F5	Toggle Breakpoint at Cursor
	Ctrl+F5	Configure Breakpoint

Watch Window

Short-cut Key	
Home	Goto first watch expression.
End	Goto last watch expression.

This is the only information this page contains.

Contents

List of Figures

List of Tables

This is the only information this page contains.

1 Getting Started

This section provides an overview of new features and shows you how to fit the PC Card and install the SNASM68K software. The READ.ME file may contain important information that was not available when this manual was printed. Please read this information before continuing.

System Requirements

The SNASM2 development system has the following system requirements:

An IBM (or fully compatible) 386 based machine or greater.
A 3.5" high density disk drive.
A hard disk with at least 1.5MB of free space available
A minimum of 2 MB of RAM. 4MB or more is recommended.
DOS 3.3 or greater

Conventions

The following typographical conventions are used in this manual.

Example	Description
SNASM68K.EXE, TEST.PRJ	Programs and filenames.
Home, End	Keys you press.
Alt+F4, Ctrl+C	Key combinations
Make, Debug	Menu items.
`end` [*Expression*]	In syntax, items inside square brackets are optional.
`public` {`on`|`off`|*Flag*}	In syntax, curly brackets and a vertical bar indicate a compulsory choice between two or more items.

1.1 Installing the PC Card

This section describes the PC Card and shows you how to fit the card. The card does not use DMA or interrupts although provison for future use of these features has been made through jumpers JP1-9 (currently unused). If DMA is required in the future then users will be informed along with the new software release.

1.1.1 Component Locations

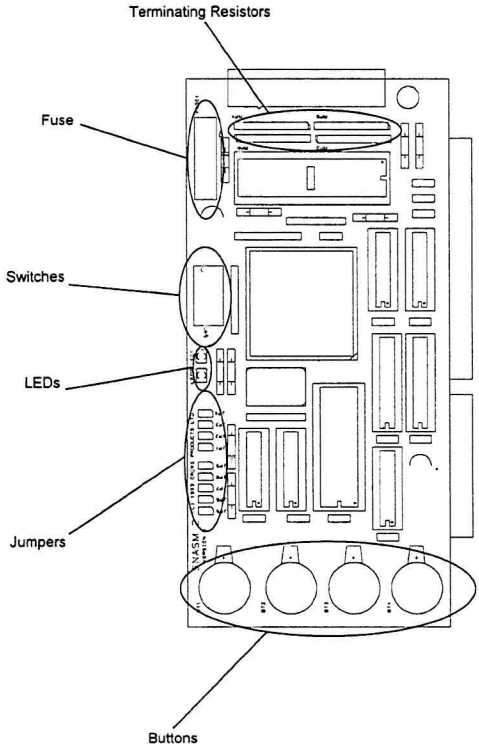


Figure 1. Component locations on the PC Card.

1.1.2 Configuring the PC Card

Caution

Electrostatic discharge (ESD) can damage components in your PC. You should install the PC Card only at an ESD workstation. If such a station is not available, you will provide some protection from ESD if you attach an anti static wrist strap to a metal part of your PC. Always hold the PC Card by its edges and be careful not to touch the board's electronic components or its gold connectors.

Setting SCSI Device Numbers

The SCSI bus supports up to eight devices, including the PC. Each device must have a unique SCSI device number, set using switches SW1-3 on the PC Card.

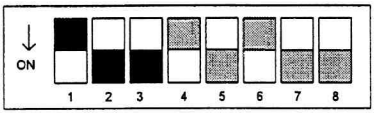


Figure 2. Setting SCSI device numbers. By default the PC Card has a SCSI Device Number of 6 (SW1 is *off* and SW2-3 are *on*). SW4 is not used.

Setting the PC Card Address

The PC Card can reside at one of sixteen addresses in your PC's port map. The default address is $390 and you will not normally need to change this. The default settings are shown in Figure 3. If you are upgrading from previous versions of SNASM, set the new card to the same address as your old one. However, if the PC Card causes conflict with other cards you will have to change the address using switches SW5-8. Table 1 shows the possible addresses that the card can reside at, which switches to set and the suitability of the address. If the default address requires changing first check other cards to see what addresses they are using then try to place the card at one of the remaining possible addresses.

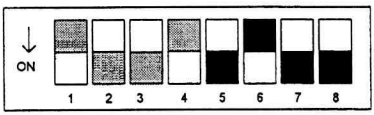


Figure 3. Setting the PC Card address. By default the PC Card resides at address $390 (SW5, 7 and 8 *on*, SW6 *off*).

SW8	SW7	SW6	SW5	Address	Suitability
Off	Off	Off	Off	200	Good
Off	Off	Off	On	210	Good
Off	Off	On	Off	220	Good
Off	Off	On	On	230	Good
Off	On	Off	Off	280	Good
Off	On	Off	On	290	Good
Off	On	On	Off	2A0	Good
Off	On	On	On	2B0	Poor
On	Off	Off	Off	300	Good
On	Off	Off	On	310	Very Good
On	Off	On	Off	320	Poor
On	Off	On	On	330	Good
On	On	Off	Off	380	Good
On	On	Off	On	390	Default
On	On	On	Off	3A0	Poor
On	On	On	On	3B0	Poor

Table 1. The PC Card can reside at any of the above addresses. The default settings are SW1, 2, 4, 5, and 6 *on*, SW3, 7 and 8 *off* so that the address is $390.

1.1.3 Fitting the PC Card

Warning

Unplug your PC before you install the PC Card. Failure to disconnect the power supply before installing the PC Card can result in personal injury or equipment damage.

EISA compatibility

For EISA machines, it is recommended that the PC Card is installed in an ISA slot if one is available. If an ISA slot is unavailable choose a slot for the PC Card and run the configuration software. with the following information:

IO space	8 bytes
IRQ	None
DMA	None
Memory	None

To install the PC Card:

1. Confirm that all switches and jumpers are in the proper positions before installing the card.
2. Remove the cover from your PC.
3. Remove and save an expansion slot cover and screw.
4. Hold the board by its top edge or upper corners and push the board into the expansion slot connector, making sure the PC Card is firmly seated.
5. Align the rounded notch in the retaining bracket with the threaded hole in the expansion slot frame. the retaining bracket fits into the space that was occupied by the expansion slot cover.
6. Insert the screw. Make sure to push the bracket up against the screw before you tighten it; otherwise, the bracket may interfere with the bracket of an adjacent board.
7. Replace the PC cover and re-install all the case screws. Do *not* turn the power on.

This completes the PC Card installation.

1.2 Setting up the Console Interface Unit

1.2.1 About the Console

The Console Interface unit contains 1, 2 (standard), 4, or 8MB of static RAM for EPROM cartridge emulation and 32K of battery-backed RAM, the top 1K of which is used by the link software. For 1MB systems the main 1MB of RAM can be mapped in on any 1MB boundary. The battery backed RAM cannot be moved and is mapped at $200000 for 1Mb and 2MB systems and at $400000 for 4MB systems.

When the processor is running code in the units internal EPROM's the main RAM is write enabled so the WR PROT light is off. When code is being run from the main RAM this RAM is write protected so the WR PROT light is on.

To use the target machine as a normal games console (i.e. as if the interface was not connected) push the ENABLE switch on the front panel of the main unit to the left. To enable the Console Interface push the ENABLE switch to the right and reset the target machine to start the SCSI software.

1.2.2 Configuring the Console

This section describes how to configure the Console Interface unit via the two banks of DIP switches (labelled A and B). Setting up the Console Interface does not normally require changing the default DIP switch positions but they may need to be changed during the course of game development, for example when testing a binary image.

DIP switches

Bank A configures the start address of the console interface RAM and bank B sets the target SCSI device number and determines how the SNASM68K software controls the target. By default the RAM start address is 0 (for a 1MB system all switches are *on*, for a 2MB system SW1-7 are *on*, SW8 is *off*, for a 4MB system SW1-5 are *on*, SW6-8 are *off*).

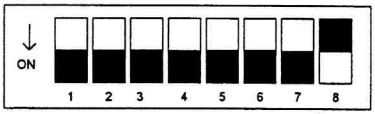


Figure 4. On a 2MB system the default positions for Bank A set the RAM start address to 0.

A full description of bank A DIP switches is given in Table 2 below. If, for example, switches 4 and 8 are moved to the up position the main RAM will reside at 1MB so the unit can be operated with target RAM or EPROM in the lower 1MB with the main unit enabled. When using this facility be very careful to setup the switches and insert the cartridge before powering up either device.

Switch	Description	Address Line
SW1	Start 1MB address of main RAM	A23
SW2	Start 1MB address of main RAM	A22
SW3	Start 1MB address of main RAM	A21
SW4	Start 1MB address of main RAM	A20
SW5	End 1MB address of main RAM	A23
SW6	End 1MB address of main RAM	A22
SW7	End 1MB address of main RAM	A21
SW8	End 1MB address of main RAM	A20

Table 2. Bank A DIP switches.

The default DIP switch positions for Bank B are shown in Figure 5 below. By default the target SCSI device number is 7 (SW1-3 *on*), the Console Interface ROM's take control of the target boot up sequence (SW6 *on*) and control all I/O between the main interface and the interface unit (SW7 *on*). SW4-5 and SW8 are not used.

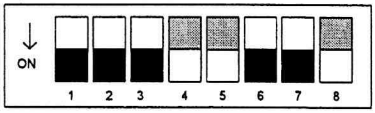


Figure 5. Default DIP switch positions for Bank B.

To watch a program run as if from a cart, download the ROM image to the target, set SW6 to *off* and press the RESET button on the target machine.

1.2.3 Connecting the Console Interface Unit

To connect the Console Interface Unit:

1. Align the notches on the 68000 sockets of the target and the interface unit (identified on the interface unit by a coloured circular label on the case) and carefully insert the interface unit into the socket of the target interface. Severe damage to both devices can result from incorrect orientation.
2. Connect the ribbon cable between the main unit and the interface unit. Plug the SCSI cable into both the PC and the Console interface.
3. Deactivate or remove any RAM or EPROM that may occupy the lower 4MB on the target.
4. Push the ENABLE switch on the front panel of the main unit to the right.
5. Connect the power supply first to the main unit and then to the target interface.
6. Turn the power on first to the Console Interface unit and then to the target. The WR PROT light on the main unit should light briefly and then go off. The ACTIVE, PWR ON and DC PWR lights should be lit indicating that communications between the console interface and the PC are now possible and can be tested using the SNTEST utility.

This completes the Console Interface Unit setup.

1.3 Installing the Software

Before You Start


Note

The SNASM2 install program is designed to customise the environment of one of the supported text editors, enabling you to run SNASM2 from within the editor. If you intend working in this way must install one of the supported text editors before installing the SNASM2 software.

The SNASM2 software requires approximately 1.5MB of free disk space. To successfully complete the software installation you will also need to know the SCSI device number of the PC Card and the address in memory at which the card is to reside.

Finally, remember to make backup copies of your disks before installing the software. Keep the original disks in a safe place and install the software from the backup copies.

Files Changed by the INSTALL Program

AUTOXEC.BAT — The install program may add a line similar to the following:

```
SCSILINK 390
```

Installing the Software

To install the SNASM2 software:

1. Insert the SNASM2 disk in a floppy drive.
2. At the command prompt, type the letter of the drive you're using, followed by :INSTALL and then press Enter. For example

```
C:\>a:install
```

3. Follow the instructions on screen. You can abort the installation at any time by pressing the Esc key.

The SCSILINK utility is described on page 201.

The final stage of the installation runs a utility called SCSILINK that checks the functioning of the PC Card. If the card has been successfully installed SCSILINK will display a message similar to the following

```
PC's SCSI ID=6,        Timer Version 1.00
Snasm2 card has passed initial tests and the drivers
↪are now installed.
```

See *Troubleshooting* below for help on correcting any errors with the PC Card.

This completes the software installation. You should reboot your computer so that changes to your system can take effect.

1.4 Exception Vectors

The 68000 vectors are setup using the file STARTUP.68K on the SNASMZ80 disk and should be placed before any other code in your program. The value of the `KeepTraps` variable determines whether the vectors point to handlers in source code or to error handlers in the Console Interface ROM and should be set at the start of your code. The file EXCEPT.68K contains the exception label definitions which, in a finished product, are placed at the same address with a piece of code to cause a reboot. To use an interrupt in your code remove the appropriate label for the exception from the list.

The software in the Console Interface provides two services, called from within your code, that provide further control over the debugging environment. The entry point for these services is 16 bytes into the interface workspace requiring a JSR to the base address plus 16 bytes to use them. The service number is held in D0 and the other registers are as described below.

Service 0 - Handle More Exceptions

Register	Contents
D1	Vector number at which to start.
D2	Number of vectors to change.

Service 0 causes further execeptions to switch execution to the SCSI software. The following piece of code allows the SCSI software to be entered by patching traps 1-F.

```
moveq   #0,d0       ;user service 0
move.l  #33,d1      ;start at offset 33(trap1)
move.l  #15,d2      ;set 15 vectors

jsr     $208000+16  ;$400000+16 on 4MB unit
bne     Error       ;zero flag set on success
                    ;could also check for d0 set to 0
```

Service 1 - Change Status Register Masks

Register	Contents
D1	Low word = ExSROrMask. High word = ExSRAndMask.
D2	Low word = TpSROrMask. High word = TpSRAndMask.

On entry to the exception handler routine, the flags ExSRAndMask and ExSROrMask are respectively ANDed and ORed with the status register. This enables the status register to be manipulated by setting these flags prior to an exception. Similarly, the flags TpSRAndMask and TpSROrMask perform the same function upon the execution of a trap zero. By default these flags are initialised on entry to the downloader to ensure that all interrupts are switched off when the downloader is invoked.

To produce the binary image of the final ROM set the KeepTraps variable to False and assemble the code with pure binary mode set. The SSP and PC should also be setup as, when the code is in ROM, the lower 8 bytes of the memory map are moved into the SSP and PC at reset. Also, during testing the initial values of the SSP and PC should be set using the REGS directive.

The REGS directive is described on page 67.

1.5 Troubleshooting

If the PC Card isn't functioning correctly, SCSILINK will display a message similar to the one below, the actual message will depend on the type of failure.

```
Card configuration ........Test Passed
Latch operation ...........Test Passed
TermPWR fuse ..............Test Passed
SCSI chip presence ........Test Passed
On-board RAM ..............Test Failed
Timer .....................Test Failed
There was a problem with the card so the drivers
↪haven't been installed.
```

The following table provides possible remedies (if any) for each possible failure. If, after trying to correct the failure, the card still does not function correctly please contact technical support for a replacement.

Test Failed	Reasons	Remedy
Card configuration	You have invoked SCSILINK with an address different to that set on the card.	Check the switch settings on the PC Card.
	There is a conflict with another card	Set the PC Card to a different address as described on page 3
Latch operation	You have invoked SCSILINK with an address different to that set on the card.	Check the switch settings on the PC Card.
	There is a conflict with another card.	Set the PC Card to a different address as described on page 3
TermPWR fuse	The fuse may have blown or become dislodged.	Replace the fuse using the supplied spare.

Continued on next page.

Continued from previos page.

Test Failed	Reasons	Remedy
SCSI chip presence	The SCSI chip has failed or become disconnected.	contact technical support for a replacement.
On board RAM	The RAM has failed or become disconnected.	Contact technical support for a replacement.

Table 3. Troubleshooting the PC Card.

The Assembler

The assembler translates assembly language source files into binary files which can be loaded into memory and executed. SNASM68K is a one-pass assembler with a sophisticated patch-back system, able to handle all forward references including forward referenced equates. Source statements are processed to produce a relocatable object file in the industry standard COFF file format, allowing mixed language projects.

The linker is fully integrated into the assembler to produce a 'linking assembler' that loads the required modules and resolves external references to produce a final loadable output or an object module for further linking. This provides flexibility over which parts of code are assembled into object files, those assembled on every build and those stored in libraries. The assembler also has a vast superset of features found in other assemblers including:

- Rationalised syntax whilst maintaining full backwards compatibility.
- Multiple processor support.
- Binary includes of files or file subsets.
- A partial expression evaluator; the link software includes a full expression evaluator.
- Extensive group attributes including specific support for ROM image generation.
- Flexible Macros with parameter list handling.
- Comprehensive conditional assembly structures
- Optional code optimisation.
- Map file showing the size and location of sections, groups and symbols in memory and also the amount of room left in groups.
- Informative listings

This part describes the assembler in detail, covering:

- Running The Assembler
- Source Code Syntax
- Assembler Directives
- Macros
- Sections and Groups

2 Running The Assembler

This section shows you how to run the linking assembler from the command line or from within one of the supported text editors.

2.1 Stand-alone Assembler

This section shows you how to invoke the assembler from the command line and describes all the switches, options and optimisations available from the command line. The options and optimisations can also be set from within the assembly code source file using the OPT directive described on page 83.

2.1.1 Command-Line Syntax

This section shows how to invoke the assembler from the command line and describes all the switches used to control the assembler options and optimisations. The command line consists of a series of optional switches, separated by white space, followed by the names of files to be used during the assembly process. The syntax is:

`snasm68k [-|/]` *Switches SourceFile,ObjectFile,Mapfile,ListFile,TempFile*

To halt the assembly type Ctrl+C or Ctrl+Break. This deletes any temporary files and does not generate any output files if the assembly failed.

The assembler can accept multiple source files using the format '*Filename+Filename+...*', treating each file as if it were an include. Any resulting map or listing files are concatenated to produce a single file of each type. Note that when assembling in this way you *must* specify the source file extensions so that the assembler can, for example, differentiate between 68000 source files and COFF files.

The assembler can also download object code direct to the target or simultaneously create a COFF file and download object code to the target.

Example 1

The following example assembles the source file TEST.68K and outputs the object code to TEST.COF. The assembler also generates a map file, TEST.MAP, but produces no listing or temporary files.

```
snasm68k   test.68k,test.cof,test.map
```

Example 2

The following example assembles two source files, TEST1.68K and TEST2.68K, and the object file TEST1.COF. The resulting object code is sent to TEST.COF and produces a single map file, TEST.MAP.

```
snasm68k test1.68k+test2.68k+test1.cof,test.cof,test.map
```

Example 3

This example is the same as the first except that the object code is sent to target 7.

```
snasm68k   test.68k,t7:,test.map
```

Example 4

This example assembles the source file TEST.68K, downloads the object code to target 7 and generates a COFF file, including source debug info (`/sdb`), called TEST.COF but does not run the code (`/d`).

```
snasm68k /d /sdb  test.68k,t7:test.cof
```

The debugger command -line is described on page 137.

To subsequently enter the debugger with the debug info use:

```
snbug68k -t7:test.cof
```

2.1.2 Filenames

The files used by the assembler and their default extensions are given in the table in the following page. Note that source files can also take any extension specified during the installation process.

Filename	Extension	Description
SourceFile	.68K, .ASM	This file contains the 68000 source code to be assembled. If no sourcefile is specified the assembler will print a help message and a description of the command-line syntax.
ObjectFile	.COF	This file will receive the object code output. If the object code is to be sent to a target the filename has the format 'T*n*:' where *n* is the SCSI device number of the target. If no file is specified then object code will be not be generated unless group FILE statements are present. (See page 129 for more information about group attributes.)
ListFile	.LST	File to receive any listing output.
MapFile	.MAP	This file contains information about symbols and the length, location and attributes of groups and sections. In addition, all files used by the assembler in the current run are listed, indicating which files have been read and written and how they were used. The current directory at the time of assembly is also displayed.
TempFile	.TMP	This is the location to be used for any data that will not fit in memory whilst assembling. A temporary file will only be created if the project is too large to fit in memory and the file size will be reported by the assembler. Assembling in this way is slower than using RAM but at least makes an assembly possible. The location of the temporary file may be a drive, a path or a full filename. If no location is specified then the assembler will look for TMP or TEMP in the environment variable. Failing this the data will be stored in a uniquely named file.

Table 4. Assembler filenames and default extensions.

2.1.3 Switches

The table below describes the switches available from the command line. To identify a switch to the assembler it must be preceded by a '/' or '-'.


Note

There must be at least one space between the switch and any parameters and that this is not compatible with previous versions of the assembler.

Switch	Description
?	Brings up on-line help describing the syntax for switches, options and optimisations.
b *Size*	Set the *Size* of the input buffers in KBytes from 1-64, the default being 16K. Note that there must be at least one space character between b and *size*.
coff	Change between big endian and little endian COFF output file.
d	Debug mode only; assemble the code but do not run it.
e *Symbol*=[*Val*|"*Str*"]	Equate a symbol to a value or a string. The symbol will be set to 1 if no value or string is specified. Multiple equates are separated by semicolons (';').
emax *NumErrors*	Abort the assembly after the number of errors exceeds that specified by *NumErrors*. The default value of *NumErrors* is zero which will not abort the assembly.
g	Write non-global symbols to linker object file.
hex *Number*	Set the width of hexadecimal output in the listing file from 2-8 words, the default being 4.
i	Display information window during assembly.

Continued on next page.

Continued from previous page.

Switch	Description
j *Dir* [;*Dir*]...	Specify the search directory for INCLUDE file. If an INCLUDE filename does not specify a path, by default the assembler first looks for files in the current directory. If it cannot be found there the assembler looks in the directories built up using the 'j' switch.
k	Enable additional conditional assembly structures. These are implemented via macros and are described on page 111.
l	Produce linkable output file.
lnos	Show source code line numbers in the listing file.
m	List external symbols to map file (linking only).
o *Options*	Set assembler options and optimisations. Note that there must be at least one space character between the switch and the parameter.
p	Produce pure binary output file.
q *Quirks*	Enable quirks. Quirks are special options that enable certain features specific to SNASM68K version 1.x.
sdb	Add source debug information to COFF file.
w	Write equates to symbol table.
z	Include line number information in linker (.COF) file.

Table 4. Assembler command-line switches.

2.1.4 Quirks

Quirks are known incompatibilities between Snasm2 and version 1.x. The behaviour of the assembler prior to version 2 occasionally became eccentric as more features were added to it. Snasm2 attempts to rationalise this behaviour but for backwards compatibility it is possible to introduce these 'quirks' into the assembler. These quirks may not be supported in future releases so it is strongly recommended that you check through your source code to track down anything using a quirk and change it to be compatible with version 2.

The command line quirks are described below. Do not use white space between the quirk name and the '+' or '-' and separate multiple quirks with commas.

Quirk	Description
`fl-/+`	*Functions in Lower Case* Specify names of functions and pre-defined constants in lower case if the case sensitivity option is enabled.
`mc-/+`	*Macro Continuation Character*. Allows the use of '\' as a line continuation character in macro calls and on the first line of a macro definition.
`mp-/+`	*Macro Parameter Lower Case*. Sets unquoted macro parameters to lower case if the assembler is set to be case insensitive.
`regs-/+`	*Registers*. Causes REGS statements to be added to the default group instead of the currently opened section.
`sa-/+`	*Section Alignment*. Aligns a section re-opened without a size modifier to the previously defined alignment for that section. This applies to both the SECTION and POPS directives.

Table 5. Assembler command-line quirks.

2.1.5 Assembler Command Files

A command file contains assembler command line parameters separated by white space or line breaks. The command file can then be used in place of or in addition to command line parameters. If you invoke the assembler with both switches and a command file, the switches take precedence over the contents of the command file (which should be the last item on the command line) as they can be set once only. The filename has the extension .CMD and must be preceded with the '@' symbol to tell the assembler that it is a command file although this does not form part of the filename itself. Comments can be introduced with '*' or '#' characters at the beginning of a line or with a ';' character anywhere in the command file.

3 Source Code Syntax

The assembler supports the standard Motorola 68000 mnemonics and will generate code for supported extensions of the instruction as determined by the addressing mode. The extensions supported by the assembler are given in the table below.

Generic	Extension
ADD	ADD, ADDA, ADDI
MOVE	MOVE, MOVEA, MOVEM

Table 6. Supported instruction extensions.

3.1 Statement Format

Each statement has the following general format:

```
LABEL    MNEMONIC     OPERAND(S)     COMMENT
Start    lea.l        Stack,SP       ; Initialise stack
```

Each field must be separated by *white space* - any combination of tabs and spaces. To improve code readability, white space is allowed in the operand field after a comma or operator. White space following the operand(s) denotes the beginning of the comment field which is ignored by the assembler. To avoid confusion it is recommended that comments start with a semi-colon (;). The assembler can be made to insist on a semi-colon before comments using the *White Space* option (see page 85). Setting this causes the assembler to ignore white space in the operand field and is a highly recommended option.

The exception to the statement format is comment lines which begin with a semi-colon or an asterisk. The comment line can begin in any column if it begins with a semi-colon but must start in column 1 if it begins with an asterisk. Blank lines and lines that contain white space only are treated as comment lines. All comments are ignored by the assembler.

The maximum line size is 1024 characters. If this is insufficient, use the line continuation character '&' to carry on to the next line. In a macro you can also use the '\' character but in either case the line continuation character

must be the last symbol on the line. The '&' character can be used in comments but will be interpreted as an ampersand and not as a line continuation character.

Example

```
   opt   ws+                ;White space allowed in operand
                            ;Comments must begin with ';'

   dc.b  'ab' ,'cd',  'ef'  ;Spaces don't end operand field
   dc.b  'gh','ij',  'kl'

   opt   ws-                ;Turn off whitespace option

   dc.b  'ab' ,'cd',  'ef'  Whitespace starts comment
   dc.b  'gh','ij',  'kl'   But is OK after a comma

;Note that whitespace is always allowed in HEX strings
;at byte boundaries.

   opt   ws+                ;Turn on whitespace option

   hex   01 02 03 04        ;Comments must begin with ';'
   hex   0a0b 0c0d

   opt   ws-                ;Turn off whitespace option

   hex   01 02 03 04        First whitespace ends number
   hex   0a0b 0c0d          And here
```

3.2 Labels and Symbols

Symbols can contain up to 48 characters from the following symbol set:

A - Z a - z 0 - 9 ? _ (underscore) . (period)

If a symbol is longer than 48 characters, it will be truncated and the assembler will generate a 'Label Truncated' warning. The first character of a symbol must not be a number except for local labels. Illegal characters such as non-printing characters will generate an error at assembly time unless they appear in a comment in which case they are ignored.

3.2.1 Labels

Unless defined otherwise, labels are global, that is they are known to the whole program. Symbols that are used as labels become symbolic addresses for actual locations in the program. Labels are optional for all assembly language instructions and for most assembler directives. If used, a label *must* start in column 1 unless it ends with a colon (:). The colon is not treated as part of the label. Global labels *must* conform to the following format:

First Character	Subsequent Characters
A - Z, a -z, _	A -Z, a - z, 0 - 9, _, ., ?

3.2.2 Local Labels

The assembler supports local labels which begin with a special local label character but this does not form part of the label itself. Local labels are labels that are declared local to a particular range of source code. They exist only within this range, known as their *scope*, and can be re-defined outside this area. This makes local labels useful for loop counters and markers. The assembler provides a number of ways to control the scope of local labels, described in the section on Scoping Local labels on page 26.

See also Symbols and Periods on page 28.

Local labels *must* confirm to the following format with the first character being *one* of the special local label characters specified below. The label itself can be any valid label and begin with a digit. Unlike most other assemblers you can also use `.w` and `.l` as local labels but this is not recommended.

First Character	Subsequent Characters
@, ., :, ?, |, !	A -Z, a - z, 0 - 9, _ (underscore), . (period), ?

The default local label character is '@' but this can be redefined using the assembler option. The local label character can be redefined using the Local Label Character option, either from the command line (`1+` to toggle between '@' and '.' or `1` *Character* to use one of the other characters given above) or with the OPT directive. If the local label character is changed, from '@' to '.' for example, then local labels previously defined using '@' can then only be referenced using '.' as the local label character.

Example

```
@Alert                  ; @ is the default
            ...
            opt   1 63  ; Change to ? (63 ASCII)
?LevelOne
?Alert                  ; Same label as before
            opt   1 '@' ; Change back to @
@Alert                  ; Still same label
```

3.2.3 Scoping Local Labels

The assembler provides extensive support for controlling the scope of local labels from the simple *between non-locals* form to the more sophisticated concept of *modules* (modules and the scope of local labels within them are described on page 81). Local labels and modules are also available inside macros with further macro specific facilities provided by the '\@' parameter and the LOCAL directive.

See also macros on page 101.

The between non-local labels form of scoping lets you define local labels using the local label character only. The scope of the local label then extends from the previous non-local label up to but not including the next non-local label. If the *descope local labels* option (`d+`) is enabled then the EQU, EQUR, EQUS and SET directives act as non-local labels as well as causing local labels to be descoped.

Example

In the code below, the scope of the first `@NoInc` label extends between the `IncNzD0` and `IncNzD1` labels. The `@NoInc` label can be re-defined after the next non-local label which is `IncNzD1`. The scope of the second `@NoInc` label is then from `IncNzD1` to the next non-local label.

```
IncNzD0     tst.w     d0
            bne.s     @NoInc
            addq.w    #1,d0
@NoInc      rts
IncNzD1     tst.w     d1
            bne.s     @NoInc
            addq.w    #1,d1
@NoInc      rts
```

3.2.4 Pre-defined Symbols

There are a number of symbols that are pre-defined and updated by the assembler. Some of these are standard internal symbols and the rest are mainly to help you keep track of version numbers and dates. Unlike many other assemblers you are free to re-define any symbol already encountered by the assembler, including assembler instructions and register names. Note that case is not important when using these pre-defined symbols so there is no difference between, for example, `_radix` and `_RADIX`. This means that you cannot define a symbol `_RADIX` even if `_radix` has been aliased.

See also ALIAS on page 44.

Symbol	Description
*	Current value of the assembly program counter, evaluated at the beginning of the line.
@	Current value of the assembly program counter, evaluated during the line.
__rs	Current value of the RS counter i.e. the current offset into a structure. Note that this function begins with a *double* underscore.
_radix	Current value of RADIX directive.
_rcount	Current iteration count, starting at 1, in a repetitive statement.
_year	The year at the beginning of assembly.
_month	The month at the beginning of assembly.
_day	The day at the beginning of assembly
_weekday	The weekday at the beginning of assembly.
_hours	The hour at the beginning of assembly.
_minutes	The minute at the beginning of assembly.
_seconds	The second at the beginning of assembly.

Continued on next page.

Continued from previous page.

Symbol	Description
`_filename`	The name of the root file.
`_current_file`	The file being assembled.
`_current_line`	The current line of the assembly.
`narg`	The number of parameters passed to the current macro.

Table 7. Pre-defined symbols.

Note

The 24 hour time and date symbols are set at the beginning of the assembly and do not change. To put the date and time into a string use the \# parameter described later in the chapter on Macros.

Example

```
AsmDay      dc.w    _day
AsmMonth    dc.w    _month
AsmYear     dc.w    _year
```

3.2.5 Symbols and Periods

The assembler allows the use of symbols containing periods, providing greater flexibility in choosing label names. However, this is not recommnded as there can sometimes be confusion as to whether a period is a size modifier or part of a label.

To explain how symbols and periods are handled it is necessary to describe the line pre-processor employed by the assembler. This is best illustrated using an example so consider the following source statement:

```
move.l   length.w,d0
```

This statement alone does not provide enough information for the assembler to tell if `length.w` is a label '`length.w`' or a label '`length`' with a word modifier. To determine the correct semantic the assembler looks first to see if `length.w` is a label. If so then no further action is needed; otherwise the string is parsed from right to left. The first character encountered is a '`w`' which does not reveal anything. The second character is a '`.`' which could mean that '`.w`' is a size modifier and so it is stripped from the string. The assembler now looks again to see if the remaining characters in the string, `length`, constitute a label. If this is the case then no further analysis is

required, otherwise the process is repeated until either a label if found or an 'undefined label error' is generated.

To explicitly state that `length` is a label enclose it in brackets:

```
move    (Length).w,d0
```

Alternatively, use a backslash ('\') in place of the period for the size modifier.

```
move    Length\w,d0
```

3.3 Constants

The assembler supports four basic types of constants:

- Integer constants
- Character constants
- Pre-defined constants
- Assembly-Time constants

3.3.1 Integer Constants

The assembler supports integer constants in any base from 2 to 16, the value of which must not exceed 32 digits. Integer constants must begin with a decimal digit with the exception of hexadecimal and binary constants which are prefixed with special characters. The default base is set using the RADIX directive. Initially it is set to 10. For numbers in bases 11-16, the characters A - F (or a - f) are used to represent the digits 10 - 16 respectively.

The assembler also supports other notational forms for certain integer constants. There a RISC style format for integer constants which is of the form *r_nnn* where *r* is a radix from 2 to 9 and *n* a valid digit. In addition, hexadecimal numbers can be specified using the '`0x`' C language notation. If the *alternate numeric* option is enabled then you can use Intel style suffixes to denote the radix. The integer constant, which must begin with a valid digit for that constant, is suffixed with the letters H, D, Q or B to specify the radix as Hexadecimal, Decimal, Octal or Binary respectively. This does not work if the default radix is greater than 10 as B and D are valid hexadecimal digits.

All integer constants must begin with a decimal digit regardless of the default base. Examples of valid constants are:

`8_123`	Constant equivalent to 123 octal (83 decimal)

Hexadecimal constants are prefixed with the '$' or '0x' characters and include the decimal values 0-9 and the letters A-F and a-f. Examples of valid hexadecimal constants are:

`0xA0`	Constant equivalent to 160 decimal
`$73E`	Constant equivalent to 1854 decimal

Binary constants are prefixed with the % character. Examples of valid binary constants are:

%11010000000	Constant equivalent to 1664 decimal
%11	Constant equivalent to 3 decimal
2_11	Constant equivalent to 3 decimal

3.3.2 Character Constants

A character constant is a string of up to 4 characters enclosed in either single or double quotes (' ' or " ") quotes. The characters are represented internally as 8 bit ASCII characters. Single or double quotes are represented by specifying the character twice or by delimiting one type of quote with the other. Control characters can be represented by preceding the control character with backslash caret (\^). Examples of valid character constants are:

'a'	Represented internally as 00000061 hexadecimal
'abc'	Represented internally as 00616263 hexadecimal
'"a'	Represented internally as 00002261 hexadecimal
"""a"	Represented internally as 00002261 hexadecimal
"'a"	Represented internally as 00002761 hexadecimal
\^M	Represented internally as 0000000D hexadecimal

Note the difference between a character *constant* and a character *string*. A character constant is an integer *value* and a character string is a list of characters. Character strings are described later.

3.3.3 Assembly Time Constants

The EQU directive can be used to assign a value to a symbol. The symbol then becomes a constant for the duration of the assembly. The value does not have to be absolute to be used in expressions. If the expression contains forward references then the symbol takes the value of the expression at the end of the first pass.

Example

```
val         equ  3
            movi #val, A0
```

3.3.4 Turning Numbers into Strings

The \# and \$ parameters can be used to substitute the decimal and hexadecimal value respectively of a symbol into your source code. They are used to turn numbers into strings for formatting data or building arrays of symbols and so on.

Example 1

```
Col0        equ     $FFF
Col1        equ     $F0F
Col63       equ     $0FF
Col99       equ     $FF0

Index       =       0
            dc.w    Col\#Index      ; expands to Col0
Index       =       1
            dc.w    Col\#Index      ; expands to Col1
Index       =       99
            dc.w    Col\#Index      ; expands to Col99
            dc.w    Col\$Index      ; expands to Col63

; The words DC'd will be $FFF,$F0F,$FF0 and $0FF.
```

Example 2

```
; Put the data and time into a string
AsmDate     dc.b    '\#_day/\#_month/\#_year'
; expands to
AsmDate     dc.b    '1/4/1993'
```

3.3.5 Current Location Counter

During assembly, the assembler keeps a variable that always contains the start address of the current line. This variable is known as the Location Counter and is represented by an asterisk ('*').

Example 1

```
MyString        dc.b    'Hello World'
MyStringLen     equ     *-MyString
```

The @ operator is similar to the '*' operator except that where '*' is the program counter at the start of the line, @ is advanced during the line. It can be used only with DC and DCB directives, usually in expressions to determine the value of the current PC.

Example 2

```
                dc.l        @,@,@
; The three longs will be 4 bytes apart
                dc.w        Fred-@,Start-@
; The offsets are from the current word
```

Example 3

The location counter can express the idea that *=*address of myself.* The location counter contains the value $8000 and the instruction will be translated into a relative jump to address $8000 from address $8000 i.e. *jump to myself.* This is useful when waiting for an interrupt but be careful you do not end up in an infinite loop.

```
        org     $8000
        jmp     *
```

3.4 Strings

See also macros on page 101.

A character string is a list of characters enclosed in singe or double quotes (' ' or " "). Strings may be used only in DC.B, EQUS and INFORM statements, as arguments to string functions and as macro parameters.

Quotes are used only to identify the string to the assembler and do not form part of the string itself. To put a single quote in a string the quote should appear twice or the whole string delimited with double quotes. Similarly, to put a double quote in the string the quote should appear twice or the whole string delimited by single quotes.

Example

```
dc.b    "a single quoted 'string'"
dc.b    'a double quoted "string"'
```

3.5 Expressions

An expression is a sequence of one or more constants and symbols separated by operators. The maximum number of symbols in an expression is dependent only on the maximum line size which is 1024 characters. Embedded blanks are allowed in expressions, after mathematical operators for example. The comparison operators =, >=, etc. return -1 if the comparison is True and 0 if the comparison is False. If a string equate or macro parameter is used in an expression there is no need to precede it with a backslash.


Note

Care should be taken when using large large numbers in expressions. The assembler uses a 32-bit expression evaluator so bit 31 must be set to negative as well as bit 23.

Example

```
MinusOne        equ     $FFFFFFFF   ;negative
NotMinusOne     equ     $FFFFFF     ;positive
```

3.5.1 Operator Precedence

The operators supported by the assembler are given in the table below. They are listed in decreasing order of precedence with operators of equal precedence evaluated from left to right. The precedence can be overridden by the use of parentheses as these have the highest precedence. To increase the clarity of complex expressions it is recommended that they be parenthesised.

Precedence	Operators
Highest	( )
	+,-,~ (unary)
	<<, >>
	&, ! or |, ^
	*, /, %
	+, - (binary)
Lowest	=, <, >, <=, >=, <>

Table 8. Operator precedence.

3.5.2 Functions

The assembler provides a set of functions providing useful information about symbols, strings, section and group sizes and start addresses, offsets and others. Note that all addresses refer to assembly addresses and not the physical address unless stated otherwise.

ADDRMODE

`addrmode(`*Expr*`)`

See also the special macro parameter _ on page 108.

The ADDRMODE function allows a macro to determine the addressing mode that an instruction will use allowing the additional structured assembly macros to be implemented.

Addressing Modes

Mode	Description	Example
0	Data register	`addrmode(d3)`
2	Address register	`addrmode(sp)`
4	Indirect	`addrmode((a3))`
6	Indirect post increment	`addrmode((sp)+)`
8	Indirect pre-decrement	`addrmode(-(a0))`
10	Displacement	`addrmode(10(a0))`
12	Displacement with index	`addrmode(2(a0,d0.w))`
14	Absolute word	`addrmode(fred\w)`
16	Absolute long	`addrmode((fred+2).l)`
18	Displacement off PC	`addrmode(lab(pc))`
20	Displacement off PC with index	`addrmode(10(pc,d0.l))`
22	Immediate	`addrmode(#fred+4)`

Table 9. Addressing modes used by the ADDRMODE function.

ALIGNMENT

`alignment(`*x*`)`

The ALIGNMENT function returns the offset of its argument from the sections alignment type (byte, word or long). In a byte aligned section ALIGNMENT(X) will always return 0, in a word aligned section it will return 0 or 1, and in a long word aligned section 0..3. This function is usually used to check if the PC is odd or even.

Example

```
if alignment(*)&1       ;If PC is odd pad with
  dc.b 0                ;zero to even boundary
endif
```

DEF

`def(`*Symbol*`)`

The DEF function checks to see if *Symbol* has been defined.

FILESIZE

`filesize(`*Filename*`)`

The FILESIZE function returns the size of *Filename* or -1 if *Filename* does not exist.

Example

```
Size    =       filesize(main.68k)
        if (size=-1)
        inform 3, "File not found"
        endif
```

GROUPEND

`groupend(`*GroupName*`)`

The GROUPEND function returns the end address of the group *GroupName*, evaluated at link time.

GROUPORG

`grouporg(`*GroupName*`)`

The GROUPORG function returns the physical address at which the group *GroupName* has been placed. This is mainly used to determine where the group is in memory so that it can be relocated.

GROUPSIZE

`groupsize(`*GroupName*`)`

The GROUPSIZE function returns the current size of the group *GroupName*. It is evaluated immediately and so reflects the current group size not the final size.

INSTR

`instr(`[*Expr*]`,`*String*`,`*SubString*`)`

The INSTR function performs a case sensitive search on the string *String* and returns the starting position of the sub-string *SubString*. The position in the string to start searching for the sub-string can be optionally specified with the expression *Expr*.

INSTRI

`instri(`[*Expr*]`,`*String*`,`*SubString*`)`

The INSTRI function performs a case insensitive search on the string *String* and returns the starting position of the sub-string *SubString*. The position in the string to start searching for the sub-string can be optionally specified with the expression *Expr*.

NARG

`narg(`*List*`)`

The NARG function returns the number of items in a parameter list.

OFFSET

`offset(`*x*`)`

The OFFSET function returns the offset of its parameter from the base of the section in which it is defined. It is not evaluated until link time so if you require the offset into the current modules contribution to a section you will need to place a label at the start of the section and do the subtract yourself. Use OFFSET(*) to get the offset of the current PC.

REF

`ref(`*Symbol*`)`

The REF function checks to see if the symbol *Symbol* has been referenced but not defined.

SECT

`sect(`*Name*`)`

The SECT function returns the base address of a section. *Name* can be a section name, label or * (the PC symbol). SECT is not evaluated until link time.

SECTEND

`sectend(`*SectName*`)`

The SECTEND function returns the end address of the section *SectName*, evaluated at link time.

SECTSIZE

`sectsize(`*SectName*`)`

The SECTSIZE function returns the current size of the section *SectName*. It is evaluated immediately and so reflects the current section size not the final size.

SQRT

`sqrt(`*Expr*`)`

The SQRT function returns the truncated integer square root of an expression *Expr*. *Expr* must evaluate to an integer with all negative arguments returning -1.

STRCMP

`strcmp(`*String1*`,`*String2*`)`

The STRCMP function performs a case sensitive comparison of two strings *String1* and *String2*.

STRICMP

`stricmp(`*String1*`,`*String2*`)`

The STRICMP function performs a case insensitive comparison of two strings *String1* and *String2*.

STRLEN

`strlen(`*String*`)`

The STRLEN function returns the length of the string *String*.

TYPE

`type(`*Symbol*`)`

The TYPE function returns the type of a symbol. If the argument isn't the name of a symbol it returns 0, not an error. The TYPE function is useful in macros where you can use it to determine what has been passed to the macro as a parameter. TYPE returns a word with the bits having the meanings described in the table below. To check specific bits returned by the TYPE function use the bitwise 'and' operator '&' as in the example following the table.

Bit	Description
0	Symbol has an absolute value.
1	Symbol is relative to start of a section.
2	Symbol as defined using the SET directive.
3	Symbol is a macro.
4	Symbol is a string equate.
5	Symbol was defined using the EQU directive.
6	Symbol was specified in an IMPORT statement.
7	Symbol was specified in an EXPORT statement.
8	Symbol is a function.
9	Symbol is a group name.
10	Symbol is a macro parameter.
11	Symbol is a short macro.
12	Symbol is a section name.
13	Symbol is absolute word addressable.
14	Symbol is a register equate.
15	Symbol is a register list equate.

Table 10. Symbol types.

Example

```
; Check bit 9
if          (type(\1)&$200)=0
  inform    3,'%s is not a group name','\1'
endif
```

This is the only information contained on this page.

4 Assembler Directives

The assembler supports an extensive range of pseudo-mnemonics called Directives. These supply program data and control the assembly process. In particular they allow you to:

- Define symbolic names for constants and variables
- Define initialised data
- Reserve memory blocks for uninitialised data
- Control listings output
- Include other files
- Assemble conditional blocks.
- Assemble code into specified sections.
- Generate Errors and Warnings

The first part of this chapter describes the directives according to function and the second part, the Directives Reference, is an alphabetical reference of the directives, including syntax information.
The topics covered in this section are:

- Changing Directive Names
- Equates
- Defining Data
- Changing The Program Counter
- Listings
- Including Other Files
- Setting Target Parameters
- Conditional Assembly
- String Handling
- Local Labels and Modules
- Sections and Groups
- Options
- User Generated Errors and Warnings

4.1 Changing Directive Names

4.1.1 ALIAS

Use the ALIAS directive to rename labels, the assembler's directives, pre-defined functions and constants. This is useful if any of the pre-defined constants or functions clash with your own or you are used to a different name. ALIAS can be used on any name already encountered by the assembler, including assembler instructions and local labels. ALIAS will define a new symbol to be used as an alias for an existing name but will not remove the current name from the symbol table. The new name is defined to be equivalent to the old name and may be used anywhere that the old name was valid. ALIAS is usually used in conjunction with the DISABLE directive (see below) to rename the pre-defined assembler symbols.

Syntax

NewName `alias` *OldName*

where

NewName is a symbol defined by this statement.

OldName is any symbol previously encountered by the assembler.

Example

```
_type   alias     type
        disable   type
```

4.1.2 DISABLE

Use the DISABLE directive to remove a name from the symbol table, provided it has already been encountered by the assembler. DISABLE can be used in conjunction with ALIAS (see above) to rename something rather than just providing an alias but care should be taken to alias the name before disabling it as otherwise it will be unavailable.

Syntax

`disable` *OldName*

where:

OldName is any symbol previously encountered by the assembler including assembler directives and instructions. *OldName* is effectively un-defined and becomes free to be re-used.

Example

```
root    alias       sqrt
        disable     sqrt
        dc.w        root(66)
```

4.2 Equates

Equates are used to assign a symbolic name to a value (a variable numeric value, constant, string, register name or register list). The symbol can then be used in place of a value in the assembly source code. If an expression in an equate cannot be evaluated immediately, the assembler substitutes the value at the end of assembly or at link time.

Equates enable you to assign meaningful names to constants and numeric variables which improves code readability and eases changes to the value of a constant. The assembler supports six types of equates which are described below.

4.2.1 EQU

The EQU directive is used to assign a symbolic name to a constant or the value of an expression. The symbol to the left of EQU is assigned the result of the expression on the right.

Symbol `equ` *Expression*

where:

Symbol is a symbol defined by this statement.

Expression is a numeric expression whose value will be assigned to the given label for the duration of the assembly.

Once a symbol has been assigned a value with EQU, any attempt to assign a new value to the same symbol will result in an error. However, assigning a symbol the same value is allowed (known as a *benign redefinition*) and usually occurs when an include file defines hardware locations for itself that are also used by the main program.


Note

SNASM68K, unlike most assemblers, allows the use of forward references in the expression to which the symbol will be equated. At assembly time, as much as possible of the expression is evaluated; the remainder is completed at the end of the first pass or deferred until link time.

Example

```
True          equ   1
IOPort        equ   $300
Entries       equ   8
EntryLength   equ   16
TotalSize     equ   Entries*EntryLength
```

4.2.2 SET

See also Labels and Symbols on page 25 and Sections and Groups on page 119.

The SET directive is used to assign a symbolic name to a variable. The symbol to the left of SET is assigned the result of the expression on the right. Unlike the EQU directive, a symbol defined with SET can have its value changed as and when required. As such, these variables are often used for loop counters and scratch variables in macros. The SET directive may also be written as '='.

Unlike EQU, the expression must not reference any undefined or external symbols. Additionally, the symbol does not inherit any information about the type of the expression. This means that variables defined in this way are not always the best choice in source code using sections.

Syntax

Label `set` *Expression*

where:

Label is a re-definable symbol defined by this statement.

Expression is a value that will be assigned to the given label until re-defined by another SET directive. The expression must not reference any external or undefined symbols and should not contain any forward references.

Example 1

```
FreeSpace     SET   0
;zero total free space
              ...
FreeSpace     =     FreeSpace+1024
;1024 bytes more free here
```

4.2.3 EQUS

String equates are used to define synonyms for string variables. The EQUS directive is used to assign a symbolic name to a string variable such as a copyright message or a scratch variable in macros.

The text to be assigned to the string variable is delimited by single or double quotes (' ' or " "). These quotes are used only to identify the string to the assembler and do not form part of the string itself. If there are no quotes the assembler assumes that the parameter is the name of a previously defined string equate. To put a single quote in a string the quote should appear twice or the whole string delimited with double quotes. To put a double quote in the string the quote should again appear twice or the whole string delimited by single quotes.

As symbols equated with the EQUS directive can be used anywhere in your code they must be preceded by a '\' character to let the assembler know that a string equate is following. If there could confusion as to where the parameter ends, use a second backslash. The exception to this in expressions where the string is automatically substituted for the string variable when the expression is evaluated. If the symbol cannot be found the assembler will not perform any string substitution and the string substitute construct, '\Version' for example, will be left in the source and may preoduce an error message.

The parameter to EQUS can also be enclosed in curly brackets ({}), the use of which is explained in the section on macros.

Syntax

Symbol `equs` {*String* | *StringEquate* | *ParameterList*}

where:

Symbol	is the symbol defined by this statement.
String	is a string enclosed in quotes.
StringEquate	is any previously defined string equate.
ParameterList	is a list of parameters (expressions or strings) separated by commas to be passed to a macro.

Example 1

```
single1    equs    'I''m ok you''re ok'
single2    equs    "I'm ok you're ok"
double1    equs    'They said "ok" and left'
double2    equs    "They said ""ok"" and left"
```

Example 2

```
Version    equs    'Demo version 2.0a 01/04/93'
           ...
           dc.b    '\Version'
; Expands to
;          dc.b    'Demo version 0.2a 01/04/93
```

4.2.4 RS Equates

RS equates are used primarily for global variables and data structures. They enable you to define lists of constant labels without using explicit numbers that would need to be changed if you decided to add or delete an item near the front of the lists. So, in a data structure you can define a set of labels as offsets without having to keep track of the offsets yourself. The RS, RSRESET, RSSET directives and the __RS internal variable enable you to do this.

The RS directive is used to define a label as an offset. If RS is used with no size modifier then it is equivalent to RS.W. The RSSET directive is used to set the RS counter to a particular value. This is useful if you want to start an offset at a value other than zero. The RSRESET directive is used to set the RS counter to zero at the start of each new structure. It can take an optional parameter which if present causes it to behave exactly like the RSSET directive. This is for compatibility only and its use is discouraged.

The assembler has an internal variable called __RS (note the double underscore) which is used to keep track of the current offset. When a symbol is defined with the RS directive, the value of __RS is assigned to the symbol and the counter advanced by the specified number of bytes, words or long words.

If the *automatic even* option is enabled then RS.W and RS.L set the __RS variable to the next even boundary before the operation is performed.

Syntax

```
          rsset          Count
          rsreset        [Count]
Label     rs[.Qualifier] Count
```

where:

Label is a symbol defined by this label.

Qualifier is an optional qualifier that can be:

.b	Byte data
.w	Word data
.l	Longword data

The RS directive operates as RS.W if no qualifier is specified.

Count is an expression that must evaluate to a value.

Example 1

```
                rsreset
FileHandle      rs.w    1       ; __RS=0
FileOpen        rs.b    1       ; __RS=2
FileName        rs.b    8+3     ; __RS=3
FilePos         rs.l    1       ; __RS=14
FileSpecSize    rs.b    0       ; =18 __RS is not
                                ; advanced here.
; We could have used
; FileSpecSize  equ     __RS
```

Example 2

In this example the RSSET operand is negative as the address register points 8 bytes into the data structure. We could have used RSRESET -8 but this is for compatibility only and its use is discouraged.

```
            rsset   -8
ObjXPos     rs.l    1
ObjYpos     rs.l    1
ObjFlags    rs.w    1
ObjSpeed    rs.w    1
```

Example 3

Assume you have a data structure consisting of a long word, a byte and another long word in that order. To make your code more readable and easier to update should the structure change, you could use lines such as

```
            rsreset
next        rs.l    1
flag        rs.b    1
where       rs.l    1
```

and access the code with lines like

```
            move.l  next(a0),a1
            move.l  where(a0),a2
            tst.b   flag(a0)
```

4.2.5 Register Equates

Register equates are used to define synonyms for registers to improve code readability. The EQUR directive is used to define a symbolic name (that may include periods) for a data or address register.

Syntax

Symbol `equr` *Register*

where:

Symbol is any symbol defined by this statement.

Register is any data or address register.

Example 1

```
            section Code

City        equr    a0
Street      equr    a1
Dude        equr    a2
...
From        equr    d0
To          equr    d1
Counter     equr    d2
Player      equr    d4
```

Example 2

```
            move.w      d0, Offs(a3,d2.w)
; Could be written as
Power       equr        d0
CarDataPtr  equr        a3
CurIndex    equr        d2
            ...
            move.w      Power,Offs(CarDataPtr,CurIndex.w)
```

Similarly, the REG directive is used to define a synonym for a list of data or address registers. A range of registers can be specified by separating names with a '-' character.

Syntax

Symbol `reg` *RegisterList*

where:

Symbol is a symbol defined by this statement.

RegisterList is a list of register names or symbols. Different types of register are separated by a forward slash ('/'). A range of registers can be specified by separating names with a dash ('-'). The register at the start of the range must be less than the register at the end of the range.

Example 1

```
SaveRegs    reg     d0-d7/a0/a2-a4/a6
```

Example 2

```
            movem.l     d0-d6/a0-a6,-(sp)
; could be written as
MainRegs    reg         d0-d6/a0-a6
            movem.l     MainRegs,-(sp)
```

4.3 Defining Data

The assembler provides a variety of ways to define initialised and uninitialised data. The DC directive is used to define constants in memory. The DCB directive is used to generate a block of memory containing a specified number of the same value. The HEX directive is similar to DC but is used to store hexadecimal data in memory. The DATA and DATASIZE directives are used to define constants larger than the maximum 32 bits. Uninitialised data is defined using the DS directive.These directives are now described in more detail.

4.3.1 DC

The DC directive takes a variable number of arguments and after evaluating them places the results in the object code in either byte, word or long format. The assembler will generate an error if the value of the expression cannot fit in the data size declared. If the *automatic even* option is enabled, constants are aligned on word boundaries for DC.W and DC.L before the operation is performed.

DC directives can have spaces after commas but not before, even if the *white space* option is disabled (i.e. a space or tab introduces a comment). To put a single quote in a string either double up the single quote or delimit the string with double quotes (" and "). To put a double quote in the string double it up again or delimit the string with single quotes. Remember that strings can be used only with the byte form of the DC directive.

Syntax

[*Label*] `dc`[.*Qualifier*] *Operand* [,*Operand*]...

where:

Label is an optional symbol defined by this statement.

Qualifier is an optional qualifier that can be:

`.b`	Byte data
`.w`	Word data
`.l`	Longword data

The DC directive operates as DC.W if no qualifier is specified.

Operand is the operand may be a quoted string or an arithmetic expression. The quoted string must not exceed the size declared in the DC except for DC.B where it can be any length, each byte being emitted separately. Expression evaluation is done in 32 bit arithmetic for all types. Unless the *truncate* option is enabled (t+) to disable expression overflow checking the assembler will generate an error if the expression value cannot fit in the data size declared.

Example 1

```
        dc.b    "Any size string"
        dc.w    "ab",$afff
        dc.l    $76543210,"abcd"
```

Example 2

```
Position        dc.w    -69,202
Line Length     dc.w    0
                dc.l    StringBuffer
Signature       dc.l    'APPL'
ExelD           dc.w    'ZM'
ErrorNum        dc.w    -1
ErrorStr        dc.b    'Maximum length exceeded',0
Dispatch        dc.l    Routine1,Routine2,Routine3
```

4.3.2 DCB

The DCB directive defines a constant block of memory. DCB takes two parameters, *Count* and *Value*. The *Count* specifies how many times the *Value* is to be repeated. The count must always be present and evaluate but if no value is specified then the default value of zero is used. If the count is followed by a comma but no value, an error message will be raised. If the *Automatic even* option is enabled (ae+), constants are aligned on word boundaries for DCB.W and DCB.L before the operation is performed.

Syntax

[Label] `dcb`[.*Qualifier*] *Count, Value*

where:

Label is any symbol defined by this statement.

Qualifier is an optional qualifier that can be:

`.b`	Byte data
`.w`	Word data
`.l`	Longword data

The DCB directive operates as DCB.W if no qualifier is specified.

Count specifies how many times the *value* should be repeated. *Count* must evaluate.

Value is the value to be placed in memory.

Example

```
dcb.b   100,63  ;100 bytes containing 63
dcb.w   256,7   ;256 words containing 7
```

Out of range parameters

Out of range parameters normally generate an error. This is different from many other assemblers that truncate out of range parameters to force them into range. The assembler truncates expressions only if the *truncate* option is enabled (`t+`) which disables expression overflow checking. The following examples all cause an error when the truncate option is not enabled (the default).

```
dc.w    70000           ; Greater than 65535
dc.w    -40000          ; Less than -32768
dc.w    260             ; Greater the 256
dc.w    -130            ; Less than -128
dcb.w   10,70000        ; greater than 65535
dcb.w   128,-40000      ; less than -32768
dcb.b   16,260          ; greater than 256
dcb.b   16,-130         ; less than -128
```

4.3.3 HEX

The HEX directive is similar to DC but is used to store hexadecimal data in memory. HEX takes as its parameter a string containing an even number of hexadecimal digits, paired up to give bytes. White space is allowed in the string at byte boundaries, providing that the *white space* option (ws+) has been set.

Do not use the HEX directive for large amounts of data as it is a less efficient storage method than binary files. It is also slower to read and assemble and is not very readable. A more efficient method is to store the data as bytes in a file and use the INCBIN directive described on page 66.

Syntax

`hex` *HexString*

where:

HexString is a list of hexadecimal nibbles which are paired to give bytes. Enabling the *white space* option (ws+) allows you to insert spaces in the hexadecimal string at byte boundary positions.

Example

```
MaskTab1   dc.b    $01,$02,$04,$08,$10,$20,$40,$80
; could be written as
MaskTab1   hex     0102040810204080
; or
           opt     ws+
MaskTab2   hex     01 02 04 08 10 20 40 80
```

4.3.4 DATASIZE

See also DATA on page 57.

DATASIZE is used in conjection with the DATA directive to define constants greater than the 32 bit limit allowed by the DC directive. DATASIZE specifies the size of constants subsequently defined with the DATA directive. For example, using DATASIZE with 4 bytes gives 32 bit constants, with 8 bytes gives 64 bit constants and so on, with the maximum size of a constant being 32 bytes (256 bits). Note that DATASIZE takes only constants as parameters, the maximum number of which is limited only by the line length.

Syntax

```
datasize    Size
```

where:

Size is a value specifying the number of bytes to be used for constants defined using the DATA directive. This value is decimal by default but can be hexadecimal if preceded by '$' or '0x' or binary if preceded by a '%' symbol. The *alternate numeric* form cannot be used.

Example

```
datasize    6                ; 6 byte (48 bit) numbers
...
```

4.3.5 DATA

See also DATASIZE on page 56.

The DATA directive takes a variable number of parameters and defines them as constants. Parameters are decimal by default but can be hexadecimal if preceded by '$' or '0x', binary if preceded by a '%' or in any other radix from 2-9 using the *'r_nnn'* form of integer constants; the *alternate numeric* option cannot be used. Note that DATASIZE takes only constants as parameters, the maximum number of which is limited only by the line length.

Syntax

`data` *Value* [,*Value*]...

where:

Value is the value of the constant. This value is decimal by default but can be in hexadecimal if preceded by a $ symbol. Binary numbers and the alternate numeric form cannot be used. No symbols or operators are allowed.

Example

```
datasize    8                   ; 256 bit numbers
data        10000,1000000
data        $100,-200
data        %11001100
data        8_100
```

4.3.6 IEEE32 and IEEE64

The IEEE32 and IEEE64 directives define 32 bit and 64 bit IEEE floating point numbers respectively.

Syntax

`ieee32` *Constant*
`ieee64` *Constant*

where:

Constant is any valid floating point constant defined by this statement.

Example

```
ieee32  1.234,3.14159,23e11
ieee64  1.23e40,-0.004
```

4.3.7 DS

The DS directive is used to reserve a block of memory and, with the exception of uninitialised data sections, initialise the contents to zero. The label is normally be set to the start of the area defined. However, if the *automatic even* option is enabled (`ae+`) DS.W and DS.L will be set to the beginning of the next word boundary.

Syntax

[Label] `ds[.`*Qualifier*`]` *Count*

where:

Label is an optional label defined by this statement.

Qualifier is an optional qualifier that can be:

`.b`	Byte data
`.w`	Word data
`.l`	Longword data

The DS directive operates as DS.W if no qualifier is specified.

Count is an expression that evaluates to the number of bytes, words or long words to be reserved. *Count* must evaluate.

Example

The following examples all reserve space for 1000 bytes.

```
ScratchBuffer     ds.b    1000
ScratchBuffer     ds.w    500
ScratchBuffer     ds.l    250
```


Note

The DS directive is used to reserve space in BSS sections, the cumulative count being used to set the size of the BSS segment. As with everything else in BSS sections, no initialisation is performed so do not assume that the memory contents have been set to zero.

4.4 Changing The Program Counter

4.4.1 ORG

The ORG directive specifies the starting address from which object code will be generated in the target memory. The address can be an expression which must evaluate and not reference external or undefined symbols; an error will be generated if ORG is used with no address specified. You can use the ORG directive in Sections and if you are producing linkable output provided you only increase the program counter. If you do not use the ORG directive then the assembler assumes an implicit ORG statement with the address set to zero.

If you are assembling to a target with a supporting operating system you can use ORG at the beginning of a section or program to specify the amount of RAM required. This is useful when developing for a machine with the operating system present. The parameter starts with a '?' character followed by the amount of RAM required. The target then returns the address at which it reserved the RAM and the program is assembled to run at that address. This form of the ORG directive has an optional second parameter which indicates the type of memory to be allocated. The value of this parameter is specific to the version of the target software being used.

Syntax

`org` [*Address* | ?*Address*[, *Type*]]

where:

Address An expression specifying the program starting address in the target memory. The expression must evaluate.

Type An optional parameter specifying the type of RAM required by a program.

Example 1

```
        org     $400
Start1  lea     MyStack,sp
        ...
```

Example 2

```
        org     ?512*1024       ;ask for 512K of RAM
        lea     VarBase(pc),a6
        ...
```

Example 3

```
ROMVec          group   org($00000000)
ROMGraphics     group   org($00001180)
ROMCode         group
ROMData         group
ProtRAMData     group   org($FFFF0000),bss
SlowRAMData     group   org($FFFF0180),bss
FastRAMData     group   org($FFFFFB80),word,bss
                org     $00000000
```

4.4.2 EVEN

See also CNOP on page 62.

The EVEN directive forces the program counter to the next even (word-aligned) address. This is useful for ensuring buffers and strings are word aligned. When using sections it is not possible to align the program counter to a larger boundary than the alignment of the current section. for example, you cannot use EVEN in a byte aligned section, the section has to be at least word-aligned.

Syntax

```
even
```

Example 1

The even directive is equivalent to:

```
        cnop    0,2
```

Example 2

```
Prompt      dc.b    'Hit a key when ready',0
            even
; Put buffer on word boundary
Buffer      ds.b    1024
```

4.4.3 CNOP

The CNOP directive sets the program counter to a given offset from a specified boundary. The offset will be from the next address that is a multiple of the boundary.

When using sections it is not possible to align the program counter to a larger boundary than the alignment of the current section. e.g. you cannot use CNOP to align the program counter to 4 bytes in a word (2 byte) aligned section. The assembler will generate a warning if you try to do this.

See also Sections and Groups on page 119.

Syntax

`cnop` *Offset, Boundary*

where:

Offset is an expression that must evaluate and not reference any external or undefined symbols.

Boundary is an expression that must evaluate and not reference any external or undefined symbols.

Example 1

```
cnop        0,2
;same as EVEN directive
cnop        0,4
;next long word boundary
cnop        64,128
;64 bytes above next 128 byte boundary
```

Example 2

```
PopName:    dc.b    'keyhandler',0
            cnop    0,4
```

4.5 Listings

The assembler will generate a program listing during the first pass if you specify a listing file on the command line or set the SNASM68K environment variable to produce one by default.

The assembler normally turns off listing generation whenever it is expanding a macro so if you wish to see your macro expansions you need to set the *List Macros* option (`m+`). Additionally, you can set the *Show Macro Calls* (`mc+`) option to show each macro invocation in the listing file, including nested macros and the nesting level. Also, code that would be ignored due to conditional assembly is placed in the listing only if the *List Failed* (`lf+`) option is set.

4.5.1 LIST and NOLIST

As listings are usually used to check how macros are expanded you will not normally want a listing of your entire program. The LIST and NOLIST directives enable you to control which parts of your program are listed. The simplest form of control involves using the NOLIST directive to turn listing off and turn it back on with the LIST directive. Note that listing generation is turned on if a listing file is specified on the command line; put NOLIST at the start of your program if you do not wish to produce a full listing.

Greater control over listings can be achieved by using LIST with '+' or '-' as a parameter to turn listing on or off respectively. The assembler has an internal listing state starting at 0 which can be incremented or decremented; listings are generated only when the value of this state is non-negative. If you have specified a listing file then the internal listing state will be set to zero i.e. listing will be turned on at the start of the program. You can then use LIST+ and LIST- to increment or decrement the value of the listing state respectively. LIST with no parameter sets the value of the listing state to zero.

Syntax

```
list Operand
...
nolist
```

where:

Operand An optional parameter which can be
- `-` Decrement the internal listing counter.
- `+` Increment the internal listing counter.

Example 1

```
nolist                              ; State=-1, no listing
; This comment will not be listed
list                                ; State=0, listing
; This comment will be listed
list       -                        ; State=-1, no listing
; This comment will not be listed
list       -                        ; State=-2, no listing
; This comment will not be listed
list       +                        ; State=-1, no listing
; This comment will not be listed
list       +                        ; State=0, listing
; This comment will be listed
```

Example 2

```
nolist
opt     ow+,oz+
opt     os+,v+
...
```

4.6 Including Other Files

The assembler provides the ability to include binary files in your source code. As your project grows you will almost certainly want to break your source code into several smaller files in order to make the code more manageable or use some parts in other programs. The assembler allows you to include source files, binary files, or specified parts of a binary file in the main body of your program.

4.6.1 INCLUDE

Normally you will have one 'root' file which includes all the other parts of your code. To do this use the INCLUDE directive which tells the assembler to process another file before continuing with the current one. The line containing the INCLUDE statement is replaced with the contents of the specified source file. These included files can themselves include other files with the total number of include files limited only by the amount of main memory.

Syntax

`include` *Filename*

where:

Filename is the name of the source code file to include.

Example

```
StartUpCode     jmp         MainEntry
                include     'equs.asm'
                include     c:\general\maths.asm
MainEntry       lea         MyStack,sp
                ...
```

If the text following the backslash could be confused with a string equate you should use a second backslash or enclose the full file name in optional quotes. If the file cannot be found it will be searched for in the directories specified by the `j` switch.

4.6.2 INCBIN

The INCBIN directive enables you to include binary data such as graphics or music in your program. The assembler knows nothing of the internal structure of data stored in binary format so you will have to put a label on the data and handle offsets into it yourself.

See also FILESIZE on page 37.

If you need to know the size of a binary file before you include it, you can use the FILESIZE function which returns the size of a file in bytes or -1 if the file cannot be found.

Syntax

`incbin` *Filename* [,*Start* [,*Length*]]

where:

Filename is the name of the binary file to include.

Start is the position in the binary file from which to start including data, specified in bytes from 0. If no start position is specified the default value of 0 is used.

Length is the length of data to include, specified in bytes as an offset from *Start*. If no length is specified the offset will be to the end of the file.

Example 1

```
                    lea       SineTable,a0
                    add.w     d0,d0
                    add.w     d0,a0
; Index words in sine table
                    ...
SineTable           incbin    'c:\tables\sintab.bin'
```

Example 2

```
BackDropPalette     incbin    ..\grafix\backdrop.pal
BackDimPalette      incbin    ..\grafix\backdimp.pal
Main3dPalette       incbin    ..\grafix\textures.pal
```

4.7 Setting Target Parameters

4.7.1 REGS

Use the REGS directive within a group to set the value of the target registers. Typically REGS is used to set the program counter to the address at which program execution is to begin and the value of the status register at this time. As the 68000 has two stack pointers you need to be specific about which you mean by using USP and SSP. If the address is an expression then it must evaluate and not reference undefined or external symbols.

The information provided by REGS is attatched to the section in which it was defined. There can be as many REGS statements as required, with subsequent statements overwriting the previously set values.

REGS can be used when assembling directly to the target or when generating files for subsequent execution. However, REGS cannot be used when producing machine specific relocatable or pure binary formats.

Syntax

`regs` *Register=Value* [,*Register=Value*]

Description

Register The register names valid here can be an extended set of the register names available for programming since processors often include registers which cannot be accessed directly form the instruction set.

Value An expression specifying the value to assign to the register. The expression can contain forward references and can even be left until link time.

Example

```
          org     $400
          regs    pc=CodeStart,sr=$2700,ssp=*
CodeStart lea     MyStack,a7
          ...
```

4.7.2 WORKSPACE

Certain older target hardware does not have its own private memory so it uses about 1K of memory for its own workspace. Use the WORKSPACE directive to change the location of the target workspace.

Syntax

`workspace` *Address*

Description

Address The address in memory where the workspace is to reside

Example

```
workspace   $80000      ;above 512K
org         $400
...
```

4.8 Conditional Assembly

See also Conditional Assembly Macros on page 111.

Conditional assembly structures enable you to modify the behaviour of an assembly under different conditions. The assembler supports six types of conditional assembly structures for evaluating conditions and assembling the correct portion of code as a result. The structures are END, IF...ELSE...ELSEIF...ENDIF, CASE...ENDCASE, REPT...ENDR, WHILE...ENDW and DO...UNTIL.

There is no maximum nesting level of conditional assembly blocks. When repeating blocks of code, the current value of the loop counter is contained in the pre-defined symbol _RCOUNT. Each loop acquires its own local loop counter so the value of _RCOUNT is local to the loop in which it is referenced.

Labels can be placed on conditional assembly directives and used anywhere in expressions. However, if you are in a CASE statement and you put a label on the '=' case selector character, it will be interpreted as a SET directive.

4.8.1 END

The END directive tells the assembler to stop processing text. It is optional as the assembler automatically stops when the end of the source file is reached. END has an optional parameter which can be used to specify the execution address of the program. This is not recommended as the REGS directive can be used to achieve the same effect.

Syntax

```
end [Expression]
```

where:

Expression optionally specifies the execution address of the program.

Example 1

```
        org     $200

        Main body of code

        end
```

Example 2

```
StartUp    lea     Mystack.sp
   ...
   jmp     MainLoop
   end     StartUp      ;Start at StartUp
```

4.8.2 IF...ELSE...ELSEIF and ENDIF

See also CASE... ENDCASE on page 72.

These directives control which parts of the program are assembled according to the result of an expression. They are primarily used to expand different parts of macros under different conditions but can also be used to generate several versions of the program.

The IF directive marks the beginning of the conditional block and has one parameter, an expression. If the expression evaluates to True (i.e. non-zero) then the code following IF and up to the next ELSE, ELSEIF or ENDIF is assembled. If the expression evaluates to False (i.e. zero) then the code following an ELSE, ELSEIF, or ENDIF is assembled. If an ELSE part is present and none of the conditions are met the code between the ELSE and ENDIF (or ENDC) is assembled. For compatibility with other assemblers the ELSEIF directive can be used without any parameters in which case it acts exactly like the ELSE directive. In addition, ENDIF can also be written as ENDC.

Syntax

```
if IfCondition
 ThenPart
elseif ElseifCondition
 ElseifPart
[elseif Condition
 ElseifPart]...
[else
 ElsePart]
[endif|endc]
```

where:

IfCondition	is an expression which must evaluate to a value.
ElseifCondition	is an expression which must evaluate to a value.
Condition	is an expression which must evaluate to a value.

ThenPart is a block of code. This can have nested control constructs as long as they are balanced.

ElsePart is a block of code. This can have nested control constructs as long as they are balanced.

ElseifPart is a block of code. This can have nested control constructs as long as they are balanced.

Example 1

See also CASE... ENDCASE, Example 1 on page 73.

```
False           equ     0
True            equ     -1
LargeBuffer     equ     True
                        ...
                        If      LargeBuffer
                  ds.b  1024
                        else
                  ds.b  256
                        endif
```

The logical not operator (~) can be used to branch on the opposite of a condition but you should be careful to parenthesise the expression correctly. So, in the examples below you may want to branch on the language not being German, . This is done with

```
if      ~(Language=German)
```

but a common mistake is to write

```
if      ~Language=German
```

which logically negates only Language and not the whole expression.

Example 2

See also CASE... ENCASE, Example 2 on page 74.

This example assembles the correct string to order two drinks according to the current language.

```
English equ     0
American        equ     1
French equ      2
German equ      3
Language        equ     English
                ...
; Assemble the correct string to order two
; drinks according to the current language.

        if
        (Language=English)|(Language=American)
          dc.b  'Two beers please',0
        elseif  Language=French
          dc.b  'Deux bieres s''il vous plait',0
        elseif  Language=German
          dc.b  'Zwei Bier bitte',0
        else
          inform        2,"Unknown language"
```

4.8.3 CASE and ENDCASE

See also IF... ELSEIF... ELSE... ENDIF on page 70.

The CASE...ENDCASE structure enables the assembly of a specific block of code given a set of choices. CASE tests the value of an expression against a list of possible values; when it finds a match the specified code is assembled and the assembler moves on to the first instruction after the ENDCASE directive. If the optional '=?' case is used and none of the *SelectorExpression*'s match *Expression* this block of code will be assembled otherwise no code will be assembled inside the CASE... ENDCASE construct (see Example 3 below). The '=?' must be the last choice in the list or the assembler will generate a warning.

Syntax

```
case      Expression
[= SelectorExpression [,SelectorExpression]...]
...
[= SelectorExpression [,SelectorExpression]...]
...
[=?]
...
endcase
```

where:

Expression	is an expression which must evaluate.
SelectorExpression	is an expression which must evaluate.

Example 1

See also IF... ELSEIF... ELSE... ENDIF, Example 1 on page 71.

```
False           equ     0
True            equ     -1
LargeBuffer     equ     True
                        ...
                        case    LargeBuffer

=False
                        ds.b    256

=True
                        ds.b    1024
                        endcase
```

Example 2

See also IF... ELSEIF... ELSE... ENDIF, Example 2 on page 71.

This example assembles the correct string to order two drinks according to the current language.

```
English equ     0
American        equ     1
French  equ     2
German  equ     3
Language        equ     English
                ...
                case    Language
=English,American
                dc.b    'Two beers please',0
=French
                dc.b    'Deux bieres s''il vous
plait',0
=German
                dc.b    'Zwei Bier bitte',0
=?
                inform  2,"Error, unknown language"
                endcase
```

4.8.4 REPT and ENDR

Use REPT and ENDR to repeat a short block of code a specified number of times. The REPT directive signals the start of the code to be repeated and takes an expression that specifies the number of times the code can be repeated. The expression must evaluate and not contain any external or undefined references. The expression is evaluated once only at the point REPT is encountered and no statements in the repeated block can affect the number of times the block is repeated. Also, the block should not contain any unbalanced control statements. The ENDR directive signals the end of the code to be repeated.

See also _RCOUNT on page 27 and ALIAS on page 44

The assembler can access the iteration count from within the body of the loop by using the pre-defined symbol _RCOUNT or any symbol aliased to _RCOUNT.

Syntax

```
rept        LoopCount
 LoopBody
endr
```

where:

LoopCount is an expression which must evaluate to a value.

LoopBody is a block of code. This can have nested control constructs as long as they are balanced.

Example 1

```
rept        16
move.w      d0,-(a0)
endr
```

Example 2

```
TableEntries    equ     24
Index           =       0
                rept    TableEntries
                dc.w    Index
Index           =       Index+64
                endr
```

Example 3

```
TableEntries    equ     24
Index           =       0
                rept    TableEntries
                dc.w    _rcount*64
                endr
```

4.8.5 WHILE and ENDW

See also DO...UNTIL on page 77.

The WHILE directive is used to repeat a short block of code whilst an expression evaluates to true. The WHILE...ENDW construct is similar to DO...UNTIL except that the condition is checked at the *start* of the loop, not the beginning and the loop terminates when the condition becomes *False*, not True.The expression must evaluate to a value and the block is repeated as long as the expression is True. The block will not be assembled if the initial value of the expression is False.

String replacement in the WHILE expression is performed once only, when the directive is encountered. This means that no string changes in the block can affect the number of times the block is repeated. The ENDW directive is used to signal the end of the block of code to be repeated.

Syntax

```
while    LoopCondition
 LoopBody
endw
```

where:

LoopBody	is a block of code. This can have nested control constructs as long as they are balanced.
LoopCondition	is an expression which must evaluate to a value

Example

```
Factor   equ      4

; Build the code required to multiply by factor
Temp     =        Factor
         while    Temp>1
         rol.w    (a0)
Temp     =        Temp>>1
         endw
```

4.8.6 DO and UNTIL

See also WHILE... ENDW on page 76.

The DO...UNTIL construct is similar to WHILE...ENDW except that the condition is checked at the *end* of the loop, not the beginning and the loop terminates when the condition becomes *True*, not False. This means that the block will always be assembled at least once, even if the initial value of the expression is False.The DO directive signals the start of start of the block to be repeated. The block is then repeated until the UNTIL expression evaluates to True.

See also _RCOUNT on page 27 and ALIAS on page 44.

The assembler can access the iteration count from within the body of the loop by using the pre-defined symbol _RCOUNT or any symbol aliased to _RADIX.

Syntax

```
do
 LoopBody
until    LoopCondition
```

where:

LoopBody	is a block of code. This can have nested control constructs as long as they are balanced.
LoopCondition	is an expression which must evaluate to a value.

Example

```
Factor   equ     4
; Build the code required to multiply by factor
Temp     =       Factor
         do
         rol.w   (a0)
Temp     =       Temp>>1
         until   Temp<=1
```

4.9 Manipulating Strings

The assembler provides a range of pre-defined functions and directives for string handling. These are usually used in macros for comparing, searching and slicing strings with character positions starting at 1.

4.9.1 STRLEN

The STRLEN function is used to determine the number of characters in a string. It can be used anywhere in an expression.

EXAMPLE

```
;Macro to DC string preceded by its length
String   macro
         dc.b     strlen(\1),\1
         endm
         ...
         String   'Hello'
```

4.9.2 STRCMP and STRICMP

Use the STRCMP and STRICMP functions to compare two strings. The comparison is case sensitive for STRCMP and case insensitive for STRICMP. If the two strings are identical the function returns True (1), otherwise False (0).

EXAMPLE

```
Language   equs    'English'
   ...
; Assemble correct order according to current language.
   if      strcmp('\Language','English')
     dc.b 'Two beers please',0
   else
     if    strcmp('\Language','French')
       dc.b       'Deux bieres s''il vous plait',0
     else
       if strcmp('\Language','German')
         dc.b     'Zwei bier bitte',0
       endif
     endif
   endif
```

4.9.3 INSTR and INSTRI

The INSTR and INSTRI functions are used to see if one string is contained within another. INSTR performs a case sensitive search and INSTRI performs a case insensitive search. If a string does contain a sub-string the function returns the position of the first character of the sub-string, otherwise INSTR returns zero. These functions have an optional parameter which specifies the start position (in the string) of the search. This does not affect reporting of the sub-string start position.

EXAMPLE

```
Version     equs    'Internal test version 0.9'
            ...
;Set DebugMode if version string contains the word 'test'
            if      instr('\Version','test')
DebugMode   =       -1
            else
DebugMode   =       0
            endif
```

4.9.4 SUBSTR

The SUBSTR directive is similar to EQUS in that it is used to equate a string to a symbol. However, SUBSTR also allows you to specify the start and end characters of the string.

Syntax

Symbol `substr` [*Start*],[*End*],*String*

where:

Symbol	is the symbol to be assigned to the sub-string.
Start	is the starting position of the sub-string in *String* to be assigned to *Symbol*.
End	is the end position of the sub-string in *String* to be assigned to *Symbol*.
String	is the string containing the sub-string.

Example

```
TestStr    equs     'What does this do?'

Temp1      substr   1,18,'\TestStr
; This is the same as
; Temp1    equs     'What does this do?'

Temp2      substr   6,9,'\TestStr'
; Temp2 will equal 'does' (without the quotes)

Temp3      substr   ,4,'\TestStr'
; Temp3 will equal 'What'

Temp4      substr   6,,'\TestStr'
; Temp4 will equal 'does this do?'
```

4.10 Modules

Modules are self-contained sections of code, delimited using the MODULE and MODEND directives. Modules are used to control the scope of local labels and are strongly recommended as they rigidly define where local labels can and cannot be referenced.

4.10.5 Local Labels in Modules

A local label is assumed to be inside a module if it is defined on any line after the MODULE directive up to and including the line on which the MODEND directive occurs. Local labels defined on same line as a MODULE directive are considered to be in the outer scope. A local label defined in a module cannot be referenced outside that module and so can be reused elsewhere. Similarly, local labels defined outside a module cannot be referenced from within it. Modules take precedence over the 'between non-local labels' form of scoping but you are still free to use this use form outside of modules. Modules can be nested but scoping is not. That is, the only local labels available inside a module are those defined within it.

Syntax

```
module
...
modend
```

Example 1

```
ClearData     module

@Loop         move.w     d0,d2
              bsr        @ClearIt
              dbra       d1,@Loop

              rts

@ClearIt      module

@Loop         clr.b      (a0)+
              dbra       d2,@Loop
              rts

              modend
; end of @ClearIt

; A reference to @Loop would refer to the first
; definition as we are back in the module ClearData.

              modend
; end of ClearData
```

Example 2

This example works only if it is enclosed in a module otherwise the assembler generates an error on the last line because the DBRA is not within the scope of @LOOP.

```
; This will not work unless it is all enclosed in a
; module.
@Loop         movem.w    d7,-(sp)
              ...
@SubModule    module
              ...
              modend
              movem.w    (sp)+,d7
              dbra         d7,@Loop
```

4.11 Options and Optimisations

The assembler has several options and optimisations which control the assembly process. They can be set from the command line when invoking the assembler or from within your source code using the OPT, PUSHO and POPO directives. The options and optimisations are briefly reviewed here.

4.11.1 Options

Options enable you to control the behaviour of the assembler and how it outputs information to the screen, listing and object file. They can be set from the command line or from within your source using the OPT directive. However, it is recommended that production code always sets options from within the source file as they can significantly alter the code generated and can cause assembly errors if not correctly set.

The options are described in the table below. Do not use white space between the option name and the '+' or '-' and separate multiple options with commas (white space is allowed after the comma but not before it).

Option	Default	Description
`ae+/-`	On	*Automatic Even.* This forces the program counter to the next word boundary before assembling the word and long forms of DC, DCB, DS and RS.
`an-/+`	Off	*Alternate Numeric.* Allows the use of character suffixes H, D, Q and B to denote Hexadecimal, Decimal, Octal and Binary constants respectively.
`bin-/+`	Off	*Show Binary.* Show full binary in the listing file.
`c-/+`	Off	*Case Sensitivity.* By default all symbols are case insensitive, for example `Main` and `main` are treated as the same label. Enable this option to make labels case sensitive so that `Main` and `main` would be two distinct labels.

Continued on next page.

Continued from previous page.

Option	Default	Description
`d-/+`	Off	*Descope Local Labels.* When used outside of modules the EQU and SET directives do not affect the scope of local labels. Set this option if you want these directives to descope local labels.
`l-/+`	Off	*Local Label Character.* Toggle between '.' (`l+`) and '@' (`l-`) as the local label character.
`l`*Value*		Define the local label character where *Value* is the ASCII code for the character. Valid local label characters are '@', '.', ':', '?', '|' and '!' *only.* The default local label character is '@'.
`lf-/+`	Off	*List Failed.* This lists instructions not assembled due to conditional assembly statements.
`m-/+`	Off	*List Macros.* Lists macro expansions in listing file.
`mc-/+`	Off	*Show Macro Calls.* In the listing file, shows each macro invocation including nested macros and the nesting level.
`s-/+`	Off	*Equated Symbols as Labels.* Treat equated symbols as labels.
`t-/+`	Off	*Truncate.* Disables expression overflow checking.
`v-/+`	Off	Write local labels to MAP file.
`w+/-`	On	*Print warnings.* Warnings are not errors but unusual occurrences that can be reported.

Continued on next page.

Continued from previous page.

Option	Default	Description
ws-/+	Off	*Allow white space.* This allows space and tab characters in operands to increase code readability. Normally a blank terminates the operand field and begins the comment field. With the WS option enabled the comment field must begin with a semi colon (;) and white space in the operand field is ignored.
x-/+	Off	*External symbols.* Assume external symbols are in the section they are declared in.

Table 11. Assembler command-line options.

4.11.2 Optimisations

Optimisations modify source statements so that they use more efficient addressing modes and instructions where possible. However, they cannot be performed on expressions containing forward references.

The optimisations are described in the table below. They can be set from the command line or from within your source using the OPT directive. Do not use white space between the optimisation name and the '+' or '-' and separate multiple options with commas (white space is allowed after the comma but not before it).

Optimisation	Description
op+/-	*PC Relative.* Changes absolute long addressing to PC relative addressing if possible and legal.
os+/-	*Short Branch.* Forces forward references in relative branches to use the short form of the instruction.
ow+/-	*Absolute Word.* Forces absolute word addresses to short word addressing if in range.

Continued on next page.

Continued from previous page.

Optimisation	Description
`oz+/-`	*Zero Displacement.* Changes address register indirect with displacement to address register indirect if the displacement evaluates to zero.
`oaq+/-`	*Quick ADD.* Changes the ADD instruction to the shorter ADDQ.
`osq+/-`	*Quick SUB.* Changes the SUB instruction to the shorter SUBQ.
`omq+/-`	*Quick Move.* Changes the MOVE.L instruction to the shorter MOVEQ. MOVE.W is not changed as MOVEQ is defined as long.

Table 12. Assembler command-line optimisations.

4.11.3 OPT

Use OPT to set the assembler options and optimisations for the duration of the assembly.

Syntax

`opt` `[ae{+|-} | an{-|+} | bin{-|+} | c{-|+} | d{-|+} | l{-|+|`*Value*`} | lf{-|+|`*Value*`} | m{-|+} | mc-|+} | op{+|-} | os{+|-} | ow{+|-} | oz{+|-} | oaq{+|-} | osq{+|-} | omq{+|-} s{-|+} | t{-|+} | v{-|+} | w{+|-} | ws{-|+} | x{-|+} |]`

where:

`ae` *Automatic even* option.
`an` *Alternate numeric* option.
`bin` *Show binary.*
`c` *Case sensitivity* option.
`d` *Descope local labels* option.
`l` *Local label character* option.
`lf` *List failed* option.
`m` *List macros* option.
`mc` *Show macro calls.*
`op` *PC relative* optimisation.

os	*Short branch* optimisation.
ow	*Absolute word* optimisation.
oz	*Zero displacement optimisation.*
oaq	*Quick ADD* optimisation.
osq	*Quick SUB* optimisation.
omq	*Quick MOVE* optimisation.
s	*Equated symbols as labels* option.
t	*Truncate* option. This disables expression overflow checking.
v	Write local labels to COFF file option.
w	*Print warnings* option
ws	*Allow white space* option.
x	*External symbols.*

Example

This example sets the *automatic even, equated symbols as labels, case sensitivity* and *write local labels to COFF File* options and enables the *short branch, absolute word* and *zero displacement* options.

```
opt    ae+,s+,c+,v+,os+,ow+,oz+
```

4.11.4 PUSHO and POPO

The PUSHO and POPO directives are used to temporarily change options and optimisations during assembly. This allows you invoke the assembler with your normal settings, change one or more settings at some point in your source code and then change back again. The PUSHO directive saves the current state of all the options and optimisations and POPO restores the state previously saved using PUSHO.

Syntax

```
pusho
...
popo
```

Example

```
            pusho           ; save state of options
            opt     ae-     ; turn off auto even
ByteStream  dc.b    3
            dc.w    456
            dc.w    512,80
            popo            ; restore state
```

4.12 User Generated Errors and Warnings

The assembler allows you to generate your own errors and warnings. This is useful for generating messages for error conditions that the assembler cannot detect such as a data table becoming too large.

4.12.1 INFORM

Use the INFORM directive to generate errors of varying severity and display a message explaining the error in detail. The directive takes two parameters, the severity and a message string plus any optional operands.

Syntax

`inform` *Severity,String*[*,Operand*]...

where:

Severity An integer in the range 0..3 inclusive that determines the action to be taken, where:

0	Prints a message but no action is taken.
1	Generates a Warning.
2	Generates an Error.
3	Generates a Fatal Error.

String The message you wish to display. A more informed message can be displayed using the %d, %h and %s parameters where:

%d	Substitutes the decimal value of the operand.
%h	Substitutes the hexadecimal value of the operand.
%s	Substitutes the string value of the operand.

Operand An optional parameter which can be

Example 1

```
StartSlowRAM  equ       *
              ...
SlowRAMSize   equ       *-StartSlowRAM
              ...
              inform    0,"Slow RAM size: $%h",SlowRAMSize
```

Example 2

```
StrucBeg      dc.w    0
              ...
StrucEnd
StrucLen      equ     StrucEnd-StrucBeg
              if      StrucLen>1024
                inform  0,'Beg=%h End=%h',StrucBeg,StrucLen
                inform  2,'Structure too long'
              endif
```

4.12.2 FAIL

The FAIL directive is supported for compatibility and is the equivalent of:

```
inform 3,'Assembly failed'
```

4.13 Linking

Linking enables you to write programs in separate parts and subsequently combine them to send direct to memory or to produce a single object file. This enables sections and groups to be built from sub-files and for address references between object files to be resolved. Link facilities are fully integrated into the assembler providing a combined 'linking assembler'. This offers the benefit of a unified command file for both assemble and link instructions so that the full range of assembly commands are available when linking. In addition, code that uses libraries can be linked in one go rather than requiring two phases.

Linking often creates the need to reference symbols defined in a different program component to the current one. To reference a symbol in another component you need to declare the symbol as external in the component that the symbol was defined, using the EXPORT directive. Then, in the component that you wish to use the symbol you need to declare that the symbol has been defined in another component, using the IMPORT directive. The PUBLIC directive is used to declare a large group of symbols as external without the need to use EXPORT for each symbol. Alternatively, use the GLOBAL directive in place of both EXPORT and IMPORT and let the assembler determine whether an IMPORT of EXPORT will ultimately be required.


Note

As linking is closely related to the concept of sections and groups this section on linking should also be read in conjunction with the chapter on Sections and Groups starting on page 119.

4.13.1 EXPORT

The EXPORT directive enables you to make symbols defined in the current file visible to the linker so that they are available to other program files. All references to the EXPORTed symbol will be resolved by the linker.

Syntax

`export` *Label* [,*Label*]...

where:

Label is any symbol defined by this statement.

Example

```
          import.w   Table
          export     Routine1,Routine2
Routine1  lea        Table,a0
          ...
Routine2  mulu       d0,d0
```

4.13.2 IMPORT

The IMPORT directive enables you to refer to symbols defined in other program components, leaving label resolution to the assembler. By default the assembler does not know where to find an imported symbol so it makes no assumptions as to it location. Enabling the *external symbols* option (x+) makes the assembler assume that the imported symbol comes from the currently active section. IMPORT should not be defined in the current module or assembler will generate a warning.

Syntax

`import`[*.Qualifier*] *Label* [*,Label*]...

where:

Label — Any symbol previously defined in another source code module.

Qualifier — An optional qualifier that can be:

`.b`	Byte data
`.w`	Word data
`.l`	Longword data

Example

```
            export    Table
            import    Routine1,Routine2
            section   Tables,BssGroup
Table       ds.w      100
            section   Code,Text
            jsr       Routine1
            jsr       Routine2
```

4.13.3 PUBLIC

The PUBLIC directive enables you to declare a large group of symbols as external without having to explicitly use the EXPORT directive for each symbol. PUBLIC can take two arguments, ON and OFF. To declare symbols as external set PUBLIC to ON, define your symbols as normal and then set PUBLIC to OFF as in the example below.

Syntax

`public` {`on` | `off` | *Flag*}

where:

`on`	Set PUBLIC active.
`off`	Set PUBLIC inactive.
Flag	A string equate whose value has been set to the string "ON" or "OFF".

Example

```
            public    on
Speed       dc.w      50      ; No need to EXPORT Speed
Direction   dc.w      100     ; or Direction
            public    off
```

4.13.4 GLOBAL

The GLOBAL directive enables you to substitute for EXPORT and IMPORT in cases where you are not sure if you need to declare a symbol as external or if the symbol you want to reference has been declared as external in another program component. If the symbol is eventually defined an EXPORT will be performed, otherwise an IMPORT will be assumed.

Syntax

`global` *Symbol* [,*Symbol*]...

where:

Symbol A symbol defined by this statement.

4.13.5 A Guide to Linking

This section provides a guide to the linking process by showing how to convert a single source file into two linkable files. This covers all the stages involved from the changes required to source files to command-line syntax. To start, consider the source file below. This has two groups, G1 and G2. The group G1 has two sections, S1 and S3 where S1 contains the routine FUNC1 and S3 contains the FUNC3 routine. The group G2 also has two sections, S2 and S4 where S2 contains the routine FUNC2 and S4 contains the FUNC4 routine.

```
            group   g1,org $100
            group   g2,org $1000

            section s1,g1
start:

            jsr     func1
            jsr     func2
            jsr     func3
            jsr     func4
func1
            move.l  #1,d0
            addq    #4,d0
            rts

            section s3,g2
func2
            move.l  #2,d0
            addq    #4,d0
            rts

            section s3,g1

func3
            move.l  #3,d0
            addq    #4,d0
            bra     func1
            rts

            section s4,g2

func4
            move.l  #4,d0
            addq    #4,d0
            bra     func2
            rts

            ds.b    1000
stack:
```

Now suppose that the above code could be split into two separate but inter-related parts such that one part contains the code for FUNC1 and FUNC2 but also needs to make use of FUNC3 and FUNC4 and similarly, the other part contains the code for FUNC3 and FUNC4 but needs to make use of FUNC1 and FUNC2. The two parts could be written to separate files, TEST1.68K and TEST2.68K for example, which could subsequently be linked together to

achieve the same effect as the code in the original single file. TEST1.68K would be as follows:

```
            export   start,func1,func2
            import   func3,func4

            section  s1
start:

            jsr      func1
            jsr      func2
            jsr      func3
            jsr      func4
func1
            move.l   #1,d0
            addq     #4,d0
            rts

            section  s3
func2
            move.l   #2,d0
            addq     #4,d0
            rts
```

The START routine needs to make use of FUNC3 and FUNC4 which are in TEST2.68K. The IMPORT directive enables START (and any other section of code) to reference FUNC3 and FUNC4. Not IMPORTing FUNC3 and FUNC4 would cause the assembler to generate a 'symbol not defined error' when assembling TEST1.68K. Similarly, TEST2.68K requires the use of START, FUNC1 and FUNC2 so they must be made available to TEST2.68K (and any other source files) by means of the EXPORT directive. Not EXPORTing START, FUNC1 and FUNC2 would cause the assembler to generate a 'symbol not defined error' when assembling TEST2.68K.

Note that the GLOBAL directive could have been used in place of IMPORT and EXPORT as shown below.

```
            global   start,func1,func2
            global   func3,func4
```

Hence, on finding that FUNC3 and FUNC4 had not been defined TEST1.68K, the first GLOBAL directive would behave in the same way as the IMPORT directive. Similarly, as START, FUNC1 and FUNC2 are defined in TEST1.68K the second GLOBAL directive would behave as it were an EXPORT directive.

Note also that the groups to which sections S1 and S2 are to be allocated have been are no longer defined in the source file. When linking, the definition of groups and the allocation of sections to groups can be left until later and is typically done from within the root file used to include the various project components. A sample root file is described later on.

The TEST2.68K file is shown below. The IMPORT directive enables FUNC3 and FUNC4 (plus any other section of code) to reference FUNC1 and FUNC2 respectively. Not IMPORTing FUNC1 and FUNC2 would cause the assembler to generate a 'symbol not defined error' when assembling TEST2.68K. Similarly, TEST1.68K requires the use of FUNC3 and FUNC4 so they must be made available to TEST1.68K (and any other source files) by means of the EXPORT directive. Not EXPORTing FUNC3 and FUNC4 would cause the assembler to generate a 'symbol not defined error' when assembling TEST1.68K.

```
            export   func3,func4
            import   func1,func2

            section  s3

func3
            move.l   #3,d0
            addq     #4,d0
            bra      func1
            rts

            section  s4

func4
            move.l   #4,d0
            addq     #4,d0
            bra      func2
            rts

            ds.b     $1000
stack:
```

The two source files TEST1.68K and TEST2.68K now require assembling prior to linking. Although the linker is incorporated into the assembler, assembly and linking of the same source file cannot be performed at the same. This is because it would create a situation where the assembler would attempt to both define and import symbols at the same time so that references to symbols could never be resolved. As such, assembly and linking remain two distinct processes.

To produce linkable object files the assembler must be invoked with the linkable output switch (/l). The following example produces an output file TEST1.COF that can either be linked with other COFF object files.

```
snasm68k /l test1.68k,test1
```

Before progressing, it is worth remembering that multiple files can be assembled or linked at the same time. The following example assembles both TEST1.68K and TEST2.68K to produce a single linkable output file TEST.COF.

```
snasm68k /l test1.68k+test2.68k,test
```

SNMAKE is described on page 167.

However, assemblies involving multiple source files can be achieved more efficiently by using the make utility SNMAKE.

Multiple object files can be linked as follows:

```
snasm68k /l test1.cof+test2.cof,t7:
```

This will link TEST1.COF and TEST2.COF together and download them to target 7. Note that files linked in this way must have the .COF file extension in order for the assembler to determine that they are object files. Otherwise, the assembler and will attempt to assemble the files as if they were source files.

To return to the discussion, assume that the source files TEST1.68K and TEST2.68K have been assembled to produce corresponding linkable output files TEST1.COF and TEST2.COF. The issue of sections and groups can now be addressed. As stated earlier, this is typically done from the root file. The root file TEST.68K, shown below, includes the two object files TEST1.COF and TEST2.COF and orders groups and sections within groups.

```
group    g1,org $8000
group    g2,org $9000

section  s1,g1
section  s3,g1

section  s2,g2
section  s4,g2

regs     pc=start

include  test1.cof
include  test2.cof
```

The root file can now be assembled as follows:

```
snasm68k  test.68k,t7:test
```

This example assembles TEST.68K, downloads the object code to target 7 and generates an executable COFF file TEST.COF. The executable could also be downloaded to a target using the debugger or the SNGRAB utility.

4.13.6 The Command File

For more information on command files see page 22.

Linking can be achieved either from the command-line or by using a command file. Both assembly and link information is specified in a single command file. This contains instructions on which object files to use, their starting addresses and information about groups. The following example command file entry produces a linkable object file TEST.COF and downloads it to target 7.

```
snasm68k test.68k,t7
```

If an object file extension other than .COF is used then object files should be included in a command file as follows:

```
include test1.o,cof
include test2.o,cof
```

Library files can be included as follows:

```
include lib1,lib
```

or

```
include lib1.lib
```

Groups are normally placed in memory in the order in which they are declared in a program. However, groups declared in a program but not in the command file are placed at the end of the declared groups. Sections within each group are placed in memory in the order in which they are specified. Section fragments from different source files are concatenated in the order in which the source files are specified.

This is the only information this page contains.

5 Macros

Macros let you to assign a symbolic name to a sequence of processor instructions and assembler directives. The sequence can then be assembled whenever required by invoking the symbolic name of the macro. Macros can be used as many times as you wish and parameters passed to it, simplifying programming and improving code readability.

The assembler lets you:

- Define your own macros.
- Manipulate strings.
- Define conditional and repeatable blocks within a macro.
- Control macro expansion listing.
- Manage macro memory use.

The topics covered in this section are:

- Introducing Macros
- Macro Parameters
- Short Macros
- Extended Parameters
- Advanced Macro features

5.1 Introducing Macros

There are three stages to macro use; *definition*, *invocation* and *expansion*, described below.

5.1.1 Defining Macros

The MACRO directive is used to introduce the definition of a macro and ENDM terminates the definition. All subsequent statements up to the corresponding ENDM directive are then copied into memory. A macro can subsequently be redefined at any point in the program, the old copy being first removed from memory. Explicitly removing macros from memory is achieved using the PURGE directive discussed later.

The macro name is defined in the label field of MACRO. A macro can have the same name as a label as macro names are stored in separate symbol table to normal symbols. This has been done so that macro names do not clash with similarly named routines in your main code. However, if you attempt to define a macro with the name of a current directive or processor instruction, the assembler will generate a warning stsring that the macro cannot be called. You can however, ALIAS a directive or instruction, DISABLE the old name and define a macro using the old directive or instruction name.

The assembler allows you to define *nested* macros - a macro defined or redefined within a macro. How deeply you can nest macros is limited only by the amount of available memory. The assembler allows you to redefine or purge a macro within itself. The redefinition or purge will take place only when the macro exits.

Syntax

MacroName `macro` [*ParameterList*]
MacroBody
[`mexit`]
`endm`

where:

MacroName is a symbol defined by this statement.

ParameterList is a comma delimited list of parameters as described in sect.

MacroBody is a block of code which can include nested macros and balanced control constructs. If a structure is started in a macro it must be finished before the ENDM directive. Similarly, you cannot terminate a structure that was not initiated within that macro. In both cases the assembler will generate an error.

5.1.2 Invoking a Macro

A macro is invoked by using its name as if it were an assembler directive. This is known as a *macro call.* A macro can be called with an optional modifier which can be any text. The modifier is usually used to convey size information but can be any information required by the macro such as whether a jump is long or short. For more information on invoking macros see the section on macro parameters.

5.1.3 Expanding a Macro

When the source program calls a macro, the assembler substitutes the statements within the macro definition for the macro call statement. The MEXIT directive can be used within a macro to immediately terminate the macro expansion. The assembler then continues from the line after the ENDM directive. MEXIT is supported for compatibility only as the conditional assembly macros render this directive redundant. If MEXIT *is* used, care should be taken when using MEXIT as it creates multiple exit points from the macro. Note that whilst both MEXIT and ENDM terminate a macro *expansion* only ENDM terminates a macro *definition.*

Example 1

```
BIOSCall    macro
            move.w      #\1,d0
            if  narg=2
            lea.l       \2,a0
            endif
            jsr         _CDBIOS
            endm
```

Example 2

This macro checks that longs are on a long boundary. The macro exit condition uses MEXIT as an illustration only as this would normally have been implemented using the IF... ELSE... ENDIF construct.

```
longs    macro
         if       (*&3)<>0
           inform 2,"Longs not on long boundary"
           mexit
         endif
         dc.l     1,2,3
   endm
```

5.2 Macro Parameters

Once a macro has been defined it can be called with up to 32 parameters. They can be used anywhere in the macro in the same way as string equates as the assembler treats macro parameters and string equates in a similar way. There are two ways to access a parameter, as a *numbered parameter* which is the parameter number preceded by a backslash, and as a *named parameter* which uses the symbol given to the parameter when the macro is defined. There are also three *special parameters* that perform functions associated with the macro invocation.

5.2.4 Numbered Parameters

Numbered parameters are denoted with a backslash ('\') followed by a number from 0 to 31. If this could cause confusion as to where the macro parameter ends, use a second backslash after the parameter. If a macro parameter needs to include spaces or commas then the parameter should be enclosed in angle brackets ('<' and '>'). These are not considered to be part of the parameter, so if you need to include a '<' or '>' you need to double up the required character to read '>>' or '>>'.

Example 1

```
lotus           macro       joe
                while       joe
; fred evaluated at each iteration
                shift
                dc.w        joe
                endw
                endm
                ...
                lotus    1,2,3
```

Example 2

```
Slasher  macro    1,2
; Single backslash used here
         dc.b     'X is \1 and Y is \2'
; Both backslashes required
         dc.b     '123\1\456'
         endm
```

Example 3

```
infinite        macro       jim
                while       \jim
                  shift
                  dc.w      \jim
                endw
                endm
                ...
                infinite    1,2,3
```

5.2.5 Named Parameters

The assembler allows you to use symbolic names for the parameters \1 to \31 and these named parameters do not require a preceding backslash when used in expressions.

Example

```
Scale     macro     X,Y,Factor
          dc.w      \X*Factor,\Y*Factor
```

5.2.6 Variable Numbers of Parameters

Macros can take variable numbers of parameters. NARG and SHIFT are used to determine how many parameters a macro has and then to step through them. When a macro is invoked, the number of parameters is given by the pre-defined symbol NARG. The SHIFT directive is used to remove the first parameter and renumber and (less commonly) relabel the remaining parameters.

Syntax

```
shift
```

Example

```
; Double the given parameters and then DC them
DCx2    macro
        rept        narg
          dc.\0     \1*2
          shift
        endr
        endm
        ...
        DCx2.w      2,8,9
; The words 4,16 & 18 are dc'd
```

5.2.7 Labels as Parameters

A macro can import the label on the macro invocation line and use it in the same way as any other parameter. To do this the first named parameter of the macro definition must be an asterisk ('*'). Then '*' is used to substitute for the label. The label must be explicitly defined by the macro, it is not defined to be at the current program counter as is usually the case. If the label is not defined * will be a null string, possibly causing errors.

Example

This macro assigns labels relative to the start of a data table.

```
rc          macro     *,Data
            if strlen('\*')=0   ;Check for null \*
              inform 2,'Label undefined'
            else
\*          equ       *-VarBase ;Make L1 & L2 relative to VarBase
            rept      narg(Data)
              dc.\0   \Data
              shift   Data
            endr
            endm
            ...
VarBase     equ       *
L1          rc.w      {1,2,3,4}  ;L1 not treated as a label
L2          rc.w      {5,6,7,8}  ;L2 not treated as a label
            ...
            lea       VarBase(pc),a6
            move.w    L1(a6),d0
```

5.2.1 Special Parameters

There are three special parameters available for use in macros. The '\0' parameter denotes the size modifier of a macro when it was invoked, the '_' parameter returns a string containing the entire macro parameter string and the '\@' parameter which is used to generate unique labels each time a macro is called.

The '\0' parameter denotes the size modifier of the macro when it was invoked. The size is specified by a period and a size modifier immediately following the macro name. If the macro is invoked without a size modifier, '\0' will be replaced with a null string, the default value.

The '_' parameter returns a string containing the entire parameter string from the remainder of the macro invocation line up to but not including the end of line or a comment. This feature enables a macro to interpret its invocation line which is useful when you are invoking a macro from within a macro.

The '\@' parameter is used to generate unique labels each time a macro is called. The label is terminated by the '\@' which expands to an underscore (_) followed by a decimal number of the form *nnn*. The number increments each time any macro is called, thus guaranteeing a unique label for each macro invocation. Note that within a particular macro call all references to '\@' will return the same string even if other macros are called from within that macro.

Example 1

```
        inform    0,"Assembling \_filename""
```

Example 2

```
Inc     macro             ; increment register
        addq.\0   #1,\1
        endm
        ...
        Inc       d0      ; Expands to addq. #1,d0
        Inc.b     d1      ; Expands to addq.b #1,d1
        Inc.w     d0      ; Expands to addq.w #1,d0
        Inc.l     d7      ; Expands to addq.l #1,d7
```

Example 3

```
BraNz    macro              ; branch if register not zero
         tst.w      \1
         bne.\0     \2
         endm
         ...
         BraNz      d0,Exit
         BraNz.s    d7,Again
```

Example 4

Assuming that no other macros have yet been called, the first time the Delay macro is called `Loop\@` expands to `Loop_000`. Calling three other macros and calling Delay again results in `Loop\@` expanding to `Loop_004`.

```
Delay    macro
         lda        \1
Loop\@   dec        a
         bne        Loop\@
         endm
         ...
         Delay      #3,d0
```

5.3 Short Macros

Short macros are defined using the MACROS directive. They contain only a single line of code and do not have a ENDM directive. Short macros are useful for porting code from other assemblers where you may need to define a macro to imitate a control structure.

Syntax

`macros` *MacroLine*

where:

MacroLine is a line of code.


Note

Short macros, by definition, can contain only part of a conditional assembly structure. So, unlike other macros, short macros are expanded inside failed conditions and other assembly flow constructs in case they define a closing condition statement.

Example 1

A macro to implement the IFEQ (if equal) conditional assembly construct.

```
ifeq     macros
           if      \1=0
; Note that short macros don't have an ENDM
           ...
           ifeq    DebugMode
           ...
           endif
```

Example 2

A macro to implement the IFND (if not defined) conditional assembly construct.

```
ifnd     macros
           if      ~def(\1)
         ...
         ifnd      Count
Count      dc.w    0
         endif
```

5.3.2 Conditional Assembly Macros

Command line switches are described on page 19.

The assembler provides several pre-defined short macros that enable you to implement additional conditional assembly structures and these are listed below. These macros are automatically defined if you invoke the assembler with the k command line switch. The short macro definitions are listed below.

Conditional	Macro
If Defined	`IFD macros` ` if def(\1)`
If Not Defined	`IFND macros` ` if ~def(\1)`
If Zero	`IFEQ macros` ` if (\1)=0`
If Not Zero	`IFNE macros` ` if (\1)<>0`
If Greater Than	`IFGT macros` ` if (\1)>0`
If Less Than	`IFLT macros` ` if (\1)<0`
If Greater Than or Equals	`IFGE macros` ` if (\1)>=0`
If Less Than or Equals	`IFLE macros` ` if (\1)<=0`
If Strings are Equal	`IFC macros` ` if strcmp(\1,\2)`
If Strings are Not Equal	`IFNC macros` ` if ~strcmp(\1,\2)`

Table 13. Conditional assembly macros.

5.4 Advanced Macro Features

5.4.1 Extended Parameters

The assembler allows you to pass a list of items enclosed in curly brackets ('{' and '}') to a macro parameter. The parameter is treated like an EQUS symbol to which a list has been previously assigned. The NARG symbol can be used to report how many items have been assigned to the parameter and SHIFT can be used to manipulate the list in a similar way to macro parameters. The example below may look complex but note how all the complexity is hidden away inside the macro and how neat the main code looks.

Example

```
Black       equ         0
Green       equ         1
Red         equ         2
            ...
; Macro which takes colours and point lists e.g.
; Black,{0,2,3},Green,{0,3,6,8},Red,{2,4}
; and generates data containing the colour, the count of
; points and then the point data e.g.
; dc.b Black
; dc.b 3
; dc.b 0,2,3
; dc.b Green
; dc.b 4
; dc.b 0,3,6,8
; dc.b Red
; dc.b 2
; dc.b 2,4

PolyList    macro

polys\@     =           narg/2
; Check narg was even
            if          polys\@*2<>narg
            inform      2,'Bad parameter list'
            else
; Handle all polygons
            rept        polys\@
            dc.b        \1
points\@    =           narg(2)
            dc.b        points\@
            rept        points\@
            dc.b        \2
            shift       2
            endr

            shift
            shift

            endr
            endif
            endm
            ...
            PolyList
            Black,{0,2,3},Green,{0,3,6,8},Red,{2,4}
```

Continued on next page.

For compatibility with other assemblers it is possible to pass a list of parameters enclosed in angle brackets ('<' and '>') instead of curly brackets. This means having to write your code slightly differently to that used to take parameters enclosed in curly brackets ('{' and '}'). The example below shows the differences.

Example

```
; DefItems macro with parameters enclosed in {}
DefItems   macro
           rept    narg(1)
           dc.b    \1,0
           shift   1
           endr
           endm
           ...
           DefItems {'Mon','Tue','Wed','Thu','Fri'}


; DefItems macro with parameters enclosed in <>
DefItems   macro
Day        equs    {\1}
           rept    narg(Day)
           dc.b    \Day,0
           shift   Day
           endr
           endm
           ...
           DefItems <'Mon','Tue','Wed','Thu','Fri'>
```

5.4.2 Local Labels in Macros

You can use local labels and modules in macros. The scope of local labels defined in modules is not affected by nested macros. This means that if you invoke a macro in a module you can reference any of the local labels defined in that module when the macro is expanded. If you want to have a macro with its own local labels you can use a module in the macro, use the '\@' parameter or use the LOCAL directive.

LOCAL

Use the LOCAL directive to declare local labels in macros. Labels declared in this way are defined to be local to the outermost macro nesting level, which for non-nested macros is the macro in which the label was defined. The LOCAL directive does not type the symbols it defines; they can be used as labels, text equates or as you wish.

Syntax

```
local       [Label]
```

where:

Label is any symbol defined by this statement.

Example 1

```
Delay       macro
            local       Loop
            move.w      \1,\2
Loop        dbra        \2,Loop
            endm
```

Example 2

```
Demo        macro
            local    Skip,String1,Gravity
            bra      Skip
String1     equs     '\1\\2'     ; String equate
Gravity     equ      10          ; Numeric equate
Skip                   ; Label
            endm
```

5.4.3 PUSHP and POPP

The assembler maintains a global stack which can be manipulated with the PUSHP and POPP directives. PUSHP allows you to push some text onto the stack and POPP will pop the top element of the stack into any string variable. As the stack is global you are not restricted to popping the parameter in the same macro that pushed it. This allows a great deal of flexibility when writing macros to handle self-referencing data structures.

Syntax

```
pushp
...
popp
```

Example

```
; DC parameters in reverse order
BackDC      macro
            local   temp
; Push them all
            rept    narg
            pushp   '\1' ; push text contents of \1
            shift
            endr
; now pop and DC them
            rept    narg
            popp    temp ; pop text pushed earlier
            dc.\0   \temp
            endr
            endm
            ...
            BackDC.w        1,5,7,8
```

5.4.4 PURGE

The assembler stores macros very efficiently but they can still take up a lot of memory. If the macro is no longer needed it can be removed using the PURGE directive. This removes the macro from the symbol table and frees up the memory it was using.

A macro is allowed to purge itself, in which case the definition of the macro is not removed until the macro exits. A macro does not need to be explicitly purged before it can be redefined. When the assembler encounters a new definition of an existing macro, the exiting macro is automatically purged. The assembler also allows macros to redefine themselves. The new definition will be effective the next time the macro is called but does not interfere with the expansion of the current call. However, you should be very careful when doing this.

Syntax

```
purge       MacroName
```

where:

MacroName is the name of the macro to be removed from memory.

Example 1

```
BigMacro    macro
            ...
            endm
; Code that uses BigMacro
            BigMacro      1,2,3
            ...
            purge BigMacro
; BigMacro no longer exists so can be redefined
```

Example 2

```
Strange     macro
            inform 0,'Hello'
Strange     macro
            inform 0,'GoodBye'
            endm
            Strange
            endm
            ...
            Strange
            Strange
; This will output :-
; Hello
; GoodBye
; GoodBye
```

This is the only information this page contains.

6 Sections and Groups

In simple terms, sections provide a way to structure your programs, and groups provide a way of organising those sections in memory. Sections encourage modular programming as they get you to think in terms of blocks of code. Groups provide great flexibility in placing code in memory.

The concept of sections and groups is similar to the Common Object File Format (COFF) used in Unix based systems. This is a formal definition for the structure of pure binary files in Unix System V. The assembler uses its own implementation of the COFF file structure but the concept of sections as logical blocks of code remains the same. Groups are a collection of one or more sections. They give you control over where a group of sections, and individual sections within that group, reside in memory.

The assembler provides a number of directives and functions that allow you to create and manipulate sections and groups. In particular, the topics covered in this section are:

- Section Names
- Section Alignments
- Allocating Sections to Groups
- Section Alignments
- Changing Sections
- Section Functions
- Setting Group Starting Addresses
- Setting Group Alignments
- Overlaying Groups
- Writing Groups to File
- Group Functions
- Groups and Linking

6.1 Introduction to Sections and Groups

Program code can be divided into three conceptual categories: executable machine code, initialised data and uninitialised data.

Executable machine code	This category is segregated so it can be placed in the PROM.
Initialised program data	This category represents values that the program finds when it starts to execute. It is not write protected.
Uninitialised program data	This category reserves space in memory, it does not represent any specific values. It is read and writable. This is a space-saving feature as there is no need to represent non-values in the program file.

Sections structure your code into logical blocks, usually corresponding to the categories described above although you do not have to adhere strictly to this scheme. You can have as many executable, initialised uninitialised sections as you feel you need.

Groups provide control over placing sections in memory. There are two basic types of group, initialised and uninitialised.

Initialised Groups	Initialised groups contain data or code. All groups are initialised unless the BSS group attribute has been set.
Uninitialised Groups	Uninitialised groups reserve space in memory only, they do not take up any space in the object file. Groups with the BSS attribute are uninitialised.

Sections of a similar nature are assigned to a group so that they can be placed together in the target memory. Target systems usually have different types of memory so using groups helps you use this memory more efficiently. For example, you might want to keep all your variables at the beginning of memory so that they can be direct word addressable. Additionally, space for uninitialised data groups can be reserved high in the memory map. Figure 6 below illustrates the relationship between an object file and a hypothetical target memory.

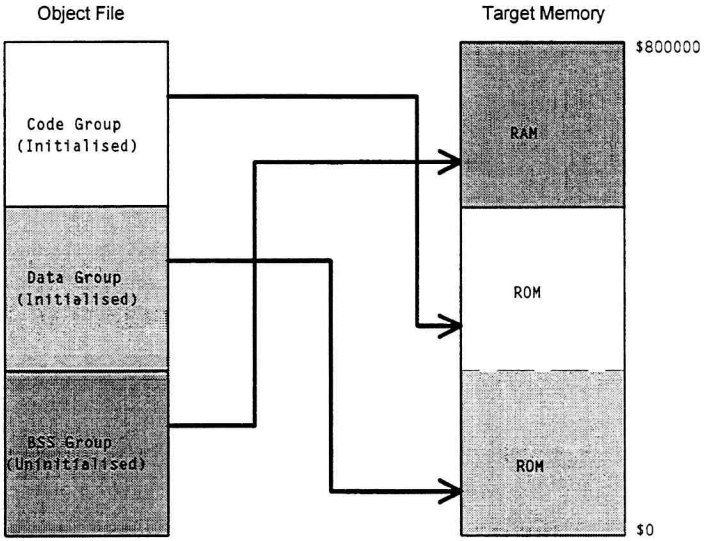


Figure 6. Partitioning target memory into logical blocks.

6.1.1 Section and Group Directives

The assembler provides 4 directives to manipulate sections and groups, summarised below.

SECTION	This directive defines a logical block of code and assigns it to a group.
GROUP	This directive defines a group and its various attributes.
PUSHS and POPS	These directives enable you to change between sections.

6.1.2 Section and Group Functions

The assembler provides 8 functions to manipulate sections and groups, summarised below. These functions are all evaluated at link time except for GROUPSIZE and SECTSIZE which are evaluated immediately and that all addresses refer to assembly addresses except for the GROUPORG function which returns the physical address.

SECT	This function takes a symbol as its parameter and returns the base address of the section in which the symbol is defined.
OFFSET	This function returns the offset of a symbol into its section.
ALIGNMENT	This function returns the offset from the section's alignment type.
GROUPEND and SECTEND	These functions return the end address of a group or section respectively.
GROUPSIZE and SECTSIZE	These functions return the current size of a group or section respectively.
GROUPORG	This function returns the starting address of a group.

6.2 Sections

A section is opened using the SECTION directive. The name of the section and the group to which it belongs are specified in the operand field. The SECTION directive can also be used with an optional size modifier to specify its alignment. A section is closed by opening another section or by reaching the end of your source file.

Syntax

`section[.`*Qualifier*`]` *SectionName* **[**,*GroupName*]
SectionName `section[.`*Qualifier*`]` *Attributes*

where:

Qualifier is an optional qualifier that can be:

`.b`	Byte data
`.w`	Word data
`.l`	Longword data

The SECTION directive operates as SECTION.W if no qualifier is specified.

SectionName is the section name. Any valid name can be used.

GroupName is the group name.

Attributes are the group attributes described on page 129. See page 125 for more information on setting section attributes.

Example

```
section   Vec
dc.l      $00FFFFF0              ; SSP
dc.l      ProgramReStart         ; PC
...
```

Note

The assembler allows great flexibility in using sections and groups to organise your code. However this means that you should be very careful as minor differences in syntax can have a large effect on how the code is structured and placed in memory. The following paragraphs describe what happens under different uses of the SECTION directive.

6.2.1 Section Names

Each section must have a name or the assembler will generate an error. Once you have opened a section you can re-open it as many times as you wish in which case your source code will be concatenated to the end of that section.

Example

```
section    Tables,BssGroup
; Tables allocated to BssGroup

section    Data,LowGroup
; Data allocated to LowGroup

section    Tables
; Tables allocated to BssGroup
```

6.2.2 Section Alignments

When you open a section you can optionally specify its alignment. If you open a section with a particular alignment and later re-open it with a different alignment then the section is treated as being aligned to the largest size. If no alignment is specified the assembler treats the section as word aligned i.e. the same as opening the section with SECTION.W. (The alignment of a group is taken to be the alignment of the largest section within that group.)

Example

```
   section        Data,LowGroup
; Data is word aligned

   section.b      Table1,BssGroup
; Table1 is byte aligned and so BssGroup is byte aligned

   section.w      Table2,BssGroup
; Table2 is word aligned and so now is BssGroup

   section.l      Table1,BssGroup
; Table1 is now long and so now is BssGroup
```

6.2.3 Allocating Sections to Groups

A section is usually opened with a specified section name and group. In this case the section will be appended immediately after the end of the previous section in that group. If the section name has been previously defined for that group then the code will be concatenated with that section. For this reason it is recommended that a section is assigned a group the first time it is defined. Sections opened without specifying a group will be placed in a default unnamed group.

Example

```
BssGroup      group      bss
LowGroup      group      word

              section    Tables,BssGroup
; Tables allocated to BssGroup

              section    Data,LowGroup
; Data allocated to LowGroup

              section    Tables,BssGroup
; Code concatenated with Tables in BssGroup
```

6.2.4 Section Attributes

It is sometimes convenient to write a project where each group contains only one section. The assembler provides a short-hand way of defining a group with a single section by enabling section attributes to be set directly, dispensing with the GROUP directive. This feature reduces the effort required to implement simple groups and can be used whether or not you are linking.

Note

Defining section attributes in this way uses an alternative SECTION syntax. First, the section name is now defined in the label field and second, section attributes are specified in the operand field with multiple attributes separated by white space.

Example

```
Code    section    word org($8000)
        ...
```

6.2.5 Changing Sections

The PUSHS and POPS directives are used to change sections, the PUSHS directive saves the current section and the POPS directive restores the section previously saved using PUSHS.

Syntax

```
pushs
```

Example 1

```
            pushs
            section  FastRAM
PicTexture  ds.w     1
DonelString ds.w     100
DonelString ds.w     100
            pops
```

Example 2

```
            pushs
            section  Data
LevelText   dc.b     0,11,19
            dc.b     'Aggressive',0
            dc.b     'Neutral',0
            dc.b     'Passive',0
            even
            pops
```

Example 3

The MARKPLACE macro puts the current PC into a separate section.

```
MarkPlace   macro
            local    Temp
Temp        equ      *
            pushs
            section  MarkSection
            dc.l     Temp
            pops
            endm
```

6.2.6 Section Functions

The assembler provides a number of functions to find the location of symbols in a section.

SECT and OFFSET

The SECT function returns the base address of the section in which its parameter is defined. This cannot be evaluated until link time when the section definitions are complete.

The OFFSET function returns the offset of a symbol in its section. This is again evaluated at link time so that SECT(*Symbol*) + OFFSET(*Symbol*) = *Symbol*.

A section is often split into several distinct parts spread throughout your source code. Each part is known as a section *Fragment*. In the example below `Sect1` is split into two fragments, the first containing $100 bytes of code and the second containing $150 bytes. If you want to know the offset of a symbol into a fragment you need to put a label at the beginning of the fragment and then do the subtraction yourself. In the example below the offset into the fragment of `Label1` is given by `Label1-Marker=$120`.

Example

```
          org             $10000
          section         Sect1
; $100 bytes of code here.
          section         Sect2
          ...
          section         Sect1
Marker
; $120 bytes of code here.
Label1    offset(Label1)
; $30 bytes of code here
          section         Main
          sect(Sect1)
          sect(Sect2)
          sect(Label1)
          ...
          end
```

At link time `offset(Label1)` returns $100+$120=$220, `sect(Sect1)` returns $10000, `sect(Sect2)` returns $10000+$100+$120+$30=$10250 and `sect(Label1)` returns $10000.

ALIGNMENT

The ALIGNMENT function returns the offset of its argument from the section's alignment type (byte, word or longword). In a byte aligned section the function will always return 0. In a word aligned section it will return 0 or 1 and in a longword aligned section it will return 0, 1, 2 or 3. ALIGNMENT is typically used to determine whether the PC is odd or even as in the example below.

Example

```
if alignment(*)&1       ;If PC is odd pad with
  dc.b 0                ;zero to even boundary
endif
```

SECTEND

See also GROUPEND on page 133.

The SECTEND function returns the end address of a section. The address is not evaluated until link time.

Example

```
                move.l      #sectend(s1),d0
;evaluates the end address of s1
                move.l      #sectend(s1)-sect(s1),d0
;evaluates the total size of section s1
```

SECTSIZE

See also GROUPSIZE on page 133.

The SECTSIZE function returns the current size of a section. This is evaluated immediately and so does not reflect the final size of the section but the size of all fragments so far seen in the assembly.

Example

```
                dcb.b       16384-sectsize(Data),$FF
; pads section up to 16k in size
```

6.3 Groups

The GROUP directive is used to allocate several sections contiguously in memory. The assembler allows you to set group attributes using the GROUP directive in your source code. The attributes are specified in the operand field with multiple attributes separated by commas.

Syntax

GroupName `group` [`org` *Address* | `obj` | `size` *Size* | `bss` | `word` | `file` *Filename* | `over` | `pad` *Value* | `scatter` *Start,Length*[,*Start,Length*]]

where:

GroupName	is any label defined by this statement that is not already a section name.
`org` *Address*	Specifies the address in memory, *Address,* at which to place the group. If no address is specified then the group will be placed in memory after any previously defined group.
`obj`	Allows a group to use assembly with offset i.e. group runs at address different to normal contiguous address.
`size` *Size*	Specifies the maximum size, *Size*, of a group. Enabling the PAD attribute forces the assembler to pad to this size.
`bss`	Defines a uninitialised data group.
`word`	Tells the assembler that a group can be absolute word addressed i.e. the first 32K or the last 32K of the address space can be compacted into a word. Enabling this option allows all references to items in that group to be accessed via word length offsets.
`file` *Filename*	is used to write the contents of a group to a binary file called *Filename*. All subsequent groups will be written to the same file until a different file is specified.

`over`	is used to overlay groups i.e. causes specified groups to start at same address in memory.
`pad` *Value*	Pads the group to the declared size with the byte value defined by the evaluable expression *Value*. This is useful in PROM burning where instead of burning the 1's of the unused space down to zero, the unused space is padded with 0's so the burn takes less time. Only valid if the `size` attribute has been specified.
`scatter` *Start,Length* [,*Start,Length*]...	Sections and groups normally follow in the order they are encountered. This attribute defines a group that is not contiguous in memory but where each section is allocated a starting address *Start*, and an amount of memory given by *Length*. Multiple parameter pairs are separated by commas. Group ordering is as before but sections within a group are reordered to achieve an optimal packing.

6.3.1 Group Starting Address

There are two ways to set the starting address of a group, which you can mix freely. The first method uses the ORG directive to set the start address of the program. Groups are then loaded contiguously into memory starting from the address specified in the ORG directive. The second method allows you to set individual starting addresses for groups. This is done by setting each groups ORG attribute with the required starting address. You cannot use ORG inside a section but if you do not use sections and groups then ORG can be used anywhere.

Groups are normally placed in memory in the order in which they are encountered in your program with groups not having an explicit starting address placed at the end of the previously declared group. Sections within each group are placed in memory in the order in which they are specified. Section fragments from different source files are concatenated in the order in which the source files are specified.

Example 1

This example uses a single ORG statement to set the starting address for the first group with subsequent groups loaded immediately after the preceding one.

```
         org      $1000          ;Program starting address
         ...
Code     group                   ;Loaded at $1000
Data     group                   ;Loaded immediately after
         ...                     ;the end of Code group
```

Example 2

This example sets individual group starting addresses.

```
Code     group    org($1000)     ;Loaded at $1000
Data     group    org($2000)     ;Loaded at $2000
         ...
```

Example 3

```
         org      $1000
Code1    group                   ;Loaded at $1000
Code2    group                   ;Loaded after Code1
Data1    group    org $2000      ;Loaded at $2000
Data2    group                   ;Loaded after Data2
         ...
```

6.3.2 Setting Group Alignments

The alignment of a group is determined by the widest alignment of sections within that group. For example, if a group contains a byte aligned section and a word aligned section then the group will be word aligned. Similarly, the alignment of a section is determined by the widest alignment of its component fragments.

Example

```
BssGroup   group

   section Table1,BssGroup
; Table1 is word aligned and so BssGroup is word
; aligned
   section.1      Table1,BssGroup
; Table1 is now long so BssGroup is now long
```

6.3.3 Writing Groups to File

The assembler allows you to write individual groups to separate pure binary files using the FILE attribute; all groups declared after a group with a file attribute will be written to that file until a new file is specified. The FILE attribute can be used in conduction with the OVER attribute to put overlays into separate files with all the overlays having the same start address.

Example

```
Code        group
Data        group
Overlay1a   group   org $8000,file('overlay1.bin')
Overlay1b   group   ; Also goes into overlay1.bin
Overlay2    group   org $8000,file('overlay2.bin')
```

The output is in pure binary format and no default extension is added to file names.

6.3.4 Overlaying Groups

The OVER attribute enables you overlay groups i.e. to have several groups starting at the same address. All the groups will have the same starting address and enough space will be left for the largest group.

Example

```
Overlay1   group
Overlay2   group   over(Overlay1)
Overlay3   group   over(Overlay2)
```

6.3.5 Group Functions

Groupend

See also SECTEND on page 128.

This function returns the end address of a group, evaluated at link time.

Example

This example evaluates the total size of the DATA group.

```
move.l   #groupend(Data)-grouporg(Data),d0
```

Groupsize

See also SECTSIZE on page 128.

This function returns the size of a group. This is evaluated immediately and so does not reflect the final size of the group.

Example

There are 2 equivalent ways to evaluate the size of the CODE group:

```
move.l    #groupsize(Code),d0
move.l    #groupend(g1)-grouporg(g1),d0
```

Grouporg

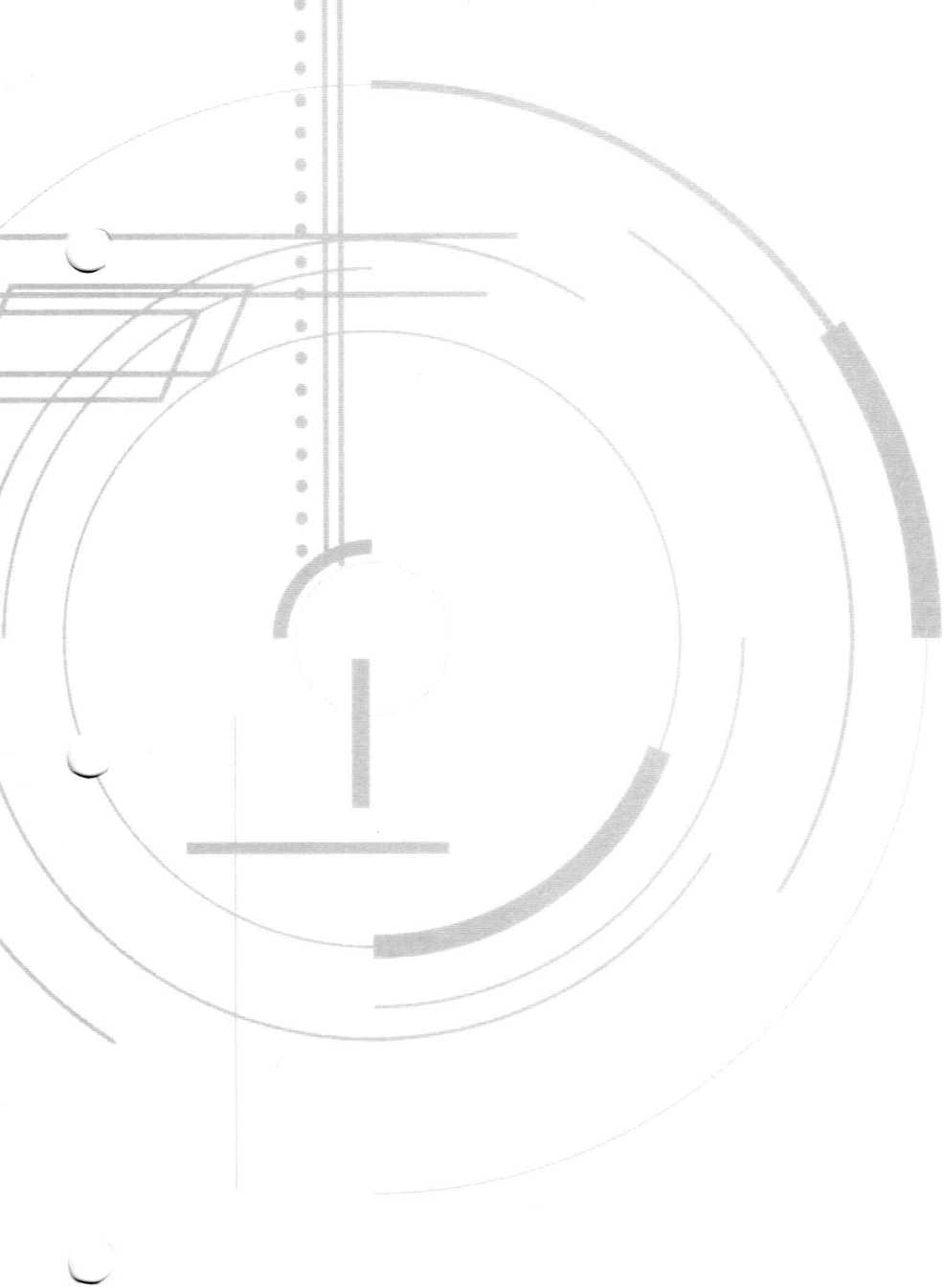
The GROUPORG function returns the physical address at which a group has been placed in memory. This function is used mainly when you want to relocate a group in memory with the OBJ directive and need to obtain the groups starting address (note that GROUPORG is unaffected by OBJ).

Example

This example evaluates the size of the CODE group.

```
move.l    #groupend(Code)-grouporg(Code),d0
```

This is the only information this page contains.



7 The Debugger

The debugger enables the target machine to be remotely debugged via the SCSI hardware. Access to symbols and a powerful expression evaluator, (which includes expression formatting) enable full symbolic debugging to be performed on the target machine. You can source level debug C, assembler and mixed language projects and use the Mixed code window to view both the original source code and the actual code the processor is running in a single window. Concurrent debugging of multiple processors can be performed on a single screen enabling code for one processor to be monitored whilst simultaneously monitoring the state and memory contents of another.

The debugger features a multiple, overlapping window interface enabling several areas of memory, variables or expressions to be examined at once. Each window type has its own local menu supported by comprehensive mouse and keyboard access to menu functions and keyboard shortcuts to commonly used functions. Multiple configurations of window layouts and associated debugger states can be saved and restored at any time.

The debugger also provides extensive breakpoint support including single-shot, and breakpoints which log output to monitor the state of variables and registers as the program runs. Complex conditions can be attached to breakpoints allowing several points to be monitored at once.

7.1 Exception Vectors

The link software must be in control of the target to enable it to be debugged. This is achieved using several of the targets exception vectors, shown in Table 14 below. An exception is a special condition that takes priority over normal processing such as an illegal instruction or an address error. Whenever one of the given exceptions occurs the debugger can gain control of the target. As exceptions should not occur during normal program execution a Trap0 should be executed at regular intervals to ensure that the debugger can gain control of the target.

Vector Number	Address	Function
2	0008	Bus error.
3	000C	Address error.
4	0010	Illegal instruction.
5	0014	Zero divide.
6	0018	CHK instruction.
7	001C	TRAPV instruction.
8	0020	Privilege violation.
9	0024	Trace.
32	0080	Trap0.

Table 14. Exception vectors used by the debugger.

7.2 Running the Debugger

This section describes how to invoke the debugger from the command-line, including the switches used to control COFF file downloading, saving and restoring previous debugging sessions and some examples on how to invoke the debugger.

7.2.1 Command-line Syntax

The command-line consists of two optional switches controlling the session and object files used by the debugger. The switches must be separated by white space but no white space is allowed within the argument of a switch. The syntax is:

`snbug68k` [[-|/]`i`=*SessionFile*] [[-|/][`t`*n*][`b`][`n`][:*ObjectFile*]]

The files used by the debugger and their default extensions are given in Table 15 below. Note that session files can take any extension and source files can take any extension specified during the installation process.

Filename	Extension	Description
SessionFile	INI	This file contains the information needed to restore a previous debugging session. If no file is specified the debugger will look for the default file SNBUG68K.INI.
ObjectFile	COF	The object file. This contains binary and optionally, source level debug and symbol table information (referred to as *debug info* from here on), produced by the assembler.

Table 15. Files used by the debugger.

The debugger has two optional command-line switches, described in table 15 below.

Switch	Description
`i=`*SessionFile*	*Save Project Info.* This switch causes the debugger to create a session save file containing information about currently connected targets, the object files in use for each target, update rates, breakpoints, watch expressions, log expressions and window positions and displays. If no session file is specified the information will be saved in the default file SNBUG68K.INI.
`t`*n*`[b][n][:`*ObjectFile*`]`	*Specify Target and Object File.* This switch specifies the target, identified by its SCSI device number *n* (0-7) and the object file, *ObjectFile*, to download. Using this switch to specify an object file for a target will override the setting in the session file. Options are downloading the binary from the object file (`b`) and suppressing debug information (`n`). Note that using the `n` option without also using the `b` option will have no effect as no code or symbols are required.

Table 16. Debugger command-line switches.

Example 1

The following example invokes the debugger and restores the debugging session according to the information contained in the session file INITFILE.INI.

```
snbug68k   -i=initfile
```

Example 2

This example invokes the debugger and restores the debugging session according to the default session file SNBUG68K.INI (no `i` switch). The binary (`...b...`) from the file TEST.COF will be downloaded to target 7 (`-t7...:test`) but no symbolic information is loaded (`...n...`).

```
snbug68k   /t7bn:test
```

Example 3

The following example illustrates how *not* to invoke the debugger as using the `n` option without also using the `b` option will do nothing because no code or symbols are required.

```
snbug68k   -t7n:test
```

7.2.2 Session Files

The debugger gets its initial state from a *session file* that contains information used to configure a debugging session; connected targets, menu options, colour schemes, window layout, cursor positions and memory ranges. Optionally, project specific settings such as defined breakpoints, watch expressions and the object file in use by each target can also be saved. Invoking the debugger using a session file containing project specific information restores project items if present and the COFF file binary. Any available debug info will be downloaded automatically provided the `'do_syms'` entry in the project section of the session file is non-zero. Note that command-line options override those contained in the session file. An unlimited number of session files can be saved and loaded at any time during a debugging session providing custom configurations for different debugging requirements.

Default settings are contained in the template session file SNBUG.CFG. This file *must* be in the same directory as the debugger and should not be modified. These defaults are written to the first session file created, SNBUG68K.INI by default and SNBUG.CFG will no longer be used unless the debugger is invoked with a session file that does not contain memory range information. Information in session (.INI) files can only be changed using a text editor so be careful to retain the format shown here so as not to cause unexpected behaviour in the debugger.

Memory Ranges

The debugger uses information about valid memory ranges to prevent it accessing areas that would cause an address error. Default read, write and readwrite memory areas for various targets are contained in the SNBUG.CFG file. The format of the memory range section found in a .INI file is shown below. The `Read`, `Write` and `ReadWrite` specifers denote valid read, write and readwrite ranges respecitvely and there can be as many ranges as required. The `Memory` specifer allows additional memory ranges to be specified at other locations within the .CFG file or

in another text file. Any memory ranges following the memory specifer are added to those found in the current .INI file.

```
[memory_TargetNumber]
Read=StartAddress,EndAddress,[Expression]
Write=StartAddress,EndAddress,[Expression]
ReadWrite=StartAddress,EndAddress,[Expresssion]
Memory=[Filename : ] Tag
```

where:

TargetNumber	is the target SCSI device number.
StartAddress	is the start address of a valid memory range.
EndAddress	is the end addresses of valid memory range.
Expression	is any valid expression and is used to allocate memory shared between multiple processors.
Filename	is an optional file containing memory range information not already specified in SNBUG.CFG.
Tag	is the name of the section searched for in SNBUG.CFG or *Filename* if specified.

Example

This example defines memory ranges for target 7 with addresses in hexadecimal. The expression sets the read range only if bit $80 is set in byte C2BF.

```
[memory_7]
Read=0x00001000,0x00007FFF,[C2BF] & $80
Write=0x00008000,0x0000BFFF
ReadWrite=0x0000C000,0x0000FFFF
```

Window Attributes

SNBUG.CFG also contains information about window attributes. Every processor type has a super-section that contains default position, size and colour attributes for each window. The format of a super section is shown below.

```
[[ProcessorID_Windows]]
[WindowType]
Entries
...
[WindowType]
Entries
...
[[]]
```

where:

ProcessorID	is the name used to identify the processor.
WindowType	is either COD_WIND, DIS_WIND, SRC_WIND, REG_WIND, MEM_WIND, WCH_WIND, LOG_WIND or FIL_WIND.

Example

The following example shows a super section for the 68000 that defines the size and position for the Disassembly window.

```
[[68000_Windows]]
[dis_wind]
Row=10
Column=10
Width=35
Depth=12
[Reg_Wind]
...
[[]]
```

7.3 The Debugger Interface

7.3.1 Selecting Targets

Because the debugger can support multiple targets a target must first be selected before any debugging can be performed on it. Selecting a target is often the first action performed after invoking the debugger and if working with multiple processor systems you will often want to switch between targets.

There are three ways to select a target (and initialise it if it has not yet been connected). Press Shift+Target Number to select a target from a dialog box or in the Main window (described later) choose Target|Select from the menu or click the left mouse button on the targets status line. The currently selected target is highlighted in red in the Main window and any other windows associated with the target will be brought to the front. The selected target becomes the foreground target and all debugging actions apply to this target.

7.3.2 Working with Windows

The multiple windowing interface allows as many windows of each type to be open at any one time as required, limited only by the amount of memory available (with the exception of Register windows which are limited to 1 per target). You can move or resize any active debugger window and edit any value in the Memory and Register windows.

Opening Windows

To open a window select a target as discussed above then choose the type of window required from the Window menu (Ctrl+N). To select an existing window use Ctrl+W or click the left mouse button on any part of the required window. This automatically forces the currently selected target to change accordingly. Each debugging window displays the target number to which it belongs, the window type and the window number for that target in the form:

TargetNumber:*WindowName*-*WindowNumber*

Resizing Windows

To resize both the depth and width of a window click and hold the left mouse button on the bottom right corner. Drag the corner to the desired position and release the mouse button. Similarly, to resize the width or depth only drag and release the right or bottom size handle respectively (this feature is disabled when a scroll bar is visible).

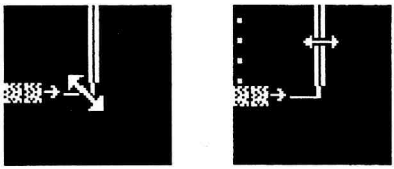


Figure 7. Resizing windows.

Moving Windows

Move windows by placing the pointer on the title bar and clicking and holding down the left mouse button. Drag the window to its new position and release the mouse button.

Selecting Windows

To cycle through the open windows for the currently selected target use Ctrl+Arrows. Each window for a target has its own number, starting from 0. Any of the first 10 windows opened for a target can be selected using Ctrl+0..9.

7.4 The Main Window

This is the main debugger control window and displays a list of currently connected targets and their status as shown in Figure 8 below. The menus provide access to general debugging tasks such as loading files, saving sessions, selecting targets, running code and opening windows. Menu selections apply to the currently selected target, highlighted red in the target list and pressing Return or Enter on a highlighted target displays the Target Configure Dialog box described later.

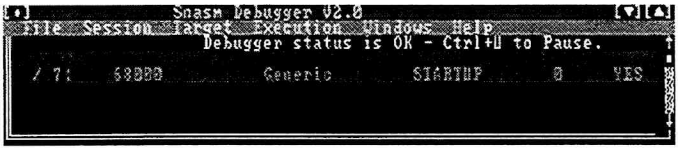


Figure 8. The Main debugger window.

The Help menu contains one item, the About box, accessed from any window using Ctrl+V. This displays the version number and compilation date and time which may be asked for during a technical support call. The remaining six menus, File, Session, Target, Execution, Breakpoints and Windows are now described in turn.

7.4.1 File Menu

The File menu shown below controls file loading and exiting the debugger.

Load Binary & Debug Info...	Shift+Ctrl+C
Load Debug Info Only...	Ctrl+C
Send Binary...	Shift+S
Get Binary...	Shift+G
Disassemble to File...	Shift+D
Hex Dump to File...	Shift+H
Shell to DOS	Ctrl+Z
Prompt and Exit	F3
Save and Exit...	Ctrl+X
Quit (No Save)	Ctrl+Q

Load Binary & Debug Info... (Shift+Ctrl+C) displays a file selector dialog requesting the name of the object file which will be used to send

the binary code to the currently selected target. Any debug info included in the file by the assembler will be loaded by the debugger.

Load Debug Info Only... (Ctrl+C) performs the same action as above but does not send any binary code to the target. The binary is assumed to be have been previously sent or already present i.e. it was assembled to the target.

Send Binary (Shift+S) displays the Binary Transfer dialog box requesting the name of the file to send, the start address and the length of file. The default start address is $00000000 and the default length is $ffffffff (i.e. the entire file).

Get Binary... (Shift+G) displays the Binary Transfer dialog box requesting the name of the file to get, the start address and the length of file. The default start address is $00000000 and the default length is $00100000.

Disassemble to File... (Shift+D) writes a disassembly of an area of memory to file. The dialog box requests the filename, the start address and the length of memory. The options are the same as the items from the Format Menu in the Disassembly window; Show Symbols, Show Upper Case, Show T-States, Show Intr. Code, Decimal, Hex.

Hex Dump to File... (Shift+H) writes an area of memory to file. The options are the same as the items from the Format Menu in the Memory window; Show ASCII, Bytes per Line, Show Bytes, Show Words, Show Longs.

Shell to DOS... (Shift+Z) Type EXIT to return to the debugger.

Prompt and Exit... (F3) displays a prompt requesting the name of the session file to save, by default the last saved session file or SNBUG658.INI if no custom session files have yet been saved. To enable the project to be restored exactly as it is, select the Save Project Information tick box to save the COFF information in use and any breakpoints, watch expressions, log expressions and so on in addition to window positions and displays.

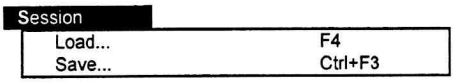
Save and Exit... (Ctrl+X) exits the debugger and saves the session information to the default session file SNBUG658.INI, not the current INI file. Project information will be saved only if it was restored from a session file (either SNBUG658.INI or a custom INI file) or the Project

Information box was ticked during a previous session save (using Ctrl+F3).

Quit (no Save) (Ctrl+Q) quits the debugger without any prompts or saving any session information.

7.4.2 Session Menu

The Session menu shown below enables sessions to be saved and loaded at any point during a debugging session.



Load... (F4) loads a new session file, discards the current setup without saving it and restores the settings from the file chosen in the file selector dialog shown in Figure 9 below. If the session file was saved with Save Project Information ticked then the binary file, breakpoints, log and watch expressions and debug info will also be restored if present.

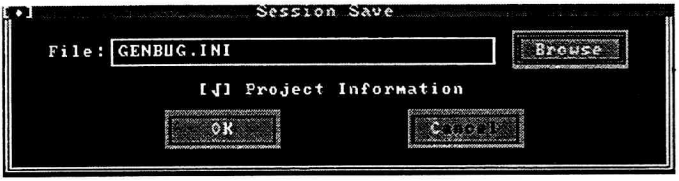
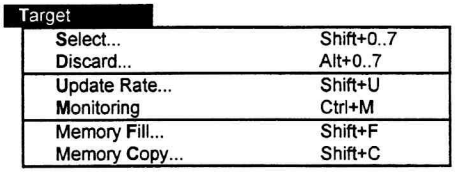


Figure 9. The Session Save dialog box.

Save... (Ctrl+F3) saves the debugging session in its current state to a specified file, optionally including project information. The default file is the last saved session file or SNBUG68K.INI if no session file has yet been saved.

7.4.3 Target Menu

The Target menu shown below controls target connection, update rates and monitoring.



Select... (Shift+0..7) displays the dialog box shown in Figure 10 below where the radio buttons correspond to available target device numbers.

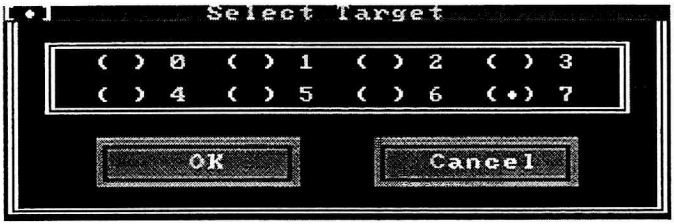


Figure 10. The Target Select dialog box.

The default radio button is the current target or button 7 if no targets are currently selected. Selecting a target will cause any windows associated with it to be brought to the front, the active window being the one last worked in. Selecting a non-existent target will create it and add it to the Main window status display. Selecting a target can also be done by clicking the left mouse button on the targets information line in the Main window or by pressing Shift+Target Number from anywhere within the debugger.

Discard... (Alt+0..7) prompts for a target to discard. Discarding a target will close all windows for the target and disconnect it.

Update Rate... (Shift+U) selecting this from the menu displays the Update Rate dialog box shown in Figure 11 below which is used set the foreground and background update rate for the currently selected target. Using Shift+U does not invoke the dialog box but instead toggles continuous updating on and off. The foreground update rate is the rate at which the target is updated when it is the currently selected target.

Similarly, the background rate is the rate at which the target is updated when it is not the currently selected target. In the Main window, the status line for each target displays the current update rate or an X if continuous update is disabled.

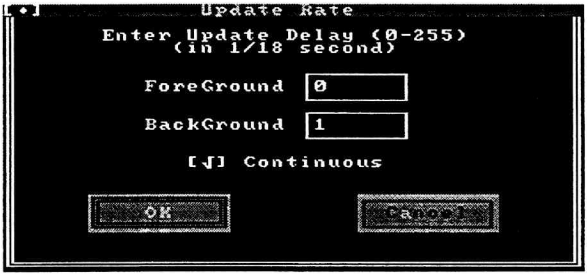


Figure 11. The Update Rate dialog box.

If the Continuous check box is enabled the target will be continuously updated. Note that the foreground and background update rates now refer to the *minimum* update rate for the target. If a continuously refreshing window has a higher refresh rate and requires data from the target then the target will be updated at this higher rate (see individual window descriptions for setting refresh rates). If the Continuous check box is not enabled the target update rate will be the highest refresh rates of its windows . If no windows are open for this target it will not be updated.


Note

The behaviour of conditional breakpoints may be affected if slow update rates are specified for a target or continuous update is disabled. This is because there will be significant delays between breakpoints halting the execution of code on the target and the debugger detecting and processing them.

Monitoring... (Ctrl+M) disables the connection or ability to connect to, a target. No target updating or window refreshing takes place and connection to the target cannot take place without first turning monitoring back on. Turning off monitoring is the equivalent of disabling continuous update for a particular target and ignoring requests for a forced update such as looking at a memory range. This feature can also be used to disable attempts to continuously attempt reconnection to a target after a SCSI error without the need to discard the target.

Memory Fill... (Shift+F) fills a range of memory with a specifed byte. The specified memory ranges can contain invalid areas but these areas will not be read from or written to.

Memory Copy... (Shift+C) copies a range of memory to another location. The copy destination memory must not overlap the copy source memory. The specified memory ranges can contain invalid areas but these areas will not be read from or written to.

7.4.4 Execution Menu

See also Code Windows on page 152.

The Execution menu shown below provides a subset of the Execution menu found in Code windows.

Execution	
Run from PC	F9
Stop Program	Esc
Stop (DMA & Interrupts)	Shift+Esc
Reset Processor	Shift+Ctrl+R

Run from PC (F9) starts the target executing code from the current position of the PC.

Run to Address (Shift+F9) executes the current program until it reaches a specified address or executes a specified source file until it reaches a given line number.

Single Step (F7) In disassembled code, the target executes the instruction at the PC with the current register values and then stops. In source code, the target executes the instruction at the PC with the current register values. the target stops when all low level assembly instructions generated by the single source instruction have been executed.

Step Into (Shift+F9) In disassembled code, the target executes the instruction at the PC with the current register values and then stops. In source code, the target executes the instruction at the PC with the current register values and then stops at each individually generated assembler instruction.

Step Over (F8) In disassembled code, the target executes the instruction at the PC with the current register values and then stops. In source code, the target executes the instruction at the PC with the current register values and then stops when the source file reference has changed.

Unstep (Ctrl+F7) Untraces individually single stepped instructions only.

Stop (Esc) stops the target executing code as soon as possible and leaves the PC at the start of the next instruction that would have been executed.

Stop Program (Esc) stops the target executing code as soon as possible and leaves the PC at the start of the next instruction that would have been executed.

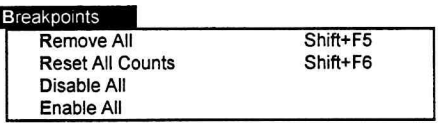
Stop (DMA & Interrupts) (Shift+Esc) stops the target in the same way as Stop Program but also disables all DMA accesses and stops all interrupts so that the target is forced into a safe state.

Reset Processor (Shift+Ctrl+R) causes the currently selected target to perform a processor reset. This is simulated on 68000 systems such that the SSP and PC registers are reloaded from locations 0 and 4 respectively and the SR register is loaded with a value of $2700.

Save Registers (Ctrl+S) saves the current contents of the target registers.

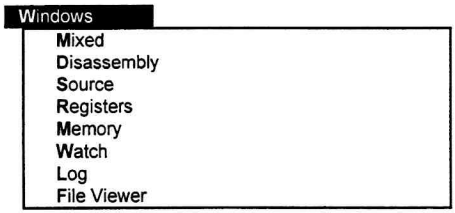
Retrieve Registers (Ctrl+R) restores the previously saved register contents.

7.4.5 Breakpoint Menu



7.4.6 Windows Menu

The Windows menu creates a new window for a target which can be any one of those shown below. A target can have an unlimited number and mixture of windows with the restriction that only one Register window is permitted per target. Mixed, Disassembly and Source windows are discussed together in *Code Windows* on page 152



7.5 Code Windows

The debugger supports three types of code window: Mixed, Disassembly and Source. The Mixed window displays source code in the upper region and the corresponding disassembled code in the lower region. Click the left mouse button in the relevant region or use Space to toggle the focus between the two regions. To select a line click the left mouse button on the line in the left hand side of the display. To start the display at a given line double click on the line in the left hand side of the display. The Disassembly window displays code in disassembled format and the Source window displays the original source code. Disassembled code can optionally show symbols in place of hexadecimal values to make the code more readable.

See also Breakpoints on page 158 and Tracing on page 160.

All code windows provide the facility to run, trace and set breakpoints. For example, select the source region of a mixed window to trace code at source level and view the assembly language statements corresponding to each source statement executed in the disassembly region below. Alternatively, in the same window select the disassembly region to trace at the instruction level and view the source statement that generated each instruction or group of instructions.

7.5.1 Mixed Window

The Mixed window (called Code on the title bar) shown in Figure 12 below has two regions. The upper region displays source code (if debug info is loaded) and the corresponding disassembly of the target code in the lower region. If the target's PC is at one of the lines in the disassembly it is marked with a '>'. Selecting a source instruction in the upper region highlights the correspondingly generated instruction(s) in the lower region. If a program contains macros or C source code components there may be a one to many relationship between source and disassembly instructions.

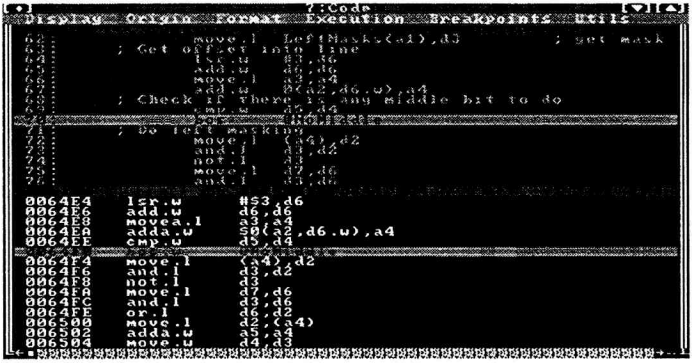


Figure 12. The Mixed (Code) window.

The Display menu controls the window refresh rate and how the contents of the upper and lower window displays are centred.

Display

Switch Active	Space
Update Rate...	Shift+U
Centre on Trace	
Centre on Bpoint	
Centre on Instr. Error	
Centre on Scroll	

The Origin menu controls the starting position of the upper or lower window display.

Origin

Goto Address...	Ctrl+G
Go to Cursor	Home
Toggle Window Lock	Ctrl+L

The Format menu determines the format of items displayed in the window.

Format

Show Symbols	
Show Upper-Case	
Show T-States	Ctrl+T
Show Instr. Code	Ctrl+I
Decimal	Ctrl+D
Hex	Ctrl+H

See also Tracing on page 160.

The Execution menu provides the ability to run code and trace program execution. In addition, the target can be stopped and reset and the contents of its registers saved and retrieved (only one level of save).

Execution	
Run from PC	F9
Run to Address	Shift+F9
Run to Cursor	F6
Single Step	F7
Step Into	Shift+F7
Step Over	F8
Unstep	Ctrl+F7
Stop	Esc
Stop (DMA & Interrupts)	Shift+Esc
Reset Processor	Ctrl+Shift+R
Save Registers	Ctrl+S
Retrieve Registers	Ctrl+R

See also Breakpoints on page 158.

The Breakpoints menu provides the ability to set and configure individual breakpoints and to remove all breakpoints and reset all breakpoint counts.

Breakpoints	
Toggle at Cursor	F5
Configure...	Ctrl+F5
Remove All	Shift+F5

The Utils menu provides access to the expression evaluator.

Utils	
Expression Calculator...	Ctrl+E

7.5.2 Disassembly Window

The Disassembly window, equivalent to the lower region of the Mixed window, displays a full window of the disassembled memory at the target.

7.5.3 Source Window

The Source window, equivalent to the upper region of the Mixed window, displays a full window of the source code.

7.6 The Memory Window

The Memory window shown in Figure 13 below displays the contents of a given address in either byte word or long format. Local menu items set the address to view, the format of memory contents and change the contents of memory locations. Additional items control the update rate and invoke the expression calculator.

Figure 13. The Memory window.

The default display shows memory contents as bytes and with the corresponding ASCII display on the right-hand side. To show memory contents as words use Shift+W; use Shift+L for longs and Shift+B to return to bytes. Clicking on an ASCII character moves the cursor to the corresponding byte position in the memory display. To turn the ASCII display on and off use Alt+A. The Goto Pointer @ Cursor menu item takes the value of the long under the current cursor position and moves the cursor to this address. If the address is already visible in the window the cursor will be moved to the new address. If the address is not visible, the window will be updated so that the display starts from the address.

To change the contents of a memory location type the new value at the cursor or press Enter to enter an expression. The result of the expression will be truncated to fit the currently selected display format (either byte, word or long). To increment or decrement a value use the + and - keys.

7.7 The Registers Window

The Registers window displays the contents of the processor's general registers. Choose **Registers** from the **Windows** menu to open the Registers window, shown in Figure 18 below. The window defaults to a horizontal, hexadecimal display. You can edit the value of any of the registers by entering the new value at the cursor position.

Figure 14. The Registers window.

The **Edit** menu provides access to commands for changing register values. Increment or decrement the value of a register using + and -, or press **Enter** to enter an expression. The result of the expression will be truncated to fit the size of the register under the cursor.To save the contents of the registers use **Ctrl+S** and **Ctrl+R** to retrieve them. The processor is reset using **Shift+Ctrl+R**. To invoke the expression evaluator use **Ctrl+E**.

To change the window refresh rate choose **Update** from the **Display** menu or use **Shift+U**. To change the window name choose Window Name from the display menu or use **Shift+N**. To print the contents of the registers choose **Print** from the **Utils** menu or use **Ctrl+P**.

7.8 Other Windows

7.8.1 The Watch Window

For more information on Watch and Log expressions see page 163.

Watch expressions are used to determine the point at which a value changes in memory. The Watch Window displays a list of all the watch expressions set and is dynamically updated. To add a watch expression use Ctrl+A or press Enter on an empty watch line which will invoke the watch expression editor. The current expression can be edited by pressing Enter or deleted using Ctrl+D. You can use the cursor keys to move between expression entries.

7.8.2 The Breakpoints Window

For more information on Breakpoints see page 158.

The Breakpoints Window displays a list of all breakpoints watch set and is dynamically updated. To add a breakpoint use Ctrl+A or press Enter on an empty breakpoint line which will invoke the breakpoint configuration dialog. The current breakpoint can be edited by pressing Enter or deleted using Ctrl+D. You can use the cursor keys to move between expression entries.

7.8.3 The Log Window

The Log window displays a list of log expressions evaluated as a result of triggering a breakpoint. The log expressions are set from the Breakpoint Configuration dialog box for individual breakpoints in code windows. Viewing the resulting contents of the Log window can be helpful if you need to analyse the status of your program at specified points during its execution. This can be likened to a simple form of profiling code. Use Ctrl+P to print current window contents and optionally, in addition to the displayed information, all generated expressions may be sent to a file specified for each individual log window.

7.8.4 The File Viewer Window

The File Viewer window displays text files, usually source code. Files cannot be modified in this window.

7.9 Breakpoints

The debugger provides a powerful and flexible breakpointing facility encompassing simple single-shot to complex conditionals. To set a breakpoint open a code window and use F5 or Ctrl+F5 on a highlighted instruction. In a code window, pressing F5 or clicking the left mouse button in the left-hand margin sets a default breakpoint. Default breakpoints are permanent with no attached conditions or counts i.e. when a default breakpoint is encountered during program execution the only resulting action will be the halting of execution. These can be set whilst the target is running and take effect immediately. In a code window, pressing Ctrl+F5 configures an existing breakpoint or sets a new breakpoint and invokes the breakpoint configuration dialog box. The breakpoint will not take effect until configuraton is complete. This allows you to specify individual breakpoint behaviour according to requirements. To clear a breakpoint highlight it and use F5 or click the left mouse button in the left-hand margin; to clear all breakpoints use Shift+F5. To reset the current count for all breakpoints use Shift+F6.

Breakpoints are controlled through the Breakpoint Configuration dialog box shown in Figure 15 below. There are two sets of check boxes for configuring the type of breakpoints and the action they take: Condition and Action.

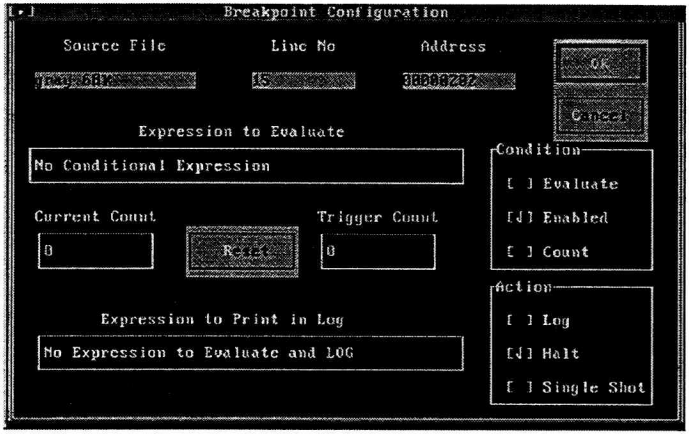


Figure 15. The Breakpoint Configuration dialog box.

In the Condition check box:

Use Evaluate to evaluate the expression. Clearing this check box retains the conditional expression but does not evaluate it i.e. the breakpoint is treated as unconditional.

Use Enabled to enable or temporarily ignore a breakpoint without the need to discard it, thereby keeping all the current settings. This is the most powerful type of breakpoint. They allow an action at a particular address only if a set of conditions apply. Each conditional breakpoint has an expression associated with it which is evaluated each time the breakpoint is reached. Only if the expression evaluates to a non-zero value i.e. True, will an action be taken. If an invalid expression is entered, an expression error will be detected on evaluation and the breakpoint disabled. The evaluation will be forced to True as a result, an immediate unconditional breakpoint will occur and a warning issued.

Use Count to configure the breakpoint as a counter. Each time such a breakpoint is reached a counter associated with it is incremented and displayed in the configuration dialog box. These breakpoints are useful for profiling in that they act like monitors.

If both an expression string and a Count or Trigger Count are specified and relevant condition boxes are ticked, then each time the expression evaluates to True i.e. non-zero, the count will be incremented. On the value of Current Count reaching the value of Trigger Count the whole conditional breakpoint is deemed True and the specified action will be performed.

In the Action Check box:

Use Log to send the specified expression to a Log window every time the breakpoint is hit.

Use Halt to stop program execution after the breakpoint instruction has been executed.

Use Single Shot to set one-off breakpoints which are cleared when executed, otherwise the breakpoint will remain set.

The Reset button sets the current count to zero.

7.9.1 Tracing

All trace operations take precedence over breakpoints i.e. any breakpoints encountered whilst tracing a block of code are ignored.

Single Step F7

In disassembled code, the target executes the instruction at the PC with the current register values and then stops. A Trap, Line-A or Line-F is treated as a single instruction and program execution halted on returning.

In source code, the target executes the instruction at the PC with the current register values and then stops when all low level assembly instructions generated by the single source instruction have been executed i.e. all instructions for a source macro instruction or C instruction which generates several assembler instructions have been executed.

If you are single stepping source instructions in the upper region of a mixed window the animated single stepping of individual assembler instructions will be displayed in the lower region of the window i.e. for the lower region instructions a Trap, Line-A or Line-F is treated as a single instruction but program execution continues until the next source instruction.

Step Over F8

In disassembled code, the target executes the instruction at the PC with the current register values and then stops. A Trap, Trap V, Line-A, Line-F, BR, JSR or DBRA is treated as a single instruction and program execution halted on returning.

In source code, the target executes the instruction at the PC with the current register values and then stops when the source file reference has changed. A Trap, Trap V, Line-A, Line-F, BSR, JSR or DBRA is treated as a single instruction and program execution is halted on returning.

Step Into Shift+F7

In disassembled code, the target executes the instruction at the PC with the current register values and then stops. A Trap, Line-A, Line-F, or subroutine is entered and program execution halted inside subroutines and branches at the first instruction. Traps and JSR's etc. are therefore *step*ped *into*.

In source code, the target executes the instruction at the PC with the current register values and then stops at each individually generated assembler instruction. Each individual assembler instruction is traced using the step into mechanism i.e. a Trap, Line-A, Line-F, or subroutine is entered and program execution halted inside. You may therefore need to press Shift+F7 several times on this source instruction before progressing to the next source instruction. In Mixed windows your progress through the instruction trace is displayed in the lower region. This enables you to debug macro and complex source instructions at a detailed level.

Unstep Ctrl+F7

Only individually single stepped instructions may be untraced i.e. it is not possible to untrace any instructions stepped over, or any Traps, Line-A's, Line-F's encountered whilst single stepping. All instructions traced using step into may be untraced. Source instructions can be unstepped only if they were single stepped and no Traps, Line-A's or Line-F's were encountered in the body of their generated code or they were executed using multiple step into requests.

7.10 Expressions

The debugger uses a similar expression evaluator as the assembler with the addition of powerful expression formatting facility. Use Ctrl+E to invoke the expression calculator shown in Figure 16 below. In contrast to the assembler, the default base is 16 and * is used for multiplication only. Note that expressions starting with a hexadecimal number must have a leading 0 to differentiate it from a register name so for example, the register `d0` is not confused with the hexadecimal value `d0`.

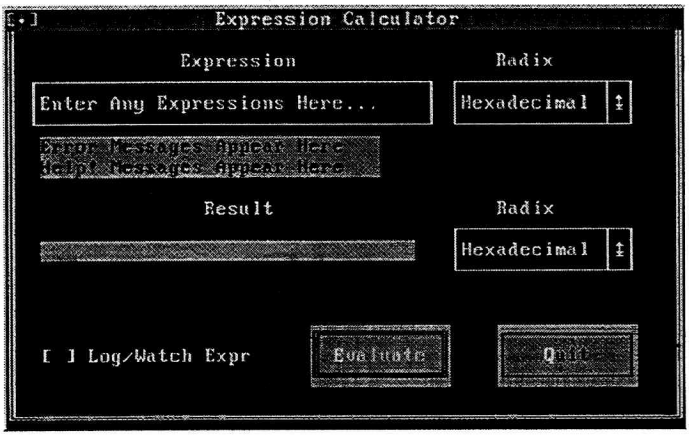


Figure 16. The expression calculator.

7.11 Expression Formatting

The debugger provides a powerful expression formatting facility for controlling the display of expressions in Log and Watch windows. To enter Formatting is controlled by the use of *formatting expressions* which work in a similar way to the C 'prinf' function, consisting of a *formatting string* followed by any number of comma separated expressions. The expressions are numbered from 0 and can be any valid debugger expression referencing register names or memory locations. The syntax for a formatting expression is:

["*FormattingString*"|*FormattingString*,] [*Expression*]...

7.11.1 The Format Specification

The formatting string consists of one or more *format specifications*. Each specification starts with a '%' symbol optionally followed by one or more modifiers and terminates with a format specifier; multiple specifications are separated by spaces or by enclosing the sequence in quotes and separating each specification with a comma. The syntax for the format specification is:

%[*Pointer*][*Width*][*Repeat*]*Specifier*

where

%	denotes the start of a format specification.
Pointer	denotes an optional modifier that repositions the parameter pointer.
Width	denotes an optional modifier that specifies the display width of the expression.
Repeat	denotes an optional modifier that specifies the number of items to display.
Specifier	controls pointers and formats affecting the display of the expression.

7.11.2 The Format Specifier Character

The format specifier character is used to control pointers and formatting instructions that affect the display of an expression. The characters and their effects are given in the following table.

Character	Effect
D, d	Decimal signed integer.
C, c	ASCII character.
U, u	Decimal unsigned integer.
O, o	Octal unsigned integer.
X, H	Hexadecimal unsigned integer using 'A'-'F'
x, h	Hexadecimal unsigned integer using 'a'-'f'
S, s	Pointer to null terminated ASCII string.
T,t	Displays the time in the form HH/MM/SS
!	Display parameter expression as a string.
I, i	Pointer to instruction to disassemble.

Table 17. Format specifier characters and their effects.

Examples of valid format expressions are:

%d	Format parameter as a decimal signed integer.
%u	Format parameter as a decimal unsigned integer.
%H	Format parameter as a hexadecimal unsigned integer (using 'A'-'F').

7.11.3 The Pointer Modifier

A parameter pointer holds the position of the current expression, the first expression starting at position 0. The optional pointer modifier repositions the parameter pointer and follows directly after the '%' symbol. The modifier consists of a decimal number, optionally preceded by a '+' or '-' symbol and terminated with a '#' symbol. The syntax is:

[+|-]*Number#*

where:

+|-	repositions the parameter pointer relative to its current position.
Number	denotes an absolute value for the parameter pointer or the size of the relative movement if used in conjunction with '+' or '-'.


Note

Setting the parameter pointer to a value before the first parameter causes the pointer to be set to the first parameter. Conversely, setting the pointer to a value beyond the last parameter invalidates the action of subsequent specifiers and they are copied verbatim into the display string.

The following example shows a formatting string that displays its three parameters as decimal signed integers and in reverse order.

```
%2#d %1#d %0#d
```

The following example shows a formatting string that displays its parameter first in hexadecimal and then in decimal.

```
"%0#x,%-1#d"
```

7.11.4 The Width Modifier

The optional width modifier specifies the field width within which the expression is to be displayed and follows the # modifier (or '%' symbol if no pointer modifier is specified). The field width is given either as a decimal number or by the value of the next parameter expression. The syntax for the width modifier is:

[-][*Number*|*]

where

- denotes that the field is left justified within the field width. If the '-' symbol is not specified the field will be right justified.

Number is a decimal number denoting the width of the field; prefixing *Number* with a zero will pad the display field with zeroes. For '%s' formats the width specifies the maximum number of characters to display.

* denotes that the field width is specified by the value of the next parameter expression.

Examples of valid uses of the width modifier are:

`%4x` Format parameter as a 4 digit right justified hexadecimal unsigned integer (using 'a'-'f').

`%-8s` Format parameter as a 8 character left justified string.

`%08X` Format parameter as a 8 digit hexadecimal unsigned integer (using 'A'-'F') and pad with zeroes.

`%3#-15S` Format the 4th parameter as a 15 character left justified string.

`%*s` Format parameter as a string according to the value of the next parameter.

`%4#*d` Format parameter as a 4 digit right justified decimal signed integer according to the value of the next parameter.

7.11.5 The Repeat Modifier

The optional repeat modifier controls the number of items displayed and follows the pointer and width modifiers (if specified). The modifier consists of a '@' symbol followed by an optional size modifier and terminated with the number of items to be displayed. The syntax is:

@[*Size*]*Number*

where:

@	denotes the start of the repeat modifier.
Size	denotes the size of items fetched from memory which can be one of the following: `b` byte `w` word `t` triple `l` long The endianess of the target processor is preserved when fetching multi-byte items.
Number	is a decimal number denoting the number of items to be displayed.

Displayed items will be comma separated if the format specifier is decimal or octal, by spaces if the specifier is hexadecimal and not spaced at all if the specifier is characters. The repeat modifier has no effect if the format specifier is a string or instruction.

8 SNMAKE

The SNMAKE utility enables the SNASM2 development system to be easily used from within a text editor. SNMAKE works on the make utility principle of reading a file supplied by the user in which are defined the relationships between the target(s) the user wishes to create and the source files from which that target is to be created. The target is said to be dependant upon its source files which are known generically as *dependencies*. The file in which these relationships are defined is known as the *project file* (sometimes called the *make file*). The project file contains rules specifying how to recreate targets. Once SNMAKE has read this file it determines which targets have dependants that have been updated since the target file was created, and therefore which targets must be recreated from their dependants.


Note

If you are familiar with project files you should note that there are a number of differences in the SNMAKE syntax compared to conventional make utilities.

8.1 Editor Macros for SNMAKE

Some of the macros used to enhance the editor environment interface the editor with SNMAKE and allow the utilty to be invoked from within the editor. The following description assumes that the macros are being used as supplied, without any reallocation of key-bindings.

Use Alt+F9 to invoke the SNASM2 main menu from within one of the supported text editors. The menu item Select Project File will display a window listing all project files in the current directory or a message if none can be found (SNMAKE project files must have a '.PRJ' extension). The first line of each file is displayed as an 'aide memoir'. SNMAKE can also be invoked from the command line with the /p switch set (command-line switches are described on page 178) in which case it will attempt to append a '.PRJ' extension to the project file name it is given. In this mode it is said to be in *project mode*. SNMAKE is always in project mode when invoked from an editor.

8.2 Project Files

8.2.1 Creating Project Files

Project files *must* contain a label beginning in the first column of the form:

```
[SnMake]
```

SNMAKE performs a case insensitive search for this label and ignores any text before it. If SNMAKE reaches the end of file before encountering the label it will generate an error and exit, resulting in an error window appearing in the editor. SNMAKE treats everything following the [SNMAKE] label as valid input until it encounters another '[' in the first column or the end of the file.

There are two other labels, [DEBUG] and [EVAL] which are significant only if the project file is selected within the editor. The next non-blank line following the [DEBUG] label contains the command-line used to invoke the debugger from within an editor. Similarly, the line following the [EVAL] label contains the command-line to invoke the expression evaluator, EVALSYM. The command-line consists of the special token '$$$' which represents the expression passed to EVALSYM by the editor and the the COFF file to get the symbols from.

Example

A simple project file might look as follows:

```
project file to assemble prog.68k to t7:
[snmake]

t7:;        prog.68k
   snasm68k $! /sdb prog.68k,t7:prog

[debug]
   snbug68k -t7:prog

[eval]
   evalsym /v$$$ prog
```

8.2.2 Defining Targets

SNMAKE regards anything starting in column 0 and terminated with a ';' as a target declaration. The following are all valid target names:

```
target1;
t5;
e:netwrkt7;
```

Note that white space in target declarations is stripped out.

8.2.3 Special Targets

SNMAKE supports a number of special targets.

.RESOURCE;

Declares a list of programs that are able to use resource files. It must be the first item declared in the project file after the [SNMAKE] label.

.INIT;

Commands following this target declaration will always be carried out first when this project file is executed. The INIT target does not need to be the first declaration in the project file.

.DONE;

Commands following this target declaration will always be carried out last when this project file is executed.

.INIT and .DONE do not have to be declared as the first and last targets within the project file, SNMAKE will recognise them and re-adjust its list of targets accordingly. .INIT and .DONE will always be executed and should not be declared with dependants.

t?: ;

The targets on the SCSI bus are recognised and will always cause the rules associated with it to be invoked. These targets should be declared with dependencies.

.SNRES

A project file that specifies multiple targets and invokes the assembler will normally cause the assembler to be invoked for each target. Use .SNRES to specify a list of commands that are able to use resource files. This means, for example, that several 68000 source files can be placed in a single file and the assembler invoked once only. This reduces the overhead associated with invoking the assembler for each file. See Example 3 below.

Example 1

This example declares that SRC1.68K should be assembled down to SCSI target t7: . This will occur every time the project file is executed. See *Special Macros* on page 176 for more information on the '$!' command.

```
t7:;                src1.68k
   snasm68k $! src1.68k,t7:
```

Example 2

Program command-lines in a project file that exceed 128 characters in length in are placed in a temporary file. Each such command is placed in an individual temporary file called '*Filename*.$$$' and each command-line argument placed on a new line in that file. SNMAKE calls the command using '@*Filename*.$$$' as shown below.

```
[SnMake]
.RESOURCE;          somecmd

tgt;                depl
   somecmd  depl ... very long command line > 128 chars
```

SNMAKE will detecting an over-long command-line and create a temporary response file and call SOMECMD as follows:

```
somecmd @tmp1.$$$
```

Example 3

```
[SnMake]
.SNRES;     snasm68k

t7:;        testmain.68k
            snasm68k $! testmain.68k,t7:

t5:;        testsub.68k
            snasm68k $! /K testsub.68k,t5:

!ifdef(debugstr)
            snbug68k
!endif

[debug]
            snbug68k
```

8.2.4 Defining Dependencies

Anything following the ';' on the same line as a target declaration is regarded as a dependency declaration with multiple dependencies separated by white space.

Example

This example declares that TARGET1.COF is dependant on SRC1.68K and SRC2.68K. SNMAKE will attempt to invoke any rules defined for TARGET1.COF if either SRC1.68K or SRC2.68K have been updated since TARGET1.COF was last created.

```
target1.cof;        src1.68k src2.68k
```

8.2.5 Defining Explicit Rules

Anything following the end of a target declaration line is considered a rule declaration. Valid rule declarations must be indented by at least one space or tab. Blank lines following a target declaration are ignored. More than one command may follow a given target, each starting on a new line.

Example

```
target1.cof;       dep1 dep2
   snasm68k  /l dep1 dep2,target1.cof
```

This example defines a rule telling SNMAKE to issue the following command if TARGET1.COF is younger than either DEP1 or DEP2.

```
snasm68k  /l dep1 dep2,target1.cof
```

8.2.6 Defining Implicit Rules

If SNMAKE cannot create a target using explicit rules it will attempt to do so using any implicit rules defined in the project file. Implicit rule declarations must begin in the first column with a '.' character and comprise two parts. The first part is the file extension of the target for which the rule will apply. SNMAKE searches the list of implicit rules it has defined looking for a rule that matches the extension of the target it is trying to make. Thus to define an implicit rule that SNMAKE will use to deal with any targets with a '.COF' extension the first part of the declaration is:

```
.cof
```

Having found this, SNMAKE attempts to match the second part of the declaration. This informs SNMAKE that any targets with a '.COF' extension are to be created from a dependency of the same name but with a '.658' extension as follows:

```
.cof,.68k
```

The second part of the declaration must begin with a '.' character, unless a path name is specified as below:

```
.cof,e:temp\.68k
```

This tells SNMAKE that any targets with a '.COF' extension are to be created from a dependency of the same name but with a '.68K' extension in directory E:\TEMP.

8.2.7 Defining Rules for Implicit Targets

The rule following an implicit target definition must be indented by at least one space or tab. Two macros exist to aid specifying implicit rules, $+ and $-. $+ specifies the target and $- its dependency. Thus the following implicit rule

```
.cof,.68k
   snasm68k $+,$-
```

when invoked on target PROG1.COF will result in the following command:

```
   snasm68k  prog1.68k,prog1.cof
```

Note that explicit rules will always be used in preference to implicit rules if explicit rules have been set for a given target. In addition, if more than one set of implicit rules are defined for the same target group, the implicit rule most recently defined (in terms of position within the project file) will be invoked on any suitable targets so that:

```
.cof,.68k
```

will create any suitable targets in this area from files of the same name with a '.68K' extension.

8.2.8 Line Continuation

Use the '\' character followed *immediately* by a carriage return to continue a line without introducing a newline character. The following example declares DEP1 to DEP11 as dependencies to target 7. Without the continuation mark SNMAKE would truncate the dependency list at dep9.

```
target7; dep1 dep2 dep3 dep4 dep5 dep6 dep7 dep8 dep9 \
            dep10 dep11
```

8.2.9 Comments

Comment lines begin with a '#' character and can start at any position on the line.

8.2.10 Macros

Macros can be passed into SNMAKE from the command line using the e switch (see *Command-line Syntax* on page 178 for more information on switches). Macros are defined within the project file using the following syntax:

`macro1=`*MacroName*

Macro definitions must begin in the first column of the line. White space is stripped out of macro definitions. Defined macros are referenced using the following syntax :

`$(macroname)`

Given the above macro definition $(macro1) will expand to `MacroName`. '$' signs can be protected from attempted macro expansion by the addition of the macro syntax breaker '$'. Thus $$20000 is passed though SNMAKE as $20000.

The following macro functions allow the user to manipulate defined macros.

Function	Description
`$e(`*MacroName*`)`	Expands to the extension of *MacroName*.
`$n(`*MacroName*`)`	Expands to only the filename of the macro definition.
`$p(`*MacroName*`)`	Expands to the pathname of the macro definition.
`$d(`*MacroName*`)`	Expands to the drive name of the macro definition.
`$b(`*MacroName*`)`	Expands to the filename in the macro definition.

Table 18. SNMAKE macro functions.

Example

```
macroname=test.obj
$e(macroname)
# $e(macroname) expands to '.obj'

macroname=e:test\test.obj
$n(macroname)
# $e(macroname) expands to 'test.obj'

macroname=e:test\temp\test.obj
$p(macroname)
# $p(macroname)   expands to 'e:test\temp\'

macroname=e:test\temp\test.obj
$d(macroname)
# $d(macroname)   expands to 'e:'

macroname=e:test\temp\prog1.obj
$b(macroname)
# $b(macroname) expands to 'prog1'
```

8.2.11 Special Macros

SNMAKE provides a special macro to set the `i` and `d` assembler command-line switches from within the project file.

The `i` switch creates an output window to which output is sent whilst running, enabling progress to be monitored from within the editor. This option is always set by the supplied macros.

The `d` switch puts the assembler into debug mode, i.e. the code is assembled but not run. This allows the debugger to be entered before the code is executed. This option can be controlled from the SNASM2 menu using the Set Debug Mode menu item.

Control of these switches from within an editor is possible only if the $! macro is present on the SNASM658 command lines in the project file as shown below:

```
targ1;     dep1 dep2
   snasm68k $!  /l dep1 dep2,targ1
```

Assuming that debug mode is set to 'ON' (using the Set Debug Mode option from within the editor) the above command will expand to

```
   snasm68k /i /d /l dep1 dep2 ,targ1
```

In addition the d and i switches set up two macros, DEBUGSTR and INFOSTR, which can be tested with the !IFDEF command as described below.

8.2.12 Conditionals

A conditional capability is provided within SNMAKE by the !IFDEF... !ELSE... !ENDIF construct, providing the ability to test for macro definitions.

Example 1

If the special macro DEBUGSTR is set i.e. debug mode is on, the debugger will be invoked every time SNMAKE is invoked with this project file. The !ENDIF command is required to terminate the !IFDEF call. SNMAKE will generate an error if it reaches the end of the project file with an unbalanced number of calls to !IFDEF and !ENDIF.

```
!ifdef(debugstr)
SRC_DB=/sdb
!else
SRC_DB=""
!endif

!ifdef(debugstr)
t7:;        prog1.68k
   snasm68k $! /sdb prog1.68k,t7:prog1
   snbug68k -t7:prog1
!else
t7:;        prog1.68k
   snasm68k $! prog1.68k,t7:
!endif
```

Example 2

This is a more efficient implementation of the previous example.

```
t7:;        prog1.68k
   snasm68k $! $(SRC_DB) prog1.68k,t7:prog1
!ifdef(debugstr)
   snbug68k -t7:prog1
!endif
```

8.3 Command-line Syntax

SNMAKE can be invoked from the command-line using the following syntax:

```
snmake [Switches] [ProjectFile] [ErrorFile]
```

Invoking SNMAKE with no arguments causes it to look for a project file called 'MAKEFILE' and process that. The optional *ProjectFile* parameter specifies an alternative project file name. If *ErrorFile* is specified all error information will be output to that file.

8.3.1 Switches

SNMAKE accepts five switches from the command-line.

Switch	Description
q	*Quiet mode.* No echoing is done as SNMAKE proceeds.
p	*Project mode.* This forces SNMAKE to treat make files as project files i.e. as if invoked from within an editor. If no make file name is specified SNMAKE will default to MAKEFILE.PRJ. In project mode all output from SNMAKE goes to a file called SNMK.ERR, any error output from the programs invoked by SNMAKE will appear on-sceen unless an error file is specified.
d	*Set debug mode.* Sets the special macro $!. Only of use if invoking SNASM2 from the project file.
i	*Set info mode.* As above.
e *Name=Exp*	*Pass a macro definition into SNMAKE.* Sets up a macro *Name* which will expand to *Exp*.
b	*Build all.* All rules carried out regardless.

Table 19. SNMAKE command-line switches.

8.3.2 Example

The following example produces a file called TEST1.COF from the source files E:SRC1.68K E:SRC2.68K . *Text in this style is comment text added to aid the reader and is not part of the SNMAKE syntax.*

```
#file to create test1.cof
```
This text will appear in the project file select menu

```
[SnMake]
```
SNMAKE in project mode starts reading at this label
```
src1.cof;  e:\src1.68k
   snasm68k $! /l /sdb e:\src1.68k,src1.cof

src2.cof;  e:\src2.68k
   snasm68k $! /l /sdb e:\src2.68k,src2.cof


test1.cof; src1.cof src2.cof
   snasm68k $! src1.cof+src2.cof,test1.cof
!ifdef(debugstr)
   snbug68k -t7b:test1.cof
!endif

[Debug]
```
SNMAKE stops reading here.
```
   snbug68k -t7b:test1.cof
```
Debugger is invoked using this string

```
[Eval]
   evalsym /v$$$ test1.cof
```
Expression evaluator is invoked with this string

This is the only information this page contains.

9 SNTEST

See also the SCSILINK utility on page 201

The SNTEST utility is used to test the integrity of the SNASM2 SCSI link and the memory in the target interface.

This section first describes the memory test and then shows you how to invoke SNTEST from the command line, including all the command line switches. SNTEST can also be invoked using a command file enabling you to perform more complex memory checks than can be achieved from the command line. The command file and its associated commands are described in the second part of this section, complete with examples.

9.1 The Memory Test

The SNTEST utility checks the integrity of memory by performing a sequence of read/write operations using randomly generated data. SNTEST first generates a buffer of random numbers. Then for each specified memory range SNTEST generates a random starting point in the buffer and downloads sufficient random numbers to fill the memory range. When all the memory ranges have been filled the contents of the memory ranges are read back from the target and compared to the buffer contents. If there are no differences then the memory is deemed to be functioning correctly.

A variant on the above test can be performed using the RANDOM command. In this case the memory range is not filled but a 1, 2, 3 or 4 byte stream of random numbers is placed at a random position in each specified memory range. To provide a comprehensive check use both forms of the test.

9.2 Syntax

`sntest` *Switches*

9.2.1 Switches Summary Table

Switch	Description

`c`	Check the functioning of the SNASM2 PC card. This will print out a report similar to the following:
	`PC card address is : 0x310`
	`DMA Channel : 1`
	`Initiator ID : 6`

`t`*n*	Test link to target number *n*. Tests to see if a connection can be made with SCSI target *n*.
`l`	Test links to all connected targets. SNTEST will attempt to connect to all SCSI targets from 0-7. It will then print out the name of the target interface for each connected target. The '/l' and '/i' switches can be used together to print out a list of all the commands supported by a target.
`m` *CmdFile*	Perform memory test on a target interface using memory areas specified in the command file *CmdFile*.
`i`	Display list of supported commands. A list of the SCSI commands supported by the target interface will be displayed on screen.
`r`*n*	Repeat memory test *n* times. This switch will cause SNTEST to test memory areas specified in the command file. Use the Esc key to abort the test.
`k`	Continuous memory check. This switch will cause SNTEST to continuously test memory areas specified in a command file. Use the Esc key to abort the memory check.

Table 20. SNTEST command line switches.

Example 1

To test the SCSI link, invoke SNTEST as follows:

```
sntest      /t7 /i /c
```

This tests to see if a connection can be made with target 7 and, if the connection can be made, displays a list of SCSI commands supported by that target. Lastly, the functioning of the PC card is checked.

Example 2

To perform a memory check using the command file COMMAND.DAT, invoke SNTEST as follows.

```
sntest /m command.dat
```

9.3 The SNTEST Command File

The command file is used to specify more complex integrity checks than can be achieved from the command line. SNTEST provides a number of commands that enable you to perform any combination of tests. These commands are identified to SNTEST by prefixing them with a '!' symbol. All numbers are in hexadecimal with those starting with 'A' - 'F' taking a '$' prefix..

9.3.1 The # Command

The # symbol denotes the beginning of a comment line.

Syntax

 # Comment

9.3.2 The MEMBLK Command

The MEMBLK command specifies the memory areas to test on the target machine.

Syntax

`!memblk` *Start,Length,Step,Count*

Start	The start address for the memory check.
Length	The length of the memory block to check.
Step	The length of memory to step over to reach the next area to check.
Count	The number of blocks to check.

Example 1

```
!memblk 8000,8000,8000,20
```

This starts memory checking at address $8000 and continues for a length of $8000. The memory range starting from the end address of the tested block and continuing for a length of $8000 is then stepped over. Memory checking is continued in this manner until $20 checks have been done.

Example 2

```
!memblk 0,100000,0,1
```

This checks memory from $0 to $100000 in one go as a contiguous block.

9.3.3 The RANDOM Command

The RANDOM command performs a random memory check.

Syntax

`!random` *Start,Length,Step,Count*

Start	The start address for the memory check.
Length	The length of the memory block to check.
Step	The length of memory to step over to reach the next area to check.
Count	The number of blocks to check.

Example 3

```
!random 8000,8000,8000,20
```

This performs random memory checks consisting of 1, 2, 3 or 4 byte writes over the memory range starting at address $8000 and continuing for a length of $8000. The next $8000 bytes of memory are stepped over and testing continued in this manner until $20 tests have been performed.

9.3.4 The TARGET Command

The TARGET command sets the SCSI target ID from within the data file.

Syntax

`!target` *n*

n The target ID number.

This is the only information this page contains.

10 SNGRAB

Transfer binary data and files with SNGRAB

The SNGRAB utility enables you to transfer binary data and files between the assembler and a target. SNGRAB will accept binary data on the PC in the form of a binary file and will upload from the target to the PC into a binary file. COFF files can also be downloaded from the PC onto a target machine.

This section first shows you how to invoke SNGRAB from the command line and describes all the command line switches. SNGRAB can also be invoked using a command file enabling you to perform more complex uploading and downloading configurations than can be achieved from the command line. The command file and its associated commands are described in the second part of this section, complete with examples.

10.1 Command Line Syntax

`sngrab` [*FileName*] [[-|/]*Switches*]

If SNGRAB is invoked with a file name only, it assumes that this is a binary or COFF file that you want to download, determining by inspection which type of file it has been given. If SNGRAB is downloading a binary file it will begin at address 0 on the target and download the entire length of the file contiguously. COFF files are downloaded as specified within the files themselves.

Example

```
sngrab      tank.cof -r
```

This invokes SNGRAB with the COFF file TANK.COF. SNGRAB determines the file type and downloads the file to the current target. Once the download is complete, SNGRAB sets the target running.

10.1.1 Command Line Switches

Switch	Description
d *CmdFile*	Specifies *CmdFile* as a command file. A command file enables you to perform more complex downloads and uploads than can be achieved from the command line.
h	Interpret all numbers in the command file as hexadecimal. Numbers are otherwise interpreted as decimal unless prefixed with a '$' character
o *FileName*	Overrides the first filename in a command file with *FileName*. This enables you to use a single command file to download multiple different files from the command line.
p *FileName*	Forces a download of the file *FileName* as a binary image. This switch can be used if the binary image to be downloaded contains data that would otherwise fool SNGRAB into trying to download the binary as a COFF file.
r	Send a `run` command to the target after download. This is analogous to `!run` in a command file
t*n*	Connect to SCSI target *n*.

Table 21. SNGRAB command-line switches.

10.2 The Command File

The command file is used to specify more complex downloads and uploads than can be achieved from the command line. SNGRAB provides a number of commands that enable you to perform any combination of uploads or downloads.

These commands are identified to SNBRAB by prefixing them with a '!' symbol, except for the TO and FOR commands which are not prefixed. The command file can also include comment lines which begin with the '#' symbol. Numbers can be decimal or hexadecimal if prefixed with a '$' symbol.

10.2.1

The # symbol denotes the beginning of a comment line.

Syntax

\# *Comment*

10.2.2 DLBLOCK

The DLBLOCK command provides SNGRAB with information about the areas of memory to download.

Syntax

`!dlblock` *Start,Length,Step,Count*

Where:

Start	is the start address for the upload or download.
Length	is the length of the memory block to upload or download.
Step	is the length of memory to step over after each upload or download.
Count	is the number of uploads or downloads to perform.

10.2.3 DLCOFF

The DLCOFF command tells SNGRAB to download a COFF file.

Syntax

```
!dlcoff    FileName
```

Where:

FileName is the name of the COFF file to download.

10.2.4 DOWNLOAD

The DOWNLOAD command sets SNGRAB to download mode, the default setting. When switching from upload to download mode you must specify a filename using !NAME or an error will be generated.

Syntax

```
!download
```

10.2.5 FILL

The FILL command fills subsequent target memory with the character specified using the FILLCHAR command. The default is $FF.

Syntax

```
!fill
```

10.2.6 FILLBLOCK

The FILLBLOCK command provides SNGRAB with information about the areas of memory to upload or download.

10.2.9 NAME

The NAME command specifies the filename to upload or download.

Syntax

`!name` *Filename*

Where:

Filename is the name of the file to upload or download.

10.2.10 RUN

The RUN command sets the current target running.

Syntax

`!run`

10.2.11 TARGET

The TARGET command sets the SCSI target number.

Syntax

`!target` *n*

Where:

n is the SCSI target number.

10.2.12 TO

The TO sets a target memory range prior to uploading or downloading.

Syntax

Start `to` *End*

Where:

Start	is the start address in target memory.
End	is the end address in target memory.

10.2.13 ULBLOCK

The ULBLOCK command provides SNGRAB with information about the areas of memory to upload.

Syntax

`!ulblock` *StartAddress,Length,Step,Count*

Where:

StartAddress	is the start address for the upload or download.
Length	is the length of the memory block to upload or download.
Step	is the length of memory to step over after each upload or download.
Count	is the number of uploads or downloads to perform.

10.2.14 UPLOAD

The UPLOAD command sets SNGRAB to upload mode. When switching from download to upload mode you must specify a filename using `!name` or an error will be generated.

Syntax

`!upload`

10.3 Examples

Example 1

```
!dlblock $8000,$8000,$8000,30
```

Downloading will start at target address $8000 for a length of $8000 bytes, it will then step over a range of $8000 and download $8000 bytes at that address. This process is repeated until 30 downloads have been completed.

Example 2

In the following example UPLOAD will generate an error because a new filename must be specified before switching from download to upload mode.

```
!download
!name afile.bin
0 for $8000

!upload
0 for $8000
```

The correct procedure is:

```
!download
!name afile.bin
0 for $8000

!name file1.bin
!upload
0 for $8000
```

Example 3

A sample SNGRAB command file is given below.

```
# Target 5
!target=5 !name=tank.bin 0 to $80000 !run

# Target 7
!target=7
!fillchar $FA
!fillblock $8000,$8000,$8000,4
!download
!name sfxdemo.bin
!dlblock $8000,$8000,$8000,$2
!upload
!name sf.bin
$8000 to $8200
$8400 for $200
!run
```

The first line is a comment indicating that the following lines refer to target 5. The next line sets the SCSI target to 5, the name of the download file to TANK.BIN, the download memory range from $0 to $80000 and sets the target running. The next line is a comment indicating that subsequent lines refer to target 7. The following line sets the SCSI target to 7, !FILLCHAR $FA sets the memory filling character to $FA. The next line instructs SNGRAB to fill the target memory with $FA starting at address $18000 for a length of $8000, skipping a range of $8000, filling at that address for a range of $8000 and continue until four fills have been completed. The next three lines set the name of the file to download to SFXDEMO.BIN, tell SNGRAB that download mode is on, and to download SFXDEMO.BIN onto the target starting at address $8000 for a length of $8000, skipping a range of $8000 and doing another download at $18000. The next three lines tell SNGRAB that an upload is required, that the name of the file to upload into is SF.BIN, and the memory ranges to use are $8000 to $8200 and $8400 to $8600. The last line runs the code in target 7.

This is the only information this page contains.

11 SNLIB

SNLIB is a utility program for creating and maintaining object module libraries ('libraries'). A library is a file containing several object modules. These libraries can be searched by the assembling linker if it cannot find a symbol in the object files. If the assembling linker finds that the external symbol it needs is defined in a library module then the module will be extracted and linked with the object modules.

11.1 Running SNLIB

11.1.1 Command-line Syntax

`SNLIB` [-|/]*Switches Libraryfile Modules*

Switch	Description
a	*Add.* Add modules to library
d	*Delete.* Deletes modules from library.
l	*List.* Lists modules in library.
u	*Update.* Updates modules in library.
x	*Extract.* Extracts modules from library.

Table 22. SNLIB command-line switches.

This is the only information this page contains.

12 SNBUTTON

The SNBUTTON utility reads and if necessary updates the information contained in the button(s) on the PC Card.

12.1 Command-line Syntax

`snbutton [[-|/]u`*DataFile*`]`

Switch	Description
u *DataFile*	*Update button.* Update the information in the button optionally using the data contained in the file *DataFile*.

Table 23. SNBUTTON command-line switches.

12.2 Reading the Button Serial Number

To read the button serial number invoke SNBUTTON which will then display a screen similar to that shown in Figure 17 below.

Figure 17. Button information displayed by SNBUTTON.

Serial # shows the serial number required to obtain technical support. Location shows the physical location of the button on the system, either SNASM Card or an LPT port. The Subkey 0 ID will be SN2 DOS for a valid button but Subkey 1 and Subkey 2 may not be set.

Press Esc to exit SNBUTTON.

12.3 Updating the Button

Update the button by invoking SNBUTTON as above, optionally using the update switch (u) to use a data file. Press any key in the button

information window to display the data entry screen shown in Figure 18 below. The data is shown as a hexadecimal representation of four seventeen byte strings. If SNBUTTON was invoked using a data file its contents will be displayed, otherwise the strings will be blank and ready to accept faxed or telephoned.

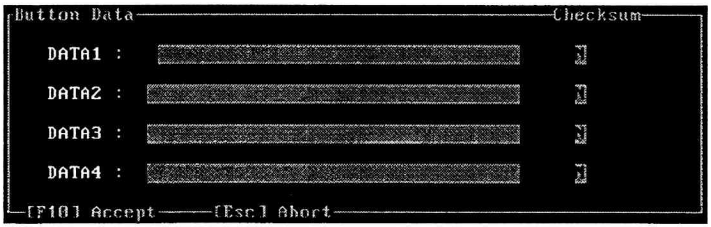


Figure 18. The data entry screen for updating the button.

If the checksum fields in this screen are not all showing 'Y' at the end of data input then the supplied data is incorrect or has been corrupted during transmission. Once these fields are correct press F10 to update the button. SNBUTTON will report a successful update or display an error message if the update data is an earlier version than that already in the button or if the data does not checksum correctly.

13 SCSILINK

13.1 Running SCSILINK

SCSILINK must be resident before any of the software can be used. Typically, the install program adds a line to AUTOEXEC.BAT to invoke SCSILINK automatically but should this not be the case, or for any other reason, SCSILINK can be invoked either from the command-line or from any batch file.

13.1.1 Command-line Syntax

`SCSILINK` *CardAddress* [,S?][,R][,Q][–]

CardAddress	The address of the PC card, set using links 0-7 on the card.
S?	*Ignore Switches*. This ignores the switches and uses ? as the SCSI device number of the PC card.
R	*Reinstall Driver*. This loads a copy of SCSILINK into memory, even if other copies are already installed.
Q	*Quiet Mode*. This suppresses installation information.
–	*Remove*. This removes the most recently installed copy of SCSILINK from memory, other copies will stay resident.

Example

If you have left the PC Card as supplied just insert:

```
SCSILINK 390
```

13.2 Interface to SCSILINK

The SCSILINK software is accessed using int 7dh. AH is used to select the required function as shown below.

AH=0 Reset SCSI Bus

Resets all devices on the SCSI bus.

In	-
Out	-
Errors	-

AH=1 Connect to Target

Arbitrates for use of bus and selects target.

In	AL=Target ID
Out	-
Errors	CF set if error, AL=Initiator error, AH=Target error.

AH=2 Send command

Sends the command block to the target and performs and related I/O.

In	ES:BX = pointer to parameter block shown below
Out	-
Errors	CF set if error, AL=Initiator error, AH=Target error

Parameter block: (All with 8086 byte ordering)

Size of command block	Dword
Offset of command block	Word
Segment of command block	Word
Size of buffer	Dword
Offset of buffer	Word
Segment of buffer	Word

The contents of the command block which are described later have most significant bytes and words first!

AH=3 Set TimeOut

Change one of SCSILink's internal timeouts to allow communication with very slow targets.

In	AL=TimeOut Number BX=New value (55ms ticks)
Out	-
Errors	-

Number	Function	Default
0	Time to wait for bus	18
1	Time to wait for new phase	5
2	Max time to send/receive block	18
3	Time to wait for reselect	180

AH=4 Get Error String

Get text to print when an error is reported.

In	AL=Error number
Out	ES:BX=pointer to zero terminated string
Errors	-

AH=5 Get Address

Get information about current hardware configuration.

In	-
Out	AL=Dma Channel, AH=Initiator ID, BX=Card Addr
Errors	-

AH=6 Put Data

Save data for later retrieval. (Stored in PSP of SCSILink)

In	CX=Byte count (1-256), ES:BX=Data to save
Out	-
Errors	CF set if error, AL=1 Too much data

AH=7 Get Data

Retrieve data stored using Function 6

In	CX=Byte count (1-256), ES:BX=Buffer to fill
Out	-
Errors	CF set if error, AL=1 Too much data

AH=8 Terminate Session

If a program has made any use of these link commands it should issue a Terminate Session command as it exits even if it is exiting due to a SCSI error. This command exists so that any software that intercepts int 7Dh knows when other programs have finished with the link.

AH=9 Poll Request

Whenever a program performs a GETREGS command and the exception that has occured is not one that it specifically handles it should perform a Poll Request call to see if any driver that has chained onto int 7Dh handles it. This service exists so that things such as resident disk servers can continue to run even whilst debuggers and profilers are running. The resident driver assumes that a connect has been performed and tries to leave the target connected.

In	AL=Target ID
Out	AL=-1 if a resident driver handled event AL=-2 if a resident driver experienced a SCSI error
Errors	-

13.3 The SCSILINK Command Protocol

The following is a list of the commands currently defined. Not all of these commands are currently supported - the SendImpl command lets you check what commands the software at the other end supports. Some of these commands will only be handled by downloaders which leave the machines OS resident, and these downloaders usually support some extra commands to support file handling and multi-tasking.

If the top bit of a command byte is set the target will disconnect after executing the command and reporting the status. This means the software can support up to128 commands.

13.3.1 Command Format

	Byte	Function
2 Byte Commands	0	Command byte.
	1	Modifier.*
10 Byte Commands	0	Command byte.
	1	Modifier.*
	2-5	Length field.
	6-9	Address field.

* Set to zero if unused

Table 24. The SCSILINK command format.

13.3.2 Command Summary

	Commands	Bytes	Data	Action
0	Noop	2	-	Do nothing except report status
1	SendImpl	2	In	Send binary array of valid commands
2	SendID	2	In	Send machine/processor ID
3	SendAddr	2	In	Send start and length of link workspace
4	RecvAddr	10	-	Change address of link workspace
5	FindRam	10	In	Find how much memory is free
6	ReserveRam	10	In	Ask OS for some memory
7	FreeRam	10	-	Hand memory back to OS
8	ReBoot	2	-	Reboot target
9	MakeSafe	2	-	Prepare for Re-send
10	Return	2	-	Return to OS
32	SendMem	10	In	Send a block of memory
33	RecvMem	10	Out	Receive a block of memory
34	VerifyMem	10	Out	Verify a block of memory
35	CheckSum	10	In	Generate checksum of a block of memory
40	SendRegs	2	In	Send register block
41	RecvRegs	2	Out	Receive register block
48	GoPC	2	-	Invoke code at PC without return addr
68	GetOSErr	2	In	Get result of last OS function

Table 25. SCSILINK command summary.

13.3.3 SCSILINK Commands

Noop

Command	0
Modifier	0
Data Transfered	None
Function	Does nothing. However this can be used to disconnect the target by setting the top bit in the usual way.

SendImpl

Command	1
Modifier	0
Data Transfered	Binary Array of Commands
Offset	0
Length	16
Function	Target sends 16 byte binary array indicating which of the 128 commands it supports. The data is bigendian so the first byte is for commands 127-120.

SendID

Command	2	
Modifier	0	
Data Transfered	Processor ID	Machine ID
Offset	0	12
Length	12	12
Function	Target sends string giving processor and machine ID.	

SendAddr

Command	3	
Modifier	0	
Data Transfered	Start of Link Memory	Length of Link Memory
Offset	0	4
Length	4	4
Function	The link software reports the address and length of the block of memory it is using. The length will be zero if the link software doesn't use any RAM.	

RecvAddr

Command	4
Modifier	-
Length	-
Address	New address for link memory
Data Transfered	New Address
Offset	0
Length	4
Function	Causes link software to relocate itself.

FindRam

Command	5
Modifier	0
Length	-
Address	Type of RAM to reserve (Target specific).
Data Transfered	Ram Free
Offset	0
Length	4
Function	Asks the targets OS the size of the largest block of contiguous memory that it can allocate.

ReserveRam

Command	6	
Modifier	0	
Length	Amount of RAM to reserve	
Address	Type of RAM to reserve (Target specific)	
Data Transfered	Memory Handle	Address of RAM
Offset	0	4
Length	4	4

Function Target machine asks OS for ram and passes back a handle which will be required to later free the memory and the base address of the block of memory.

FreeRam

Command	7
Modifier	0
Length	-
Address	Memory handle.

Function Hand memory back to OS.

ReBoot

Command	8
Modifier	0

Function Reboot target machine and go through normal boot process.

MakeSafe

Command	9
Modifier	0
Function	Target machines puts the hardware into the safest configuration possible. This usually involves disabling all DMA and interrupts, going out of polled mode and freeing any reserved RAM.

Return

Command	10
Modifier	0
Function	Target returns to OS.

SendMem

Command	32
Modifier	0
Length	Size of block in bytes
Address	Address of block in target
Data Transfered	Stream of Bytes
Offset	0
Length	Length specified in command
Function	Target sends a block of RAM to initiator.

RecvMem

Command	33
Modifier	0
Length	Size of block in bytes.
Address	Address of block in target
Data Transfered	Stream of Bytes
Offset	0
Length	Length specified in command
Function	Target receives a block of RAM from initiator..

VerifyMem

Command	34
Modifier	0
Length	Size of block in bytes
Address	Address of block in target
Data Transfered	Stream of Bytes
Offset	0
Length	Length specified in command
Function	Target receives a block of RAM from initiator but just checks it against current RAM contents.

CheckSum

Command	35
Modifier	0
Length	Size of block
Address	Address of block in target
Data Transfered	Additive Checksum of Bytes
Offset	0
Length	4
Function	Target adds together all the bytes in the defined range and returns the long word result.

SendRegs

Command	40
Modifier	0
Data Transfered	Register Block
Offset	0
Length	82
Function	The target sends its copy of the 68000 registers to the initiator.

Register block format is :

```
regd0       ds.l  1
regd1       ds.l  1
regd2       ds.l  1
regd3       ds.l  1
regd4       ds.l  1
regd5       ds.l  1
regd6       ds.l  1
regd7       ds.l  1
rega0       ds.l  1
rega1       ds.l  1
rega2       ds.l  1
rega3       ds.l  1
rega4       ds.l  1
rega5       ds.l  1
rega6       ds.l  1
regssp      ds.l  1
regusp      ds.l  1
regpc       ds.l  1
regsr       ds.w  1
regextype   ds.b  1   ; Exception type (-1=Startup, 1=Running)
regfunccode ds.b  1   ; Exception function code
regerroraddr ds.l 1   ; Address that caused error
reginst     ds.w  1   ; Instruction that caused error
```

RecvRegs

Command	41
Modifier	0
Data Transfered	Register Block
Offset	0
Length	82
Function	Initiator sends new register block to target.

GoPC

Command	48
Modifier	0
Function	Target restores all register from register block and jumps to PC.

GetOSErr

Command	68
Modifier	0
Data Transfered	OS Error Code
Offset	0
Length	4
Function	Returns the error code of the OS function last used by the link software.

This is the only information this page contains.

14 DOS Extender

The assembler requires a DOS extender to run which is transparent in use but may need to be configured for maximum performance. The SNASM68K development system is supplied with the WATCOM version of the 32-bit Rational Systems DOS/4GW DOS Extender. If the Rational Systems version is available, DOS/4G, it is recommended that you use this instead. The SNASM2 software will automatically search for DOS/4G provided it is in your PATH statement. The information provided in this section, adapted from the WATCOM C/386 Compiler documentation, refers to DOS/4GW but can largely be applied to DOS/4G.

14.1 Configuring the DOS Extender

14.1.1 Changing the Switch Mode Setting

In most cases, DOS/4GW programs automatically choose an appropriate real-to-protected mode switch technique. If this does not work however, the DOS16M DOS environment variable should be used to override the default setting. Change the switch mode settings by issuing the following command at the DOS prompt or include it in your AUTOEXC.BAT file.

```
set DOS16M=Value
```

The table below lists the machines and the settings to use with them. Settings that must be set are marked *required*, settings you can use are marked *option*, and settings that will automatically be set are marked *auto*.

Status	Machine	Setting	Alternate Name
Auto	386/486 w/DPMI	0	None
Required	NEC 98-series	1	9801
Auto	PS/2	2	None
Auto	386/486	3	386, 80386
Auto	386	INBOARD	None
Required	Fujitsu FMR-70	5	None
Auto	386/486 w/VCPI	11	None
Required	Hitachi B32	14	None
Required	OKI if800	15	None
Option	IBM PS/55	16	None

Table 26. The DOS extender switch modes.

14.1.2 Examples

```
set DOS16M=1    Sets the environment variable for the NEC 98-series
set DOS16M=5    Sets the environment variable for the Fujitsu FMR-70
set DOS16M=14   Sets the environment variable for the Hitachi B32
set DOS16M=15   Sets the environment variable for the OKI if800
```

14.1.3 Testing the Switch Mode Setting

After you have set the switch mode setting you can test that it is working by running the PMINFO utility. The PMINFO utility reports on the protected-mode resources available to your programs. If the utility runs, then the setting is usable on your machine. If you changed the switch setting, add the new setting to your AUTOEXEC.BAT file. If PMINFO does not run, use the RMINFO utility. This supplies configuration information to help you determine why DOS/4GW won't run on a particular machine.

14.1 Controlling Memory Usage

14.1.1 Specifying a Range of Extended Memory

Normally you don't need top specify a range of memory with the DOS16M variable. You must use this variable, however, in the following cases:

- You are running on a Fujitsu FMR series, NEC 98-series, OKI if800 series or Hitachi B-series machine.
- You have older programs that use extended memory but don't follow one of the standard disciplines.
- You want to shell out of DOS/4GW to use another program that requires extended memory.

If none of these conditions apply to you, you can skip this section.

The general syntax is:

```
set DOS16M= [SwitchMode] [@StartAddress[- EndAddress]]
            [:Size]
```

In the syntax shown above, *StartAddress*, *EndAddress* and *Size* represent numbers expressed in decimal or, if prefixed with 0x, in hexadecimal. The number may end with a K to indicate an address or size in kilobytes, or an M to indicate megabytes. If no suffix is given, the address or size is assumed to be in kilobytes. If both a size and a range are specified, the more restrictive interpretation is used.

The most flexible strategy is to specify only a size. However, if you are running with other software that does not follow a convention for indicating its use of extended memory, and these other programs start before DOS/4GW, you will need to calculate the range of memory used by the other programs and specify a range for DOS/4GW programs to use.

DOS/4GW ignores specifications (or parts of specifications) that conflict with other information about extended memory use. Below are some examples of memory usage control:

Example

`set DOS16M=1 @2m-4m`	Mode 1, for NEC 98-series machines, and use extended memory between 2 and 4MB.
`set DOS16M=:1M`	Use the last full megabyte of extended memory or as much as available limited to 1MB.
`set DOS16M=@2m`	Use any extended memory available above 1MB.
`set DOS16M=@0-5m`	Use any available extended memory from 0 (really 1) to 5MB.
`set DOS16M=0`	Use no extended memory.

As a default condition DOS/4GW applications take all extended memory that is not otherwise in use. Multiple DOS/4GW programs that execute simultaneously will share the reserved range of extended memory. Any non-DOS/4GW programs started while DOS/4GW programs are executing will find that extended memory above the start of the DOS/4GW range is unavailable, so they may not be able to run. This is very safe. There will be a conflict only if the program does not check the BIOS configuration call (Interrupt 15H function 88H, get extended memory size).

The default memory allocation strategy is to use extended memory if available, and overflow into DOS (low) memory. If you require that your memory allocations be confined to your specified extended memory range, use the DOS/4GW function `D16MemStrategy(MForceExt)` in your program.

In a VCPI or DPMI environment, the *start_address* and *end_address* arguments are not meaningful. DOS/4GW memory under these protocols is not allocated according to specific address because VCPI and DPMI automatically prevent address conflicts between extended memory programs. You can specify a *size* for memory managed by VCPI or DPMI, but DOS/4GW will not necessarily allocate this memory from the highest available extended memory address, as it does for memory managed under other protocols.

14.1.2 Using Extra Memory

Some machines contain extra non-extended, non-conventional memory just below 16MB. When DOS/4GW runs on a Compaq 386, it automatically uses this memory because the memory is allocated

according to a certain protocol, which DOS/4GW follows. Other machines have no protocol for allocating this memory. To use the extra memory that may exist on these machines, set DOS16M with the + option.

```
set DOS16M=+
```

Setting the + option causes DOS/4GW to search for memory in the range from FA0000 to FFFFFF and determine whether the memory is usable. DOS/4GW does this by writing into the extra memory and reading what it has written. In some cases, this memory is mapped for DOS or BIOS usage, or for other system uses. If DOS/4GW finds extra memory that is mapped this way, and is not marked read-only, it will write into that memory. This will cause a crash, but won't have any other effect on your system.

14.2 Setting Runtime Options

The DOS16M environment variable sets certain runtime options. To set the environment variable the syntax is:

```
set DOS16M= [SwitchModeSetting]^Options
```


Note

Some command line editing TSRs, such as CED, use the caret (^) as a delimiter. If you want to set DOS16M using the syntax above while one of these TSRs is resident, modify the TSR to use a different delimiter.

The options are:

0x01 — *Check A20 line.* This option forces DOS/4GW to wait until the A20 line is enabled before switching to protected mode. When DOS/4GW switches to real mode, this option suspends your program's execution until the A20 line is disabled, unless an XMS manager (such as HIMEM.SYS) is active. If an XMS manager is running, your program's execution is suspended until the A20 line is restored to the state it had when the CPU was last in real mode. Specify this option if you have a machine that runs DOS/4GW but is not truly AT-compatible. For more information on the A20 line, see the section, "Controlling Address Line A20" in this appendix.

0x02 — *Prevent initialisation of VCPI.* By default, DOS/4GW searches for a VCPI sever and, if one is present, forces it on. This option should be enabled.

0x04 — *Directly pass down keyboard status calls.* When this option is set, status requests are passed down immediately and unconditionally. When disabled, pass-downs are limited so the 8042 auxiliary processor does not become overloaded by keyboard polling loops.

0x10 — *Restore only changed interrupts.* Normally, when a DOS/4GW program terminates, all interrupts are restored to the values they had at the time of program startup. When you use this option, only the interrupts changed by the DOS/4GW program are restored.

14.3 Controlling Address Line 20

This section explains how DOS/4GW uses address line 20 (A20) and describes the related DOS16M environment variable settings. It is unlikely that you will need to use these settings.

Because the 8086 and 8088 chips have 20-bit address spaces, their highest addressable memory location is one byte below 1MB. If you specify an address at 1MB or over, which would require a twenty-first bit to set, the address wraps back to zero. Some parts of DOS depend on this wrap, so on the 80286 and 80386, the twenty-first address bit is disables. To address extended memory, DOS/4GW enables the twenty-fist address bit (the A20 line). The A20 line must be enabled for the CPU to run in protected mode, but it may be either enabled or disabled in real mode.

By default, when DOS/4GW returns to real mode, it disables the A20 line. Some software depends on the line being enabled. DOS/4GW recognises the most common software in this class, the XMS managers (such as HIMEM.SYS), and enables the A20 line when it returns to real mode if an XMS manager is present. For other software that requires the A20 line to be enabled, use the A20 option. The A20 option makes DOS/4GW restore the A20 line to the setting it had when DOS/4GW switched to protected mode. Set the environment variable as follows:

```
set DOS16M=A20
```

To specify more than one option on the command line, separate the options with spaces.

The DOS16M variable also lets you specify the length of the delay between a DOS/4GW instruction to change the status of the A20 line and the next DOS/4GW operation. By default, this delay is 1 loop instruction when DOS/4GW is running on a 386 machine. In some cases, you may need to specify a longer delay for a machine that will run DOS/4GW but is not truly AT-compatible. To change the delay, set DOS16M to the desired number of loop instructions, preceded by a comma:

```
set DOS16M=,loops
```

14.4 The Virtual Memory Manager

The Virtual Memory Manager (VMM) lets you use more memory than your machine actually has. When RAM is insufficient, part of your program is swapped out to file on disk until it is needed again. To use VMM set the DOS environment variable using the following syntax:

```
set DOS4GM= [option[#value]] [option[#value]]
```

If you set DOS4GVM equal to 1, the default parameters are used for all options.

14.4.1 VMM Default Parameters

VMM parameters control the options listed below.

MINMEM	The minimum amount of RAM manageed by VMM. The default is 512K.
MAXMEM	The maximum amount of RAM managed by VMM. the default is 4MB.
SWAPMIN	The minimum or initial size of the swap file. If this option is not used, the size of the swap file is based on VIRTUALSIZE.
SWAPINC	The size by which the swap file grows.
SWAPNAME	The swap file name. The default is DOS4GVM.SWP.
DELETESWAP	Delete swap file on exit. The default is off.
VIRTUALSIZE	The size of the virtual memory space. The default is 16MB.

14.4.2 Changing the Defaults

You can change the default in two ways.

1. Specify different parameter values as arguments to the DOS4GVM environment variable, as shown below.

```
set DOS4GVM=deleteswap maxmem#8192
```

2. Create a configuration file with the extension ".VMC" and call that as an argument to the DOS4GVM variable, as shown below.

```
set DOS4GVM=@NEW4G.VMC
```

14.4.3 The .VMC File

A ".VMC" file contains VMM parameters and settings as shown in the example below.

```
!Sample .VMC file
!This file shows the default parameter values

minmem = 512          At least 512K bytes of RAM is
required
maxmem = 4096         Uses no more than 4MB of RAM
virtualsize = 16384   Swap file plus allocated memory is 16MB
!To delete the swap file automatically when the program
!exits, add deleteswap
!To store the swap file in a directory called SWAPFILE,
!add swapname = c:\swapfile\dos4gvm.swp
```

14.5 Error Messages

`DOS4GW Fatal error: Syntax -- DOS4GW` *Executable.xxx*

This message occurs when you execute the DOS4GW program without giving the name of an executable program on the command line.

`DOS4GW Fatal error: Can't find file to load --` *Filename*

This message indicates that DOS4GW couldn't find the executable file you asked it to execute.

`DOS4GW Fatal error: Can't find loader for file[#code#] -- ` *Filename*

This message indicates that DOS4GW was unable to find a loader that can load an executable of this format.

`DOS4GW Fatal error: Can't initialize loader[#code#] --` *Filename*

This error should not occur - please call technical support if it does.

`DOS4GW Fatal error: Loader failed --` *Loader*

This error should not occur, and may indicate that the executable file has been corrupted.

`DOS4GW Fatal error: INT31 initialization error`

This error should not occur - please call technical support if it does.

`DOS4GW Fatal error: No INT31 action found`

This error should not occur - please call technical support if it does.

`DOS4GW Fatal error: Can't locate DOS extender`

This error should not occur - please call technical support if it does. You may get this error if the name of the "DOS4GW.EXE" file has been changed to something else.

`DOS4GW Fatal error: VMM initialization error[#code#]`

This error should not occur - please call technical support if it does.

Index