SNASM2.1 Saturn

Development System

User's Manual

Contents

Setup

Environment

The Assembler

The Debugger

Utilities

Index

IMPORTANT

The information contained in this publication is subject to change without notice. This publication is supplied “as is” without warranty of any kind, either express or implied, including but not limited to the implied warranties or conditions of merchantability or fitness for a particular purpose. In no event shall Cross Products be liable for errors contained herein or for incidental or consequential damages, including lost profits, in connection with the performance or use of this material whether based on warranty, contract, or other legal theory.

This publication contains proprietary information which is protected by copyright. No part of this publication may be reproduced in any form, or stored in a database or retrieval system, or transmitted or distributed in any form by any means, electronic, mechanical photocopying, recording, or otherwise, without the prior permission of Cross Products Limited.

SNASM2.1 Saturn User’s Manual

Revision History:
Revised, 21 August 1995
Selected Preliminary Release, 28 July 1995

SNASM2 Saturn User’s Manual

Revision History:
Revised, March 1995
Revised, February 1995
Revised, January 1995
Preliminary Release, December 1994

© 1994, 1995 Cross Products Limited. All rights reserved.

SNASM2, the SNASM2 logo, Cross Products, and the Cross Products logo are registered trademarks of Cross Products Limited. All other product names and services in this manual are trademarks or registered trademarks of their respective companies.

Contents

Setup

1 Setup ... 1-1
1.1 About the Hardware Setup ... 1-2
1.2 CartDev Rev. B and Modified Saturn ... 1-6
1.3 CartDev Rev. B and Saturn Programming Box ... 1-8
1.4 Testing the Hardware Setup ... 1-13
1.5 About the Software Setup ... 1-15
1.6 Troubleshooting ... 1-18

Environment

2 The SNASM2 Environment ... 2-1
2.1 The SNASM2 Main Menu ... 2-2

The Assembler

3 Running The Assembler ... 3-3
3.1 Command-line Use ... 3-3

4 Source Code Syntax ... 4-1
4.1 Instruction Set ... 4-1
4.2 Statement Format ... 4-13
4.3 Labels and Symbols ... 4-16
4.4 Constants ... 4-21
4.5 Expressions ... 4-33

5 Assembler Directives ... 5-1
5.1 Overview ... 5-1
5.2 Changing Directive Names ... 5-3
5.3 Equates ... 5-5
5.4 Defining Data ... 5-16
5.5 Changing The Program Counter ... 5-25
5.6 Listings ... 5-32
5.7 Including Other Files ... 5-34
5.8 Setting Target Parameters ... 5-39
5.9 Conditional Assembly ... 5-41
5.10 Manipulating Strings ... 5-55
5.11 Modules ... 5-58

5.12 Options and 68000 Optimisations 5-62
5.13 Custom Errors and Warnings 5-70
5.14 Linking 5-72

6 Macros 6-1
6.1 Introducing Macros 6-2
6.2 Macro Parameters 6-5
6.3 Short Macros 6-14
6.4 Advanced Macro Features 6-17

7 Sections and Groups 7-1
7.1 Overview 7-1
7.2 Introduction to Sections and Groups 7-2
7.3 Sections 7-5
7.4 Groups 7-15

The Debugger

8 The Debugger 8-1
8.1 About the Debugger 8-1
8.2 Running the Debugger 8-2
8.3 The Debugger Interface 8-13
8.4 The Main Window 8-16
8.5 Code Windows 8-30
8.6 The Registers Window 8-38
8.7 The Memory Window 8-40
8.8 The Watch Window 8-46
8.9 The Program Window 8-48
8.10 The Breakpoints Window 8-53
8.11 The Log Window 8-54
8.12 The File Viewer Window 8-55
8.13 The Local Vars Window 8-56
8.14 Breakpoints 8-57
8.15 Expressions 8-67
8.16 Expression Formatting 8-72

Utilities

9 SNMAKE 9-1
9.1 Editor Macros for SNMAKE 9-2
9.2 Project Files 9-3
9.3 Command-line Syntax 9-12

10 SNLIB 10-1
10.1 Running SNLIB 10-1

11 SN2G 11-1
12.1 About SN2G 11-1
12.2 Command-line Syntax 11-1
12.3 Considerations and Limitations 11-2

Appendix

A Hitachi Assembler Compatibility A-1
A.1 Introduction A-1
A.2 Overview of Syntax Differences A-3
A.3 Program Elements A-4

List of Figures

Figure 1-1. CartDev Rev. B and Modified Saturn.. 1-7
Figure 1-2. CartDev Rev. B and Saturn Programming Box 1-9
Figure 1-3. Removing the screws from the Programming Box rear panel 1-10
Figure 1-4. Removing the Programming Box cover.................................. 1-11
Figure 1-5. Connecting the NMI Cable.. 1-12
Figure 7-1. Partitioning target memory into logical blocks 7-3
Figure 8-2. The Main debugger window... 8-16
Figure 8-1. The Registers window. 8-38
Figure 8-2. The Memory window.. 8-40
Figure 8-2. The Breakpoint Configuration dialog box................................ 8-59

This is the only information this page contains.

List of Tables

Table 1-1. Troubleshooting the hardware 1-18
Table 2-1. SNASM2 Main Menu keys 2-2
Table 3-1. SH2 assembler filenames and default extensions. 3-6
Table 3-2. 68000 assembler filenames and default extensions. . 3-7
Table 3-3. Assembler command-line switches. 3-9
Table 3-4. Assembler 68000 command-line quirks. 3-12
Table 4-1. SNASM2 SH2 Instruction Set 4-2
Table 4-1. SNASM2 68000 Instruction Set. 4-3
Table 4-1. Addressing Modes 4-4
Table 4-1. Addressing Modes 4-5
Table 4-1. Pre-defined constants 4-27
Table 4-1. Operator precedence 4-35
Table 4-1. Addressing modes used by ADDRMODE. 4-36
Table 4-1. Symbol types. 4-42
Table 5-1. Assembler command-line optimisations. 5-67
Table 6-1. Conditional assembly macros. 6-16
Table 8-1. Files used by the debugger. 8-3
Table 8-1. Debugger command-line switches. 8-6
Table 8-1. Mode and Width combinations for memory searches.8-43
Table 8-1. Format specifier characters and their effects. 8-74
Table 9-1. SNMAKE macro functions. 9-9
Table 9-2. SNMAKE command-line switches. 9-12
Table 10-1. SNLIB command-line switches. 10-1

This is the only information this page contains.

Setup

About the Hardware Setup

CartDev Rev. B and Modified Saturn

CartDev Rev. B and Saturn Programming Box

Testing the Hardware Setup

About the Software Setup

Troubleshooting

1 Setup

This section shows you how to setup the SNASM2 hardware and software and is organised as follows.

Hardware Installation gives a step-by-step guide to setting up and testing the target hardware.

Software Installation provides supplementary information to the on-screen instructions given in the SNASM2 Install program.

Troubleshooting offers some general advice on correcting any problems with the development system.

Important

The SNASM2 documentation does not attempt to teach you about:

Programming the SEGA Saturn
Programming the Hitachi SH2 processor

See the following included documentation for information about the subjects not covered by the SNASM2 documentation:

Hitachi SH2 Programming Manual
Hitachi SH2 Hardware Manual

1.1 About the Hardware Setup

This section shows how to setup the Saturn Development System hardware. There are two hardware setups: CartDev Rev. B with Modified Saturn; and CartDev Rev. B with Programming Box. This section also provides a guide to configuring the supplied Adaptec 1542CF SCSI Adapter and connecting the Programming Box NMI Cable.

1.1.1 If You Already Have a SCSI Adapter

It is recommended that you install the supplied SCSI Adapter to provide a second SCSI chain for the CartDev. Problems have been known to occur when other devices, such as a hard disk or CD ROM drive, are present on the same chain as the CartDev. For this reason the CartDev should be the only device on the SCSI chain.

1.1.2 Configuring the SCSI Adapter

You may need to configure the SCSI Adapter to work with your existing development PC. The exact configuration will depend on the setup of the development PC. Before installing the SCSI Adapter you should check the settings of any cards in the development PC to avoid any potential conflicts.

To setup the SCSI Adapter to work with your development PC you may need to change one or more of the following Switch Block or SCSI*Select* settings:

Switch Block Settings

Termination

By default the termination is Software Controlled (SW1 OFF or Open). Use this default setting.

I/O Port

By default the I/O Port address is 330-333h (SW2-4 OFF or Open). The I/O Port address should not normally require changing.

Floppy Controller

By default the floppy controller is enabled (SW5 ON or Closed). The floppy controller should be disabled by setting SW5 to OFF or Open.

BIOS

By default the BIOS resides at DC000h. The BIOS address should be changed or turned off (SW6-8 ON or Closed) so that it does not conflict with existing cards in the development PC.

SCSI*Select* Settings

The SCSI Adapter is supplied with the SCSI*Select* program. This program allows most of the option settings to be changed without reconfiguring the SCSI Adapter. After the SCSI*Select* program has been installed you may have to change some of the settings as shown below. To use the SCSI*Select* software boot the development PC and type Ctrl+A when prompted.

Interrupt (IRQ) Channel

By default the SCSI Adapter uses IRQ 11. This may conflict with other cards and have to be changed to a different channel e.g. IRQ 9.

DMA Channel

By default the SCSI Adapter uses Channel 5. This may conflict with other cards and have to be changed to a different DMA Channel e.g. Channel 6.

SCSI ID

By default the SCSI Adapter uses SCSI ID 7. This should not normally require changing.

Example Configuration

An example CONFIG.SYS is shown below. In this example the development PC uses two different SCSI Adapters to provide separate SCSI chains for peripherals and the CartDev. The peripherals chain is hosted by a Adaptec 2842VL SCSI Adapter card and has two SCSI devices: a hard disk and a CD ROM drive. The CartDev chain is hosted by a Adaptec 1542CF SCSI Adapter card and has a single SCSI device: the Saturn CartDev. **Note that the CartDev ASPI driver must be installed before any other ASPI drivers otherwise the CartDev will not be recognised.**

```
...
Rem For 1542CF Adaptec Card (CartDev):
DEVICE=C:\SCSI\ASPI4DOS.SYS /D
Rem For 2842VL Adaptec Card (HD and CD-ROM):
DEVICE=C:\SCSI\ASPI7DOS.SYS /D
DEVICE=C:\SCSI\ASPICD.SYS /D:ASPICD0
...
```

1.1.3 Configuring the CartDev

The CartDev requires minimal configuration before it can be used. There are two settings that may require configuring; the SCSI ID and SCSI Termination.

SCSI ID

Set the SCSI ID rotary switch to an unused ID (as defined by the existing configuration of the development PC) between “2” and “6”. A typical setup uses SCSI ID 5.

Termination

The CartDev should be at the end of the SCSI chain and so should be terminated. How you set the termination depends on the type of TERM (SCSI termination) switch used in the supplied CartDev. If the CartDev has a TERM slider switch set the switch to “1” (ON) to terminat ethe CartDev. If the CartDev has a TERM toggle switch set the switch to “0” (OFF) to terminate the CartDev.

1.1.4 CartDev Power On LED Sequence

When the CartDev is powered on both STATUS LEDs will light for a brief period and then LED0 will blink. LED1 will be on during SCSI communications.

Warning NEVER power the Saturn (or Programming Box) before powering the CartDev. Doing so may damage the Saturn (or Programming Box).
ALWAYS power off the Saturn (or Programming Box) before powering off the CartDev.

1.2 CartDev Rev. B and Modified Saturn

1.2.1 Setting up the Hardware

To setup the development system hardware:

1. Install the SCSI Adapter as described in the supplied Adaptec documentation, with reference to “Configuring the SCSI Adapter” on page 1-3 of this manual.
2. Connect the SCSI Adapter to the CartDev SCSI connector (located on the rear of the CartDev) using the supplied SCSI cable.
3. Connect the Saturn NMI cable to the CartDev SATURN CONTROL INTERFACE connector (located on the rear of the CartDev).
4. Connect the CartDev Interface cable to the Saturn unit.
5. Plug the 5V power supply cable into the CartDev.
6. Plug the Saturn AC power cable and CartDev power supply cable into the outlets.
7. Turn the development PC on.
8. Install the SCSI*Select* software according the the supplied Adaptec documentation. **Note that the CartDev ASPI driver must before any other ASPI drivers otherwise the CartDev will not be recognised.**

   The SCSI Adapter may require configuring using SCSI*Select*. See “SCSISelect Settings” on page 3 for more information.

Note See also “CartDev Power On LED Sequence” on page 5.

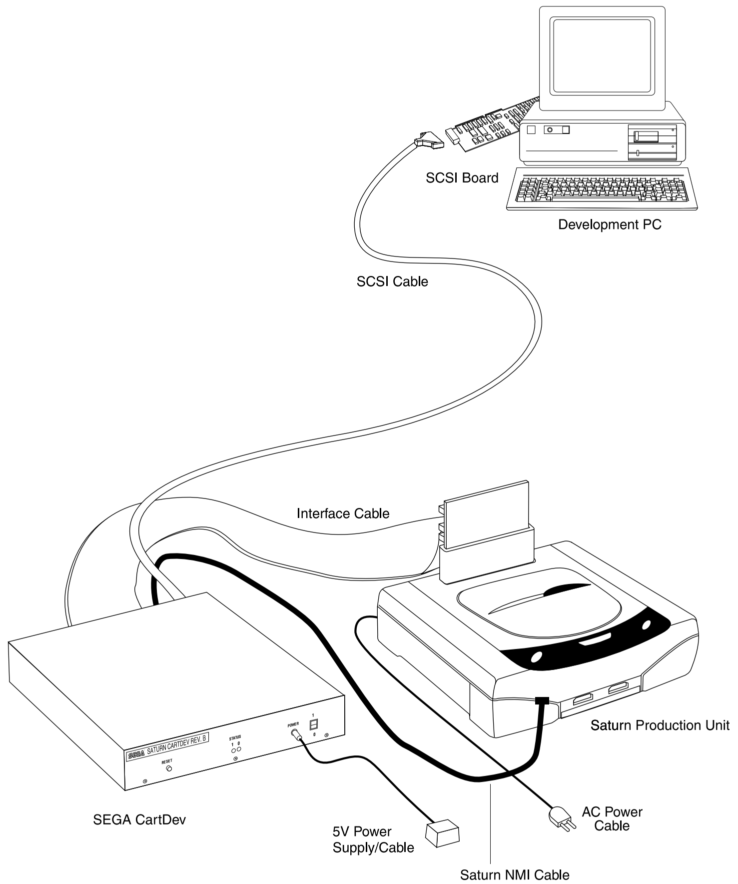


Figure 1-1. CartDev Rev. B and Modified Saturn

1.3 CartDev Rev. B and Saturn Programming Box

This section shows how to setup the CartDev Rev. B and Saturn Programming Box hardware. The Saturn Programming Box is supplied with the NMI Cable already connected. If this was not the case you will have to connect the NMI Cable according to the instructions shown below, otherwise continue to the next section.

1.3.1 Setting up the Hardware

To setup the development system hardware:

1. Install the Adaptec SCSI adapter as described in the supplied Adaptec documentation, with reference to "Configuring the SCSI Adapter" on page 1-3 of this manual.
2. Connect the Adaptec SCSI adapter to the CartDev SCSI connector (located on the rear of the CartDev) using the supplied SCSI cable.
3. Connect the Programming Box NMI cable to the CartDev SATURN CONTROL INTERFACE connector (located on the rear of the CartDev). See "Programming Box NMI Cable Connection" on page 10 if the NMI Cable is not already connected to the Programming Box.
4. Connect the CartDev Interface Cable to the Programming Box.
5. Plug the 5V power supply cable into the CartDev.
6. Plug the Programming Box AC power cable and CartDev power supply cable to the outlets.
7. Turn the development PC on.
8. Install the EZ-SCSI software according to the supplied Adaptec documentation. **Note that the CartDev ASPI driver must before any other ASPI drivers otherwise the CartDev will not be recognised.**

   The SCSI adapter may require configuring using the supplied Adaptec SCSI software. See "SCSISelect Settings" on page 3 for more information.

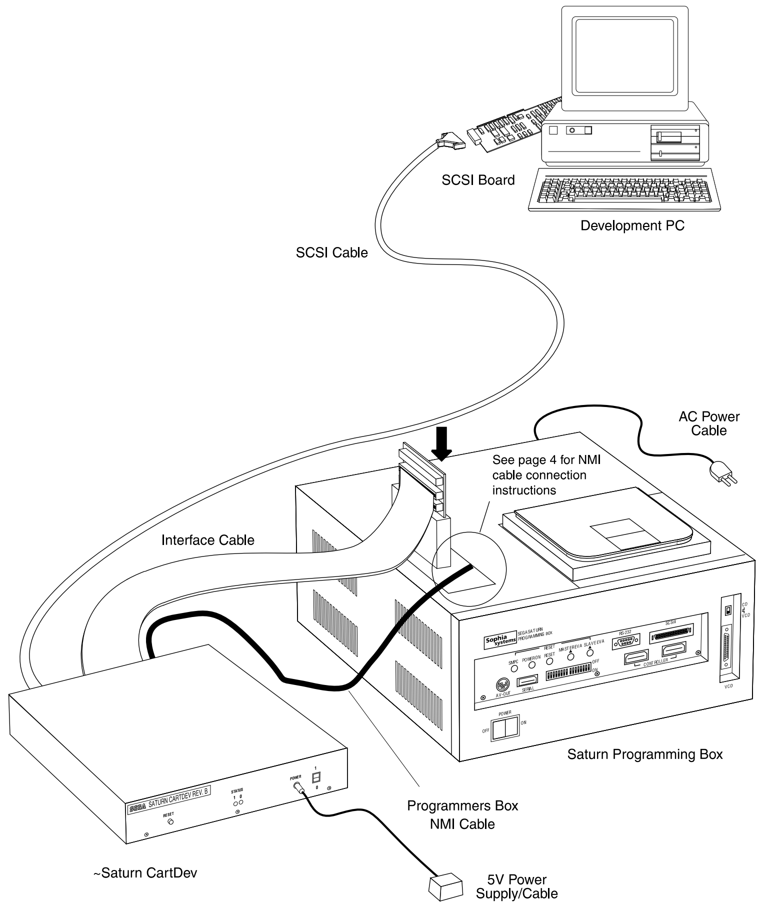


Figure 1-1. CartDev Rev. B and Saturn Programming Box

Note See also “CartDev Power On LED Sequence” on page 5.

1.3.2 Programming Box NMI Cable Connection

This section shows how to connect the NMI cable to the Saturn Programming Box.

To connect the NMI Cable:

1. Remove the two screws, labelled ① in Figure 1-1 below, from the Programming Box rear panel.

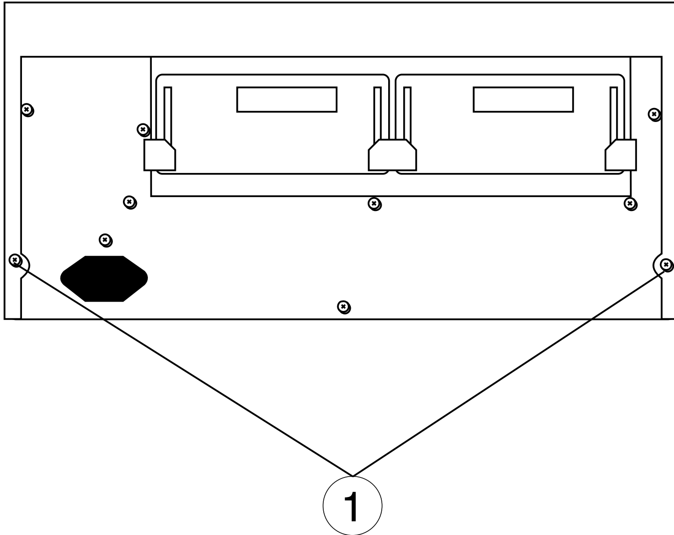


Figure 1-1. Removing the screws from the Programming Box rear panel

2. Remove the two screws, labelled ② in Figure 1-2 below, from the Programming Box side panels and remove the cover.

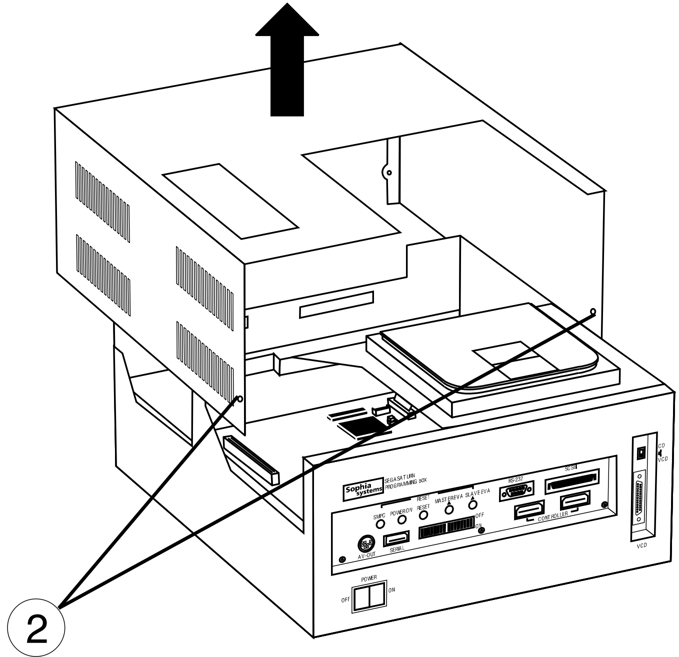


Figure 1-2. Removing the Programming Box cover

3. Plug the Programming Box NMI Cable into connector CN8, labelled ③ in Figure 1-3 below, on the Programming Box motherboard.

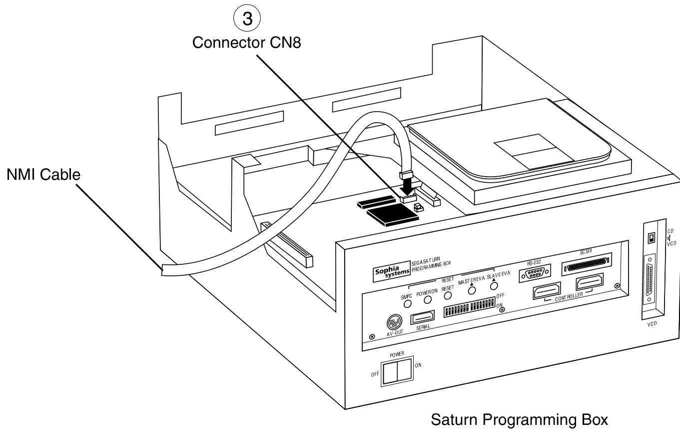


Figure 1-3. Connecting the NMI Cable

4. Attach the grounding lug to a Programming Box chassis screw.
5. Replace the cover and screws.

1.4 Testing the Hardware Setup

Before installing the SNASM2 software you should test that the development hardware is functioning correctly. Performing a test at this stage helps to isolate possible reasons for failure in the event that the complete system does not function correctly.

Before proceeding check that:

- The Adaptec card is correctly installed
- The ASPI software is correctly installed
- The CartDev is correctly terminated
- The CartDev Termination Power light is on

1.4.1 Hardware Setup Test

To test the development hardware:

- Reboot the development PC. During the boot sequence there will be a number of messages displayed on the screen. There are two messages that relate to the functioning of the development hardware, described below.
- The first message is displayed during the PC's BIOS boot phase and should be similar to

```
SCSI ID #0 - SEGA OA Saturn CartDev
```

  This indicates that:

  - The CartDev is turned on
  - The SCSI bus is terminated correctly
  - The SCSI cable between the CartDev and the Adaptec card is connected correctly
  - The Adaptec card is installed correctly.

- The second message is displayed after the PC's BIOS boot phase is complete and should be similar to

```
Host Adapter #0 - SCSI ID 5-LUN0: SEGA OA Saturn CartDev BO69*
```

*In this message, BO69 refers to the CartDev firmware revision number and may be greater than stated above.

This indicates that:

- The ASPI drivers are installed
- The ASPI drivers have located the CartDev and are able to communicate with it.

1.5 About the Software Setup

1.5.1 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. SNASM2 currently supports the following editors:

- Brief
- Multi-Edit

1.5.2 Installing the SNASM2 Software

Requirements

The SNASM2 software requires:

- IBM or 100% compatible PC with a 386 or greater processor.
- At least 2MB (4MB or greater recommended) RAM
- Approximately 5MB of free disk space.

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

Installation

To install the SNASM2 software:

1. Insert SNASM2 disk 1 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 exit the installation at any time by pressing the Esc key.

1.5.3 Changes to AUTOEXEC.BAT

The SNASM2 Install program can optionally make changes to your AUTOEXEC.BAT file. You will be prompted to accept or reject each change before it is made. This section describes the changes that can be made.

General

1. The name of the directory in which the SNASM2 software was installed can be added to the PATH variable.

Brief Variables

1. -MSNASM and -MSNASM1 can be added to the BFLAGS line. This enables the SNASM2 macros to be invoked automatically when Brief starts up.
2. A variable can be setup to tell Brief to start a make (via the SNMK_RUN macro) when Alt+F10 (the compile key in Brief) is pressed. The variable is of the form BC*xxx*, where *xxx* is the default filename extension for the default source code type. For example, if this is to be '.SH2' the line:

   ```
   set bcsh2=snmk_run
   ```

   is added to AUTOEXEC.BAT.
3. If Brief EMS/XMS swapping is switched off (no 'M' in the BFLAGS line) the install program can add -M to that line to switch it on. This is recommended to provide more conventional RAM during a make.
4. If it is not already set, install will recommend that the BFILE variable is set to 'STATE.RST' . This is to give status to files saved in the current directory. Accepting this recommendation will add the following line to AUTOEXEC.BAT:

   ```
   set bfile=state.rst
   ```

5. The BTMP variable can be pointed to a fast disk (usually a RAM disk) to facilitate disk swapping when necessary. Accepting this prompt will add the following line to AUTOEXEC.BAT:

   ```
   set btmp=x:
   ```

   where *x* is the specified disk drive letter.

Multi-Edit

No changes are made to your AUTOEXEC.BAT file.

1.6 Troubleshooting

Table 1-1 below provides possible remedies for the most common hardware related problems. If after trying to correct the problem the hardware still does not function correctly please contact technical support for assistance.

Problem	Reason
The Adaptec card fails to see the CartDev.	The CartDev may not be powered up.
	The Adaptec SCSI ID may be conflicting with other SCSI adaptors.
	The CartDev SCSI ID may be conflicting with other SCSI devices.
The debugger fails to see the CartDev.	The target hardware may not be connected properly. When the debugger successfully sends the monitor code to the target the standard SEGA copyright text will appear briefly.

Table 1-1. Troubleshooting the hardware

The SNASM2 Environment

The SNASM2 Main Menu

2 The SNASM2 Environment

The SNASM2 development system can be used from within two supported text editors; Brief and Multi-Edit. SNASM2 is supplied with a set of macros to customise these editors enabling you to use SNASM2 from within your favourite programming editor.

The SNASM2 Software Installation program can optionally customise an editor, using the editors' macro language, to provide a development environment for the SNASM2 toolset.

The SNASM2 Development Enviornment provides menu driven access to the SNAMS2 development tools and extra functions. This section describes the SNASM2 environment, detailing menu options and keyboard controls.

Project Files

See also "SNMAKE" on page 9-1.

The behaviour of menu commands is determined by the contents of a specified Project File. You must have at least one project file for commands to function.

The SNASM2 environment relies on the creation of a *Project File* for each project the user is undertaking. The project file contains information about the file dependencies for a project and the rules governing how the output file(s) can be created. The SNMAKE utility uses the project file to determine which targets have dependants that have been updated since the target file was created, and therefore which targets must be recreated from their dependants. This means that commands and their options must be put into the project file in the same way as they would be entered from the command-line.

2.1 The SNASM2 Main Menu

The SNASM2 Main Menu is invoked by pressing Alt+F9. The menu items and options are discussed in sequence below. Navigation through the menus is performed using the Up and Down arrow keys and items are selected using the Enter key. The Escape key will return to the editor.

Description	Key
Make	Alt+F10 (Alt+F8*)
Select Project File	Ctrl+F9
Debug	Ctrl+F10
Set Debug Mode	Ctrl+D
Evaluate	Ctrl+E
Jump to Label	Ctrl+G
Undo Last Label	Ctrl+F
Save All Buffers	Alt+S
Error Window	Ctrl+Q
Next Error	Ctrl+N

Table 2-1. SNASM2 Main Menu keys. * Multi Edit only.

Make (Alt+F10) (Alt+F8 in Multi-Edit)

Invokes the SNMAKE utility using the current project file.

Select Project File (Ctrl+F9)

Displays a window listing all project files (‘.PRJ’) in the current directory, highlighting the current file. Use the cursor keys to highlight a file and Enter to select it. This will display a pop-up menu with four items:

Select this file Makes the highlighted file the current project file i.e. invoking the Make option will start SNMAKE with the highlighted project file.

Select and Make	Make the highlighted project file current and invoke the make utility.
Show comment lines	The first line of each file, usually a comment, is displayed to the right of the filename. To see the remaining comments select the file and chose Show comment lines from the pop-up menu. This displays all the text in the project file from the top of the file to the `[SNMAKE]` label. This item is not available in Multi-Edit.
Edit and select	Edit the highlighted project file.

Debug (Ctrl+F10)

Invokes the debugger specified in the current project file.

Set Debug Mode (Ctrl+D)

Selecting this option brings up a sub-menu with two options: On and Off, one of which will be highlighted. These options control the setting of the special macro '$!' in the project file. This macro expands to the settings of the debug and info switches on the SNASM2 command-line. Info mode is always on when SNMAKE is invoked from within an editor. This displays a Status window during assembly allowing progress to be monitored. Debug mode can be set from this menu option, determining whether the program being made, (assuming it is being downloaded to a target machine) is run immediately (debug mode Off) or waits with the program counter set to the value specified by the user with the REGS directive (debug mode On). Note that this control depends on tthe correct use of the special macro '$!' in the project file.

Evaluate (Ctrl+E)

This option invokes the expression evaluator specified in the current project file. Any text highlighted in the current window is passed to the expression evaluator, otherwise further input is requested.

Jump to label (Ctrl+G)

See also "Labels and Symbols" on page 4-16.

This option examines the current cursor position and determines if it is on a valid label. If so it jumps to that label, if not it prompts for a label name to look for.

Undo last label (Ctrl+F)

This option undoes the effects of the last Jump to label.

Save all buffers (Alt+S)

This option saves all buffers currently being edited.

Error Window (Ctrl+Q)

This option opens an Error window displaying the current contents of the error file ERRORS.ERR located in the current directory. This is the file to which all error output is redirected by SNMAKE. If there are errors (in a format that this function can understand) they can be stepped through using the Up and Down arrow keys. Pressing Enter on a highlighted error will displays the location of the error in the relevant source file. The Home and End keys can be used to move from the top and bottom of the error buffer respectively and pressing Ctrl whilst using the up and down arrow keys allows the user to move around in the error buffer line by line. Striking enter on a line that the macros do not recognise as containing an error message will result in an error message to that effect.

Next Error (Ctrl+N)

This option scans the file ERRORS.ERR and moves the cursor to the next error in the source code. Repeated invocations will step through the errors. If no further errors can be found a message to that effect is displayed on the status line.

The Assembler

Running The Assembler

Source Code Syntax

Assembler Directives

Macros

Sections and Groups

The Assembler

The assembler translates assembly language source files into binary files which can be loaded into memory and executed. The SNASM2 assembler 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 (Common Object File Format) file format. This allows mixed language projects and in addition support is provided for linking with Sierra C and GNU format COFFs.

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 an extensive superset of features found in other assemblers including:

- Rationalised syntax whilst maintaining maximum 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

This is the only information this page contains.

3 Running The Assembler

This section shows how to run the assembler from the command-line. To run the assembler from within one of the supported text editors see Chapter 2, "The SNASM2 Environment".

3.1 Command-line Use

The OPT directive is described on page 5-68.

This section shows you 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 assembly code source files using the OPT directive.

3.1.1 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:

See also "Assembler Command Files" on page 3-13.

`[snasm68k|snasmsh2]` *Switches Source,Object,Map,List*

or

`[snasm68k|snasmsh2]` *Switches @CommandFile*

To halt the assembly type Ctrl+C or Ctrl+Break. This halts the assembler after deleting any temporary or partially written output files.

See also "Example 2" on page 3-4.

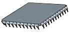
The assembler can accept multiple source files using the format '*Filename.Ext+Filename.Ext+...*', treating each file as if it were an include. Note that when assembling in this way you *must* specify the standard source file extensions so that the assembler can, for example, differentiate between source files and COFF files. If no extension is given, the assembler will choose source files over object files.

Important

See also "Example 3" on page 3-5.

SNASM68K can also download object code direct to the target or simultaneously create a COFF file and download object code to the target.

SNAMSH2 cannot download directly to the target. The assembler must first generate a COFF file that can subsequently be downloaded to the target using the debugger.

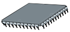


68000

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
```



SH2

Example 1

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

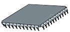
```
snasmsh2    test.sh,test.cof,test.map
```

68000

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
```

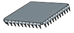


SH2

Example 2

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

```
snasmsh2 test1.sh+test2.sh+test1.cof,test.cof,test.map
```



68000

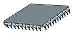
Example 3

This example assembles the source file TEST.68K, downloads the object code to target 4 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,t4:test.cof
```

To subsequently enter the debugger with the debug info use:

```
snbugsat -t4:test.cof
```



SH2

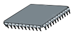
Example 3

This example assembles the source file TEST.SH and generates a COFF file, TEST.COF, that includes source debug info (/sdb).

```
snasmsh2 /sdb     test.sh,t1:test.cof
```

To subsequently enter the debugger with the debug info use:

```
snbugsat -t4:test.cof
```



68000

Example 4

This example is similar to Example 1 except that the object code is sent to target 4.

```
snasm68k    test.68k,t4:,test.map
```

3.1.2 File Extensions

The file types and default extensions used by the assemblers are given below.

File	Default Extension	Description
Source	SH, .ASM, .S	Contains the source code to be assembled. If no source file is specified the assembler will print a help message and a description of the command-line syntax.
Object	.COF, .O, .OBJ	Receives the object code output. If no file is specified then object code will be not be generated unless group FILE statements are present. (See page 7-15. for more information about group attributes.)
Library	.LIB, .A	Library files in either SNLIB format or from the GNU archiver.
Binary	.BIN	Binary Files.
List	.LST	Receives any listing output.
Map	.MAP	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.
S-Records	.S19	Motorola S-records.

Table 3-1. SH2 assembler filenames and default extensions.

File	Default Extension	Description
Source	.68K, .ASM, .S	Contains the source code to be assembled. If no source file is specified the assembler will print a help message and a description of the command-line syntax.
Object	.COF, .O, .OBJ	Receives 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 7-15. for more information about group attributes.)
Library	.LIB	Library files.
Binary	.BIN	Binary Files.
List	.LST	Receives any listing output.
Map	.MAP	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.
S-Records	.S19	Motorola S-records.

Table 3-2. 68000 assembler filenames and default extensions.

Source Filename Extensions

If a source filename is specified with no extension and does not exist with that name, the assembler will search for a file with the specified root name and one of the default extensions. Source files can also take any extension that was specified as part of the software installation process. Note that it is recommended that output files are specified without extensions; the assembler will automatically append the appropriate extension.

Ignoring Specified Files

The assembler can be made to ignore files specified on the command-line. This provides the ability to, for example, specify a filename for a map or list file but prevent them from being generated when it is inconvenient. To prevent a file from being generated prefix the filename with a '!' character. The assembler then treats the filename as blank. This feature is particularly useful in make, project or batch files.

Concatenating Map and List Files

Map and List filenames can be suffixed with a '+' to concatenate output generated by the assembler with existing files of the same name.

3.1.3 Switches

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

Note

There must be at least one space between a switch and any parameters and that this is not compatible with versions of the assembler prior to version 2.0.

Switch	Description
`[-|/]?`	Displays on-line help describing the syntax for switches, options and optimisations.
`[-|/]b` *Size*	Set the *Size* of the input buffers from 1K-64K, 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. Note that the endianness of a COFF file refers only to its structure and not to the endianness of the processor to which it is loaded. By default the COFf files is generated in the native endianess of the host processor.
`[-|/]dmax` *Num*	Controls the maximum data size that can be generated for a single DS or DCB directive. *Num* is in the range 1-32 where `dmax`=2^{Num}. By default *Num* is set to 16 (i.e. `dmax`=2^{64}) allowing up to 64K of space to be reserved by one .DATA, DS or DCB statement. The assembler will generate an error if the size exceeds 2^{dmax}.
`[-|/]e` *Symbol*{=[*Val*|"*Str*"]} {;*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 30; a value of zero will cause the assembly to continue regarless of the number of errors generated.

Table 3-3. Assembler command-line switches.

Switch	Description
[-|/]g	Write non-global symbols to linker object file.
[-|/]**hex** *Number*	Set the width of hexadecimal output in the listing file from two to eight words, the default being four.
[-|/]i	Display information window during assembly.
[-|/]im	Relax rules about importing symbols. If generating linkable output any undefined symbols are automatically marked as imported. Without the im switch any such undefined symbols which are not explicitly imported will generate an error.
[-|/]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 input files in the current directory. If not found there the assembler looks in the directories built up using the 'j' switch. Multiple j switches can be specified each one adding to...
[-|/]k	Enable additional conditional assembly structures. These are implemented via macros and described on page 6-16.
[-|/]l	Produce linkable output file; allow unresolved external references.
[-|/]lnos	Show source code line numbers in the listing file.

Table 3-3. Assembler command-line switches.

Switch	Description
`[-|/]o` *Options*	Set assembler options and optimisations. Note that there must be at least one space character between the switch and the parameter. For more information on Options see “Options and 68000 Optimisations” on page 5-62.
`[-|/]p`	Produce pure binary output file. See also the `rom` switch.
`[-|/]q` *Quirks*	Enable quirks. Quirks are special options that enable certain features specific to SNASM version 1.x. Is this relevant for Saturn?
`[-|/]rom`	Produce ROM image. This produces a pure binary output file with the space between groups padded to place them at their ORG addresses. See also the `p` switch.
`[-|/]s`	Produce Motorola S-Record output file.
`[-|/]sdb`	Output source debug information to COFF file.
`[-|/]t`	Truncate values in DC.B and DC.W directives to bytes and words respectively.
`[-|/]w`	Write equates to symbol table.

Table 3-3. Assembler command-line switches.

3.1.4 68000 Quirks

Quirks are known incompatibilities between SNASM2 68000 and version 1.x. The behaviour of the assembler prior to version 2 occasionally became ‘eccentric’ as more features were added to it. This behaviour has been rationalised in SNASM2 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 identify anything using a quirk and change it to be compatible with SNASM2.

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.
`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.
`sc-/+`	*Sierra C SDB format*. Set this quirk if, when using SDB, source and assembler loose synchronisation.

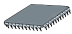
Table 3-4. Assembler 68000 command-line quirks.

3.1.5 Assembler Command Files

A command file contains assembler command-line parameters separated by white space or line breaks. The filename 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.

The command file can be used in place of or in addition to command-line parameters. If the assembler is invoked with both switches and a command file, the switches take precedence over the contents of the command file (which must be the last item on the command-line) as they can be set once only.

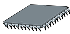
Multiple assemblies can be specified in a single command file. The assembler searches the command file until it finds a valid assembly request which it then executes. The assembler then continues searching the command file for further assembly requests. Performing multiple assemblies using a command file is faster than invoking the assembler for each assembly as the assembler is loaded only once.



SH2

Example

```
# Assemble TEST.SH and output TEST.COF.
# Generate the MAP file TEST.MAP but no listing or
# temporary files.

test.sh,test.cof,test.map


# Assemble TEST1.SH and TEST2.SH, and TEST.COF
# Output TEST1.COF and generate TEST.MAP map file.

test1.sh+test2.sh+test1.cof,test.cof,test.map


#Assemble TEST.SH
#Ouput TEST.COF, including source debug info (/sdb).

/sdb  test.sh,t1:test.cof
```


68000

Example

```
# Assemble TEST.68k and output TEST.COF.
# Generate the MAP file TEST.MAP but no listing or
# temporary files.

test.68k,test.cof,test.map


# Assemble TEST1.68k and TEST2.68k, and TEST.COF
# Output TEST1.COF and generate TEST.MAP map file.

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


#Assemble TEST.68k
#Ouput TEST.COF, including source debug info (/sdb).

/sdb  test.68k,t4:test.cof
```

4 Source Code Syntax

4.1 Instruction Set

The SNASM2 development system is fully compatible with the Hitachi SH and Motorola M68000 processors. The assemblers, SNASMSH2 and SNASM68K, support the standard Hitachi and Motorola mnemonics and will generate code for supported extensions of the instruction as determined by the addressing mode. The SNASMSH2 and SNASM68K instruction sets consists of the standard Hitachi and Motorola opcodes, common extensions and some SNASM2 specific extensions. The complete SNASMSH2 and SNASM68K instruction sets are given below.

SNASM2 SH2 Instruction Set			
ADD	CMPHI	MOVT	SHLL16
ADD.L	CMPHI.L	MOVT.L	SHLL16.L
ADDC	CMPHS	MOVE	SHLL2
ADDC.L	CMPHS.L	MOVE.B	SHLL2.L
ADDV	CMPPL	MOVE.L	SHLL8
ADDV.L	CMPPL.L	MOVE.W	SHLL8.L
AND	CMPPZ	MULL	SHLR
AND.B	CMPPZ.L	MULS	SHLR.L
AND.L	CMPSTR	MULS.L	SHLR16
BF	CMPSTR.L	MULS.W	SHLR16.L
BF/S	DIV0S	MULU	SHLR2
BRA	DIV0S.L	MULU.L	SHLR2.L
BRAF	DIV0U	MULU.W	SHLR8
BSR	DIV1	NEG	SHLR8.L
BSRF	DIV1.L	NEG.L	SLEEP
BT	DMULS	NEGC	STC
BT/S	DMULS.L	NEGC.L	STC.L
CLRMAC	DMULU	NOP	STS
CLRT	DMULU.L	NOT	STS.L
CMP/EQ	DT	NOT.L	SUB
CMP/EQ.L	EXTS	OR	SUB.L
CMP/GE	EXTS.B	OR.B	SUBC
CMP/GE.L	EXTS.W	OR.L	SUBC.L
CMP/GT	EXTU	ROTCL	SUBV
CMP/GT.L	EXTU.B	ROTCL.L	SUBV.L
CMP/HI	EXTU.W	ROTCR	SWAP
CMP/HI.L	JMP	ROTCR.L	SWAP.B
CMP/HS	JSR	ROTL	SWAP.W
CMP/HS.L	LDC	ROTL.L	TAS
CMP/PL	LDC.L	ROTR	TAS.B
CMP/PL.L	LDS	ROTR.L	TRAPA
CMP/PZ	LDS.L	RTE	TRAPA.L
CMP/PZ.L	MAC	RTS	TST
CMP/STR	MAC.L	SETT	TST.B
CMP/STR.L	MAC.W	SHAL	TST.L
CMPEQ	MOV	SHAL.L	XOR
CMPEQ.L	MOV.B	SHAR	XOR.B
CMPGE	MOV.L	SHAR.L	XOR.L
CMPGE.L	MOV.W	SHLL	XTRCT
CMPGT	MOVA	SHLL.L	XTRCT.L
CMPGT.L	MOVA.L		

Table 4-1. SNASM2 SH2 Instruction Set

SNASM2 68000 Instruction Set		
ABCD	EOR	NEGX
ADD	EORI	NOP
ADDA	EORI to CCR	NOT
ADDI	EORI to SR	OR
ADDQ	EXG	ORI
ADDX	EXT	ORI to CCR
AND	ILLEGAL	ORI to SR
ANDI	JMP	PEA
ANDI to CCR	JSR	RESET
ANDI to SR	LEA	ROL ROR
ASL,ASR	LINK	ROXL ROXR
Bcc.S	LSL,LSR	RTD
BCHG	MOVE	RTE
BCLR	MOVEA	RTR
BKPT	MOVEC	RTS
BRA	MOVEM	SBCD
BSET	MOVEP	Scc
BSR	MOVEQ	STOP
BTST	MOVES	SUB
CHK	MOVE from CCR	SUBA
CLR	MOVE to CCR	SUBI
CMP	MOVE from SR	SUBQ
CMPA	MOVE to SR	SUBX
CMPI	MOVE USP	SWAP
CMPM	MULS	TAS
DBcc	MULU	TRAP
DIVS	NBCD	TRAPV
DIVU	NEG	TST
		UNLK

Table 4-1. SNASM2 68000 Instruction Set.

4.1.1 SNASMSH2 Addressing Modes

The SNASMSH2 assembler provides addressing mode syntax compatible with the SH2.

SH2 Style	Description
#n ##n	Immediate
Rn	Direct Register
@Rn	Indirect Register
@Rn+	Post-increment Indirect Register
@-Rn	Pre-decrement Indirect Register
@(R0,Rn) @(Rn,R0)	Indirect Indexed Register
@(R0,GBR) @(GBR,R0)	Indirect Indexed GBR
@(disp:4,Rn) @(Rn,disp:4)	Indirect Register with Displacement
@(disp:8,GBR) @(GBR,Rn)	Indirect GBR with Displacement
@(disp:8,PC) @(PC,disp:8)	PC Relative with Displacement

Table 4-1. Addressing Modes

4.1.2 68000 Addressing Modes

The SNASM68K assembler provides addressing mode syntax compatible with the M68000.

68000 Stye	Description
#n	Immediate
Rn	Direct Register
(Rn)	Indirect Register
(Rn)+	Post-increment Indirect Register
-(Rn)	Pre-decrement Indirect Register
(R0,Rn) (Rn,R0)	Indirect Indexed Register
Rn(R0)	
R0(Rn)	
(R0,GBR) (GBR,R0)	Indirect Indexed GBR
R0(GBR)	
GBR(R0)	
(disp:4,Rn) (Rn,disp:4)	Indirect Register with Displacement
disp:4(Rn)	
(disp:8,GBR) (GBR,disp:8)	Indirect GBR with Displacement
disp:8(GBR)	
(disp:8,PC) (PC,disp:8)	PC Relative with Displacement
disp:8(PC)	

Table 4-1. Addressing Modes

4.1.3 Literal Pools

Literals which evaluate and are too large to fit in the instruction are placed in a literal pool and referenced using PC relative addressing. Both long and word literals may be placed in the literal pool. Unless specified with '##' literals which do not evaluate are always placed in the literal pool. Note that the literal pool is only available to the MOV instruction.

LITS

The LITS directive causes the literal pool to be emitted. When emiting the literal pool, any pool entries whose values are known and identical are merged together. Similarly, simple forward referenced symbols are also merged. Expressions that have not evaluated are never merged, even if they are identical.

Syntax

`lits`[.*Qualifier*]

where:

Label is an optional symbol defined by this statement.

Qualifier is an optional qualifier that can be:

`.w` causes LITS to emit word literals only
`.l` causes LITS to emit long literals only

The LITS directive with no qualifier emits all word literals followed by all long literals.

The RISC Option

Literal pools are handled differently depending on the setting of the RISC option. With the RISC option off (`risc-`, the default) literals can be specified using either '#' or '##'.

Literals specified using '##' are always generated as part of the instruction and therefore their values must be in range. Literals

specified using ‘#’ are generated in the instruction if they can be evaluated and their value is in range. If If these literals cannot be evaluated, for example because of a forward reference, or their value is out of range then they are placed in the next requested literal pool.

With the RISC option on (`risc+`) literals can be specified using ‘=’ or ‘#’. Literals specified using ‘=’ will always generate a literal pool value and literals specified using ‘#’ will always be forced into the instruction.

Example

The annotated listing below shows the operation of the literal pool.

```
         = 00000006       | q1    equ     6
         = 000000a0       | q1a   equ     160
                          |
00000000: 70FF            |       add     #-1,r0
00000002: C904            |       and     #4,r0
00000004: 70              |       add     #129,r0
```

Case 1. This generates an error on pass 1 with the message `"Cannot fit value 129 (0x81) into signed byte".` *This is because the literal pool is unavailable to ADD and the value 129 cannot fit into the signed byte available for the instruction.*

```
00000005: 81
00000006: 7006            |       add     #q1,r0
00000008: 70@@            |       add     #q2,r0
0000000a: 70              |       add     #q1a,r0
```

Case 2. This generates an error on pass 1 with the message `"Cannot fit value 160 (0xa0) into signed byte".` *This is because the literal pool is unavailable to ADD and the value of* `q1a`*(160) cannot fit into the signed byte available for the instruction.*

```
0000000b: A0
0000000c: 70@@            |       add     #q2a,r0
```

Case 3. This generates an error on pass 2 with the message `"Cannot fit value 14d into signed byte".` *This is because* `q2a` *is undefined at this point so the assembler assumes that when* `q2a` *is defined its value will fit the instruction i.e.* `#q2a` *is interpreted as* `##q2a`.

`q2a` *is subsequently defined with a value of 333 (0x14d). The literal pool is unavailable to ADD and the value of* `q2a` *is too large to fit the instruction and so generates an error.*

```
0000000e: 70FF          |       add     ##-1,r0
00000010: C904          |       and     ##4,r0
00000012: 70            |       add
##129,r0
```

Case 4. This generates an error on pass 1 with the message `"Cannot fit value 129 (0x81) into signed byte".` *This is because the literal pool is unavailable to ADD so that #129 and ##129 are equivalent causing an error for the same reasons as for Case 1.*

```
00000013: 81
00000014: 7006          |       add     ##q1,r0
00000016: 70@@          |       add     ##q2,r0
00000018: 70            |       add
##q1a,r0
```

Case 5. This generates an error on pass 2 with the message `"Cannot fit value 160 (0xa0) into signed byte".` *This is because the literal pool is unavailable to ADD so that* `#q1a` *and* `##q1a` *are equivalent causing an error for the same reasons as for Case 2.*

```
00000019: A0
0000001a: 70@@          |       add
##q2a,r0
```

Case 6. This generates an error on pass 2 with the message "Cannot fit value 333 (0x14d) into signed byte". *This is because* q2a *is undefined at this point so the assembler assumes that when* q2a *is defined its value will fit the instruction i.e.* #q2a *is interpreted as* ##q2a. q2a *is subsequently defined with a value of 333 (0x14d) causing an error for the same reasons as for Case 3.*

```
0000001c: E0FF          |       mov     #-1,r0
0000001e: E004          |       mov     #4,r0
00000020: D0??          |       mov     #129,r0
00000022: E006          |       mov     #q1,r0
00000024: D0??          |       mov     #q2,r0
00000026: D0??          |       mov     #q1a,r0
00000028: D0??          |       mov     #q2a,r0
                        |
0000002a: E0FF          |       mov     ##-1,r0
0000002c: E004          |       mov     ##4,r0
0000002e: E0            |       mov
##129,r0
```

Case 7. This generates an error on pass 1 with the message "Cannot fit value 129 (0x81) into signed byte". *This is because the "*##*" makes the literal pool unavailable to MOV causing an error for the same reasons as for Cases 1 and 4.*

```
0000002f: 81
00000030: E006          |       mov     ##q1,r0
00000032: E0@@          |       mov     ##q2,r0
00000034: E0            |       mov
##q1a,r0
```

Case 8. This generates an error on pass 1 with the message "Cannot fit value 160 (0xa0) into signed byte". *This is because the "##" makes the literal pool unavailable to MOV causing an error for the same reasons as for Cases 2 and 5.*

```
00000035: A0
00000036: E0@@          |       mov
##q2a,r0
```

Case 9. This generates an error on pass 2 with the message
`"Cannot fit value 14d into signed byte".`
This is because the "`##`" makes the literal pool unavailable to MOV causing an error for the same reasons as for Cases 3 and 6.

```
         = 00000003      | q2     equ     3
         = 0000014d      | q2a    equ     333
                         |
00000038: @@@@ @@@@ 00.. |        lits
```

This is the only information thispage contains.

4.2 Statement Format

Each statement has the following general format:

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

White Space

See also "Options and 68000 Optimisations" on page 5-62.

Each field must be separated by *white space* i.e. any combination of tabs and spaces. To improve code readability, white space is allowed in the operand field after a comma or operator.

The *White Space* option determines whether the comment field starts with white space or a semicolon. By default (`ws-`), 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 semicolon (;). The assembler can be made to insist on a semicolon before comments by setting the *White Space* option (`ws+`). Setting this causes the assembler to ignore white space in the operand field and is a highly recommended option.

Comment Lines

The exception to the statement format is comment lines. These can begin with a semicolon (;), an asterisk (*) or, if GNU mode is on (`g+`), a forward slash followed immediately by an asterisk (/*). A comment line can begin in any column if it begins with ';' or '/*' but must start in column 1 if it begins with '*'. A comment line beginning with '/*' can cross line boundaries but a comment line beginning with a ';' or '*' must terminate on the line it was started on. A comment line beginning with '/*' must be terminated with '*/' but a comment line beginning with a ';' or '*' does not require a termination character. Blank lines and lines that contain white space only are treated as comment lines. All comments are ignored by the assembler.

Line Continuation Character

The maximum line size is 1024 characters. The line continuation character '&' can be used to continue a statement on to the next line. The '&' character can be used in comments but will be interpreted as an ampersand and not as a line continuation character. The '&' character must be the last symbol on the line.

If the 68000 Quirk *Macro continuation character* (`mc+`) is set you can use '\' as a line continuation character in macros. Again, the '\' character must be the last symbol on the line.

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
```

4.2.1 Changing the Processor Mode

The PROC directive changes the operand mode. Use PROC to switch between SH1 and SH2 as the operand.

Syntax

```
proc  [sh1|sh2]
```

4.3 Labels and Symbols

Symbols can contain up to 80 characters from the following symbol set:

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

If a symbol is longer than 80 characters, it will be truncated and the assembler will generate a ‘Label Truncated’ warning. The first character of a symbol must not be a digit 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.

4.3.1 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 a label is used it *must* start in column 1 unless it ends with a colon (:). The colon is not treated as part of the label.

Note that if GNU mode is on (g+) labels must terminate in a colon (:); this allows mnemonics to start in column 1.

Global labels *must* conform to the following format:

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

4.3.2 Local Labels

See also "Scoping Local Labels" on page 4-19.

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.

See also "Symbols and Periods" on page 4-20.

Local labels *must* conform 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.

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 (`l+` to toggle between '@' and '.' or `l` *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
           ...
           optl 63          ; Change to ? (63
ASCII)
?LevelOne
?Alert                      ; Same label as before
           optl '@'         ; Change back to @
@Alert                      ; Still same label
```

4.3.3 Scoping Local Labels

See also "Scoping Local Labels" on page 4-19.

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*. Local labels and modules are also available inside macros with further macro specific facilities provided by the '\@' parameter and the LOCAL directive.

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 cause local labels to be de-scoped.

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
```

4.3.4 Symbols and Periods

The assembler allows the use of symbols containing periods, providing greater flexibility in choosing label names. However, this is not recommended 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 and that label value is used; otherwise the string is scanned from right to left. Strings are scanned from right to left for periodds. As each period is found, the leftmost part of the symbol up to that period is

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
```

4.4 Constants

The assembler supports four basic types of constants:

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

4.4.1 Integer Constants

The assembler supports integer constants in any base from 2 to 16, the value of which must be expressible in at most 32 binary digits. Integer constants must begin with a decimal digit with the exception of hexadecimal and binary constants which are prefixed with special characters; $ or 0x and % or 2_ respectively. The following sections describe these notations in more detail. The default base is set using the RADIX directive. At the start of assembly RADIX is set to its default value of 10. Unless a constant specifies its own base by means of a prefix, it is taken as a number in the base set by RADIX. 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

4.4.2 Character Constants

A character constant is a string of up to 4 characters enclosed in either single or double quotes (' ' or " "). 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:

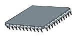
`'a'`	Represented internally as 00000061 hexadecimal
`'abc'`	Represented internally as 00616263 hexadecimal
`'"a'`	Represented internally as 00002261 hexadecimal
`"""a"`	Represented internally as 00002261 hexadecimal
`"'a"`	Represented internally as 00002761 hexadecimal
`'\^M'`	Represented internally 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.

4.4.3 Assembly Time Constants

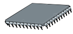
The EQU directive can be used to assign a value to a symbol. The symbol then becomes a constant for the duration of the assembly. The value does not have to be absolute to be used in expressions. If the expression contains forward references then the symbol takes the value of the expression at the end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

Example



68000

```
val     equ     3
        movi    #val, a0
```



SH2

```
val     equ     3
        move    #val,r0
```

4.4.4 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'
MyStringLenequ     *-MyString
```

The @ operator is similar to the ‘*’ operator except that where ‘*’ is the program counter at the start of the statement, @ 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 address.

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      *
```

4.4.5 Pre-defined Constants

There are a number of constants that are pre-defined and updated by the assembler. Some of these are standard internal constants and the rest are mainly to help you keep track of version numbers and dates. Note that pre-defined constants are case insensitive so you can, for example, use either or both of `_radix` and `_RADIX`.

Renaming Pre-defined Constants

See also "ALIAS" on page 5-3.

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 because pre-defined constants are case insensitive you must alias both the upper and lower case name before you can define a symbol with the same name aa a pre-defined constant. For example, you must alias both `_RADIX` and `_radix` before you can define your own symbol using one of these names.

Constant	Description
`*`	Current value of the assembly program counter, evaluated at the beginning of the statement.
`@`	Current value of the assembly program counter, evaluated during the statement.
`__rs`	Current value of the RS counter i.e. the current offset into a structure. Note that this name 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 start of assembly.

Table 4-1. Pre-defined constants

Constant	Description
_month	The month at the start of assembly.
_day	The day at the start of assembly
_weekday	The weekday at the start of assembly.
_hours	The hour at the start of assembly.
_minutes	The minute at the start of assembly.
_seconds	The second at the start of assembly
_filename	The name of the root file (the first source file name).
_current_file	The file being assembled.
_current_line	The current line number within the current file.
_section_name	The name of the current section.
_group_name	The name of the current group.
_snversion	The version of the assembler.
narg	The number of parameters passed to the current macro.
_litflags	A bit field which can be used to determine if there are any pending literals to be output. If bit value four is set there are word literals to be output; if bit value 16 is set there are long values to be output; all other bits are always zero.

Table 4-1. Pre-defined constants

Note The 24 hour time and date constants are set at the beginning of the assembly and do not change. To put the date and time into a string see “Turning Numbers into Strings” on page 4-31.

Example

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

4.4.6 Strings

See also “Macro Parameters” on page 6-5.

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

4.4.7 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'
```

4.5 Expressions

An expression is a sequence of one or more constants, symbols or functions separated by operators. White space (spaces or tabs) is allowed in expressions after mathematical operators regardless of the *White Space* option (`ws+|-`). 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 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
```

4.5.1 Operator Precedence

The operators supported by the assembler are given in Table 4-1 on page 4-35 below. They are listed in decreasing order of precedence; operators of equal precedence are grouped together and evaluated from left to right in that group. 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 appropriately parenthesised.

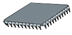
<table>
<tr><th>Operator</th><th>Type</th><th>Usage</th><th>Description</th></tr>
<tr><td>( )</td><td>Primary</td><td>(a)</td><td>Parenthesis brackets</td></tr>
<tr><td>+</td><td>Unary</td><td>+a</td><td>Positive a</td></tr>
<tr><td>-</td><td>Unary</td><td>-a</td><td>Negative a</td></tr>
<tr><td>~</td><td>Unary</td><td>~a</td><td>Bitwise NOT</td></tr>
<tr><td>!</td><td>Unary</td><td>!a</td><td>Logical NOT</td></tr>
<tr><td><<</td><td>Binary</td><td>a<<b</td><td>Shift a left b times</td></tr>
<tr><td>>></td><td>Binary</td><td>a>>b</td><td>Shift a right b times</td></tr>
<tr><td>&</td><td>Binary</td><td>a&b</td><td>Logical AND</td></tr>
<tr><td>| (or !)*</td><td>Binary</td><td>a|b</td><td>Logical OR</td></tr>
<tr><td>^</td><td>Binary</td><td>a^b</td><td>Logical XOR</td></tr>
<tr><td>*</td><td>Binary</td><td>a*b</td><td>Multiply a by b</td></tr>
<tr><td>/</td><td>Binary</td><td>a/b</td><td>Divide a by b giving quotient</td></tr>
<tr><td>%</td><td>Binary</td><td>a%b</td><td>Divide a by b giving modulus</td></tr>
<tr><td>+</td><td>Binary</td><td>a+b</td><td>Increment a by b</td></tr>
<tr><td>-</td><td>Binary</td><td>a-b</td><td>Decrement a by b</td></tr>
<tr><td>=</td><td>Binary</td><td>a=b</td><td>Equate b to a</td></tr>
<tr><td><</td><td>Binary</td><td>a<b</td><td>a is less than b</td></tr>
<tr><td>></td><td>Binary</td><td>a>b</td><td>a is greater than b</td></tr>
<tr><td><=</td><td>Binary</td><td>a<=b</td><td>a is less than or equal to b</td></tr>
<tr><td>>=</td><td>Binary</td><td>a>=b</td><td>a is greater than or equal to b</td></tr>
<tr><td><></td><td>Binary</td><td>a<>b</td><td>a is unequal to b</td></tr>
</table>

Table 4-1. Operator precedence

* The 68000 Quirk *Logical OR* (`or+`) allows the use of ‘!’ as a synonym for ‘|’.

4.5.2 Functions

The assembler provides a set of functions providing useful information about symbols, strings, section and group sizes and start addresses, offsets and others.



68000

ADDRMODE

`addrmode(`*InstructionOperand*`)`

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.The addressing modes used by ADDRMODE are as follows:

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 4-1. Addressing modes used by ADDRMODE.

ALIGNMENT

`alignment(x)`

The ALIGNMENT function returns the offset of its argument from the section's alignment type. The alignment type can be any power of 2 where :

2^0	means Byte aligned
2^1	means Word aligned
2^2	means Long aligned
2^3	means Double Long aligned
...	etc.

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.

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 previously encountered either as a definition or a reference. DEF returns -1 if *Symbol* has been previously defined, 0 otherwise.

FILESIZE

`filesize([~]`*Filename*`)`

The FILESIZE function returns the size of *Filename* or -1 if *Filename* does not exist. If the filename does not specify a path

or ‘~’, 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. If the ~ is specified the assembler will search for files only in the directory in which the assembler executable is located.

Example

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

OBJBASE

`objbase(`*Name*`)`

See also “Group Functions” on page 7-20.

The OBJBASE function returns the logical starting address of the section or group specified by *Name*, evaluated at link time.

ORGBASE

`orgbase(`*Name*`)`

See also “Group Functions” on page 7-20.

The ORGBASE function returns the physical starting address of the section or group specified by *Name*, evaluated at link time.

OBJLIMIT

`objlimit(`*Name*`)`

The OBJLIMIT function returns the last logical address containing data from the section or group specified by *Name*.

ORGLIMIT

`orglimit(`*Name*`)`

The ORGLIMIT function returns the last physical address containing data from the section or group specified by *Name*.

SIZE

`size(`*Name*`)`

The SIZE function returns the current size of the section or group *Name*. It is evaluated immediately and so reflects the current section or group size not the final size.

LINKEDSIZE

`linkedsize(`*Name*`)`

The LINKEDSIZE function returns the final link time size of the section or group specied by *Name*.

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

See also "Extended Parameters " on page 6-17.

`narg(`*List*`)`

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

OFFSET

`offset(`*Expr*`)`

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. REF returns -1 if *Symbol* has been referenced but not defined, 0 otherwise.

SQRT

`sqrt(`*Expr*`)`

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

STRCMP

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

The STRCMP function performs a case sensitive comparison of two strings *String1* and *String2*. STRCMP returns -1 if the comparrison is true, 0 otherwise.

STRICMP

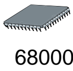
`stricmp(`*String1,String2*`)`

The STRICMP function performs a case insensitive comparison of two strings *String1* and *String2*. STRICMP returns -1 if the comparrison is true, 0 otherwise.

STRLEN

`strlen(`*String*`)`

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



68000

TYPE

`type(`*Symbol*`)`

The TYPE function returns the type of a symbol. If the argument isn't the name of a previously encountered 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 4-1. 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 this page contains.

5 Assembler Directives

5.1 Overview

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.

About This Chapter

The topics covered in this chapter 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

5.2 Changing Directive Names

5.2.1 ALIAS

See also "DISABLE" on page 5-3.

Use the ALIAS directive to rename 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. 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 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
```

5.2.2 DISABLE

See also "ALIAS" on page 5-3.

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 to rename something rather than just providing an alias; care should be taken to alias the name before disabling it if that functionality is needed later.

Syntax

`disable` *OldName*

where:

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

Example

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

5.3 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. 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.

Note that expressions in a SET directive must evaluate immediately whilst expressions in other types of equates must evaluate at the end of assembly.

5.3.1 EQU (.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.

Syntax

Symbol `equ` *Expression*

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.

The .EQU extension is provided via the standard include file SNASMSH2.MAC and can be changed by editing that file.

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 The SNASM assemblers, unlike most assemblers, allow 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
```

5.3.2 SET (.ASSIGN)

See “Labels and Symbols” on page 4-17 See “Sections and Groups” on page 7-1

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*

Label `.assign` *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.

The .ASSIGN extension is provided via the standard include file SNASMSH2.MAC and can be changed by editing that file.

Example

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

5.3.3 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.

Symbols equated with the EQUS directive can be used anywhere in your code so to inform the assembler that it is about to encounter a string equate they must, in general, be preceded by a backslash ('\') . If there could be confusion as to where the parameter ends, use a second backslash. There are two related situations in which the requirement for a preceeding backslash is relaxed, these relate to expressions and addressing modes. In expressions 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 produce an error message. Also, if a string variable is at the start of an address mode in a string equate, it will be substituted for without the need for a backslash.

Syntax

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

where:

Symbol	is the symbol defined by this statement.
String	is a string enclosed in quotes.
EqusName	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
```

Example 3

```
V1         equs    offset(a0)
           ...
           move.l  V1,d1
           move.l  d1,V1
```

5.3.4 RS Equates

RS equates are used primarily for global variables and data structures. They enable lists of constant labels to be defined without using explicit numbers that would require changing if an item was added or deleted near the front of a list. So, in a data structure a set of labels can be defined as offsets without having to explicitly keep track of the offsets. The RS, RSRESET, RSSET directives and the __RS internal variable provide this facility.

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 when starting 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.

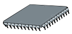
Example 1

```
                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.
;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. RSRESET -8 could have been used 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
```



68000

Example 3

This example uses a data structure consisting of a long word, a byte and another long word in that order. To make the code more readable and easier to update should the structure change, 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)
```

SH2

Example 3

```
            rsreset
Next        rs.l      1
Flag        rs.b      1
Where       rs.l      1
```

and access the code with lines like

```
            move.l    next(r4),r1
            move.l    where(r4),r2
            tst.b     flag(r4),r0
            cmp/eq    r0,r0
```

5.3.5 Register Equates

Register equates are used to define synonyms for registers to improve code readability. The assembler supports two directives for register equates; EQUR (.REG) and REG.

EQUR (.REG)

The EQUR directive is used to define a symbolic name (that may include periods) for a data or address register.

Syntax

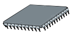
Symbol `equr` *Register*

Symbol `.reg` *Register*

where:

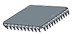
Symbol	is any symbol defined by this statement.
Register	is any data or address register.

The .REG extension is provided via the standard include file SNASMSH2.MAC and can be changed by editing that file.


68000

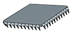
Example 1

```
                section   Code

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


SH2

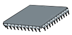
Example 1

```
                section   Code

City            equr      r0
Street          equr      r1
Dude            equr      r2
                ...
From            equr      r0
To              equr      r1
Counter         equr      r2
Player          equr      r4
```


68000

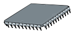
Example 2

```
                move.wd0,Offs(a3,d2.w)
; Could be written as
Power           equr  d0
CarDataPtr      equr  a3
CurIndex        equr  d2
                ...
                move.w
Power,Offs(CarDataPtr,CurIndex.w)
```


SH2

Example 2

```
                mov.w  r0,Offs(r2)
; Could be written as
Power           equr  r0
CarDataPtr      equr  r2
                ...
                mov.w  Power,Offs(CarDataPtr)
```



68000

REG

Tthe REG directive is similar to EQUR but 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.

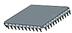
Syntax

Symbol `reg` *RegisterList*

where:

Symbol is a symbol defined by this statement.

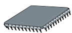
RegisterList is a list of register names or symbols. Different types of register are separated by a forward slash
(‘/’). A range of registers can be specified by separating names with a dash (‘-’). The register at the start of the range must be less than the register at the end of the range.



68000

Example 1

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



68000

Example 2

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

5.4 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. These directives are now described in more detail. Uninitialised data is defined using the DS directive.

5.4.1 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. Unless the *truncate* option is enabled (`t+`), 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+`), 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
LineLength  dc.w    0
PointerAddr 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
```

5.4.2 DCB

The DCB directive defines a constant block of memory. DCB takes two parameters, *Count* and *Value*; *Count* specifies how many times *Value* is to be repeated. *Count* must always be present and evaluate but *Value* is optional; if *Value* is not specified then the default value of zero is used. If *Count* is followed by a comma but no *Value*, the assembler will generate an error. Unless the *truncate* option is enabled (`t+`), 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 (`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 *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

See also "Options and 68000 Optimisations" on page 5-62.

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 parameters for DC, DCB, DZ and DW only if the *truncate* option is enabled (`t+`).

Example

The following examples will generate error if 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
```

5.4.3 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.

See also "INCBIN" on page 5-37.

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 to include the data.

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
```

5.4.4 DATASIZE

See "DATA" on page 5-22

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
...
```

5.4.5 DATA

See "DATASIZE" on page 5-21

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

5.4.6 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
```

5.4.7 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.

5.5 Changing The Program Counter

5.5.1 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.

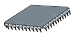
If you are assembling to a target with a supporting operating system you can use ORG at the beginning of a section or program to specify the amount of RAM required. This is useful when developing for a machine with the operating system present. The parameter starts with a ‘?’ character followed by the 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*

where:

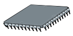
Address An expression specifying the program starting address in the target memory. The expression must evaluate.



68000

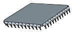
Example 1

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



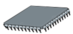
Example 1

```
            org     $400
Start1
            mov     #MySack,sp
            ...
```



Example 2

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



Example 2

```
                org     ?512*1024    ;ask for
512K
                mova    @VarBase,PC),r0
                ...
```

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
```

5.5.2 EVEN

See also "CNOP" on page 5-28.

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, EVEN cannot be used in a byte aligned section (unless it is ORG'd to a known address); the section has to be at least word-aligned.

Syntax

```
even
```

Example 1

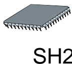
The EVEN directive is equivalent to:

```
        cnop    0,2
```

Example 2

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

5.5.3 .ALIGN


SH2

The .ALIGN directive aligns the PC to the specified boundary, filling any intervening space with NOOP instructions.

Syntax

`.align` *Boundary*

where:

Boundary specifies the boundary as word, long or double long where *Boundary* can be one of:

- 2 Word boundary
- 4 Long boundary
- 8 Double Long boundary

Example

The .ALIGN directive is equivalent to:

```
        cnop   0,Boundary,$00090009
```

5.5.4 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.

See also "Sections and Groups" on page 7-1.

When using sections it is not possible to align the program counter to a larger boundary than the alignment of the current section unless the section has a base address set e.g. CNOP cannot be used to align the program counter to 4 bytes in a word (2 byte) aligned section. The assembler will generate a warning if this is attempted.

Syntax

`cnop` *Offset*, *Boundary*[,*Pad*]

where:

Offset specifies the offset in bytes. *Offset* is an expression that must evaluate i.e. not reference any external or undefined symbols.

Boundary specifies the boundary in bytes from which the offset is added. *Boundary* is an expression that must evaluate i.e. not reference any external or undefined symbols.

Pad specifies the value to fill memory locations with between the address given by the current value of the program counter and the address resulting from the CNOP. *Pad* is an expression that must evaluate i.e. not reference any external or undefined symbols.

If the adjustment to the PC is greater than four bytes then *Pad* is treated as a 32-bit value. The ajustment may not fit exactly into a 32-bit multiple. If the remaining adjustment is greater than two bytes, the value of *Pad* is AND'ed with $FFFF. If the remianing adjustment is a single byte, the least significant byte of *Pad* is used.

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
```

5.5.5 OBJ and OBJEND

The OBJ and OBJEND directives are used to implement assembly with offset. The OBJ directive specifes the offset address and OBJEND terminates the prevous OBJ. The code between OBJ and OBJEND is assembled as if it was at the address specified by OBJ.

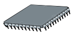
OBJ and OBJEND cannot be nested and must always be correctly balanced. Nested or unbalanced OBJ and OBJEND constructs will cause the assembler to loose track of where the PC is pointing.

Syntax

```
obj Offset
...
objend
```

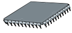
where:

Offset specifies the offset in bytes. *Offset* is an expression that must evaluate i.e. not reference any external or undefined symbols.


68000

Example

```
             org      $8000
RunAddr      equ      $400
             lea      RelocCode,a0
             lea      RunAddr,a1
             move.w   #(RelocEnd-RelocCode)/2-1,d0
@Loop        move.w   (a0)+,(a1)+
             dbrad    0,@Loop
             jmp      RunAddr

RelocCode
             obj      RunAddr
; The following code, up to but not including
; the OBJEND directive, will be set to run at
RunAddr
             jmp      Startup
             ...
Startup      move.w   #$2700,sr
             ...
             objend
RelocEnd
```



SH2

Example

```
                org     $8000
RunAddr         equ     $400
                mov     #RelocCode,r0
                mov     #RunAddr,r1
                move.w  #(RelocEnd-RelocCode)/2-
1,r2
@Loop
                move.w  (r0)+,r3
                move.w  r3,(r1)
                add     #2,r1
                dt      r2
                bf      @Loop
                bra     RunAddr
                nop

RelocCode
                obj     RunAddr
; The following code, up to but not including
; the OBJEND directive, will be set to run at
RunAddr
                jmp     Startup
                ...
Startup

; Disable divide unit overflows
                mov     #div_base,r0
                mov     #0,r1        ;ovfie=0,ovf=0
                mov.l   r1,@(div_cont,r0)
                ...
                objend
RelocEnd
```

5.6 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 SNASM 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 malls* (`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.

5.6.1 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 a listing file is specified then the internal listing state will be set to zero i.e. listing will be turned on at the start of the program. LIST+ and LIST- can then be used 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 is 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+
           ...
```

5.7 Including Other Files

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

5.7.1 The Standard INCLUDE File

The assembler will automatically include a standard include file if it is present in the same directory as the assembler. This file contains, among other things, extensions to the standard SNASM2 directive names and can be edited using a text editor. The file name is of the form SNASM*xxx*.MAC where *xxx* is the last 3 characters of the assembler executable name. The assembler treats this file as the first include in the assembly. If such a file is not present it is not included and no error is generated.

5.7.2 INCLUDE

Normally there will be 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

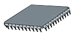
`include` [?,][~]*Filename*[,*Type*]

where:

`?`	causes the include to be conditional upon the existence of *Filename*. Using INCLUDE without the `?` parameter causes the assembler to generate an error if *Filename* cannot be found. Using INCLUDE with the `?` parameter causes the assembler to treat INCLUDE as a NOOP if *Filename* cannot be found.
~	Sets the search path to the directory from which the assembler was invoked.
Filename	is the name of the source code file to include.
Type	is an optional qualifier that can be:

`68k|sh2|asm|c`	Source file
`cof|o|obj`	Object file
`lib`	Library file
`bin`	Binary file

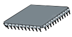
Without a *Type* qualifier the file type is taken from the filename extension or treated as a source file if no extension is specified. The *Type* qualifier can be used to override the default associations with filename extensions or to use a non-standard extension.



68000

Example

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



Example

```
StartUpCode    bra        MainEntry
               nop
               include    'equs.asm'
               include    c:\general\maths.asm
MainEntry      mov        #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.

5.7.3 INCBIN

The INCBIN directive enables binary data such as graphics or music to be included in a program. The assembler does not know about the internal structure of data stored in binary format so the data must have a label on it and offsets into it handled manually.

See also "FILESIZE" on page 4-42.

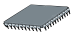
To determine the size of a binary file before it is included, use the FILESIZE function which returns the size of a file in bytes or -1 if the file cannot be found.

Syntax

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

where:

`?`	causes the include to be conditional upon the existence of *Filename*. Using INCLUDE without the `?` parameter causes the assembler to generate an error if *Filename* cannot be found. Using INCLUDE with the `?` parameter causes the assembler to treat INCLUDE as a NOOP if *Filename* cannot be found.
~	Sets the search path to the directory from which the assembler was invoked.
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.



68000

Example 1

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

SH2

Example 1

```
                mov       #SineTable,r0
                add.w     r1,r1
                add.w     r1,r0
; 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
```

5.8 Setting Target Parameters

5.8.1 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.

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

Syntax

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

where:

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 — is 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
            ...
```

5.9 Conditional Assembly

See also "Conditional Assembly (IFxx) Macros" on page 6-16.

Conditional assembly structures enable the behaviour of an assembly to be modified 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. In a CASE statement however, placing a label on the '=' case selector character, will cause it to be interpreted as a SET directive.

5.9.1 END (.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*]

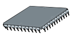
where:

Expression optionally specifies the execution address of the program.

The .END extension is provided via the standard include file SNASMSH2.MAC and can be changed by editing that file.

Example 1

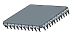
```
            org     $200
            Main body of code
            end
```



68000

Example 2

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



SH2

Example 2

```
StartUp     mov         #Mystack.sp
            ...
            bra         MainLoop
            nop
            end         StartUp     ;Start at
StartUp
```

5.9.2 IF...ELSE...ELSEIF and ENDIF

See also "CASE... ENDCASE" on page 5-46.

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 up to a ELSE, ELSEIF, or ENDIF is skipped. 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 5-47

```
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 care should be taken to parenthesise the expression correctly. In the following examples to branch on the language not being German use

```
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... ENDCASE", Example 2 on page 5-48

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

```
Englsh      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"
            endif
```

5.9.3 CASE... ENDCASE

See also IF... ELSEIF... ELSE... ENDIF on page 5-43

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. The '=?' must be the last choice in the list or the assembler will generate a warning.

Syntax

```
caseExpression

[=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 5-45

```
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 5-46

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
```

5.9.4 REPT... 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 so 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 4-30 and "ALIAS" on page 5-3

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.

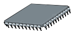
Syntax

```
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



68000

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

SH2

Example 1

```
rept    16
  move.w r1,-(r0)
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
```

5.9.5 WHILE... ENDW

See also "DO... UNTIL" on page 5-53.

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 end 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.

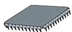
Syntax

```
while    LoopCondition
   LoopBody
endw
```

where:

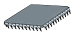
LoopBody	is a block of code. This can have nested control constructs as long as they are balanced.
LoopCondition	is an expression which must evaluate to a value.

Example



68000

```
Factor    equ      4

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



SH2

Example

```
Factor    equu         4

; Build the code required to multiply by factor
Temp      =        Factor
          mov.l    (r0),r1
          while    Temp>1
           rotl    r1
Temp       =       Temp>>1
          endw
           mov.l   r1,(r0)
```

5.9.6 DO... UNTIL

See also "WHILE... ENDW" on page 5-51

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 4-30 and "ALIAS" on page 5-3

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.

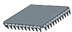
Syntax

```
do
   LoopBody
until    LoopCondition
```

where:

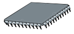
LoopBody	is a block of code. This can have nested control constructs as long as they are balanced.
LoopCondition	is an expression which must evaluate to a value.

Example



68000

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


SH2

Example

```
Factor    equu    4
; Build the code required to multiply by factor
          mov.l  (r0),r1
Temp      =      Factor
          do
           rotl   r1
Temp       =      Temp>>1
          until   Temp<=1
          mov.l  r1,(r0)
```

5.10 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. Note that characters within a string are numbered from 1.

5.10.1 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’
```

5.10.2 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
```

5.10.3 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 then these functions return the position of the first character of the sub-string, otherwise they return zero. These functions have an optional parameter which specifies the start position (in the string) of the search. Note that the sub-string is always reported relative to the start of the string and not the start of the search.

Example

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

5.10.4 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   equus       '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?'
```

5.11 Modules

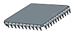
Modules are self-contained sections of code, delimited using the MODULE and MODEND directives. Modules are used to control the scope of local labels and are strongly recommended as they rigidly define where local labels can and cannot be referenced.

5.11.1 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 this form can still be freely used outside of modules. Modules can be nested but scoping is not. i.e. the only local labels available inside a module are those defined within it.

Syntax

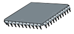
```
module
...
modend
```



68000

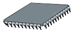
Example 1

```
ClearData    module

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


SH2

Example 1

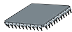
```
ClearData    module

@Loop        move.w   r0,r2
             bsr      @ClearIt
             dbra     d1,@Loop
             nop
             dt       r1
             bf       @Loop
             rts
             nop

@ClearIt     module

             mov      #0,r0
@Loop        mov.b    r0,(a1)
             add      #1,a1
             dt       r2
             bf       @Loop
             rts
             nop

             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
```


68000

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.

```
; The following code will only work if it is all
; enclosed in a module.

@Loop         movem.w   d7,-(sp)
              ...
@SubModule    module
              ...
              modend
              movem.w   (sp)+,d7
              dbra      d7,@Loop
```


SH2

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.

```
; The following code will only work if it is all
; enclosed in a module.

@Loop         move.w    r7,-(sp)
              ...
@SubModule    module
              ...
              modend
              move.w    (sp)+,r7
              dt        r7
              bf        @Loop
```

5.12 Options and 68000 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 source code using the OPT, PUSHO and POPO directives. The options and optimisations are briefly reviewed here.

5.12.1 Options

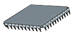
Options provide control the behaviour of the assembler and how it outputs information to the screen, listing and object file. They can be set from the command-line or from within your source using the OPT directive. However, it is recommended that production code always sets options from within the source 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 all code bytes 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.* By default 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 defined outside a module.

Option	Default	Description
g-/+	Off	*GNU mode*. Toggles between standard (g-) and GNU C (g+) interpretations of the PC Relative with Displacement (disp:8,PC) addressing mode. In standard mode the 'disp:8' expression is interpreted as already PC relative. In GNU mode the 'disp:8' expression is modified by the PC value. See also the `pcrel` option below.
`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.* Unless specified otherwise using the `l` option, 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.
`pcrel-/+`	Off	*PC Relative Syntax*. Enable GNU interpretation of PC Relative with Displacement address modes only. See also the *GNU Mode* option above.

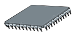
Option	Default	Description
`s-/+`	Off	*Equated Symbols as Labels*. Treat equated symbols as labels.
`t-/+`	Off	*Truncate*. Truncates out of range parameters for DC, DCB, and DW. If this option is not enabled, out of range parameters will generate an error.
`v-/+`	Off	Writes local labels to the symbol tableand puts them in the COFF and MAP files.
`w+/-`	On	*Suppress 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 whitespace 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.



68000

Example

```
        opt     c-              ;Case insensitive
Fred    dc.w    5

        move    #FRED,d0
        add     #fred,d0

        opt     c+              ;Case sensitive

        move    #FRED,d0        ;Error
        add     #fred,d0        ;Error
        sub     #Fred,d0        ;OK
```



SH2

Example

```
        opt     c-              ;Case insensitive
Fred    dc.w    5

        move    #FRED,r0
        move    #fred,r0

        opt     c+              ;Case sensitive

        move    #FRED,r0        ;Error
        move    #fred,r0        ;Error
        move    #Fred,r0        ;OK
```

5.12.2 68000 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 Table 5-1 on page 5-67. They can be set from the command-line or from within source code 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.
`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 5-1. Assembler command-line optimisations.

5.12.3 OPT

Use OPT to set the assembler options and optimisations for the subsequent source.

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* option.
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.
v	*Write local labels to COFF file* option.
w	*Print warnings* option
ws	*Allow white space* option.
x	*External symbols.*

* 68000 only

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+
```

5.12.4 PUSHO and POPO

The PUSHO and POPO directives can be used to temporarily change options and optimisations during assembly. This enables the assembler to be invoked using the normal settings, one or more settings to be changed at some point in the source code and then changed 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
```

5.13 Custom Errors and Warnings

The assembler provies the ability to generate your custom 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.

5.13.1 INFORM

Use the INFORM directive to generate errors of varying severity and display a message explaining the error in detail. The directive takes two or more 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 hex value of the operand.
`%s` Substitutes the string value of the operand.

Operand An optional parameter which can be an expression or a string.

Example 1

```
StartSlowRAMequ   *
              ...
SlowRAMSize equ   *-StartSlowRAM
              ...
              inform0,"Slow RAM size:
$%h",SlowRAMSize
```

Example 2

```
StrucBegdc.w    0
        ...
StrucEnd
StrucLenequ     StrucEnd-StrucBeg
        if      StrucLen>1024
         inform 0,‘Beg=%h
End=%h’,StrucBeg,StrucLen
         inform 2,‘Structure too long’
        endif
```

5.13.2 FAIL

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

```
         inform 3,’Assembly failed’
```

5.14 Linking

Linking enables programs to be written in separate parts and subsequently combined to be sent 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 a single step 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 first use the EXPORT directive to declare the symbol as external in the component that the symbol was defined. Then, in the component that will use the symbol, use the IMPORT directive to declare that the symbol has been defined in another component. 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 7-1

5.14.1 EXPORT (.EXPORT)

The EXPORT directive enables symbols defined in the current file to be 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.

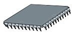
Syntax

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

where:

Label is any symbol defined by this statement.

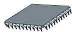
The .EXPORT extension is provided via the standard include file SNASMSH2.MAC and can be changed by editing that file.



68000

Example

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



SH2

Example

```
            import.w  Table
            export    Routine1,Routine2
Routine1    mov       #Table,r0
            ...
Routine2    add       r0,r0
```

5.14.2 IMPORT (.IMPORT)

The IMPORT directive provides the ability to reference 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:

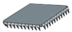
Label is any symbol previously defined in another source code module.

Qualifier is an optional qualifier that can be:

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

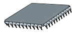
The .IMPORT extension is provided via the standard include file SNASMSH2.MAC and can be changed by editing that file.

Example



68000

```
        export      Table
        import      Routine1,Routine2

        section     Tables,BssGroup
Table   ds.w        100

        section     Code,Text
        jsr         Routine1
        jsr         Routine2
```



Example

```
             export     Table
             import     Routine1,Routine2

             section    Tables,BssGroup
Table        ds.w       100

             section    Code,Text
             bsr        Routine1
             nop
             bsr        Routine2
             nop
```

5.14.3 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`	Sets PUBLIC active.
`off`	Sets PUBLIC inactive.
Flag	is 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
```

5.14.4 GLOBAL (.GLOBAL)

Sometimes it may be unclear whether symbol needs to be declared as external or if a symbol to be referenced has been declared as external in another program component. In these cases the GLOBAL directive enables you to substitute for EXPORT or IMPORT where it is not clear which is required. If the symbol is eventually defined an EXPORT will be performed, otherwise an IMPORT will be assumed.

Syntax

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

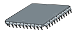
where:

Symbol is a symbol defined by this statement.

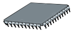
The .GLOBAL extension is provided via the standard include file SNASMSH2.MAC and can be changed by editing that file.

5.14.5 Introduction to Linking

This section introduces linking using SNASM2 by providing a step-by-step guide to converting a single source file into two linkable files. This conversion 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.

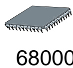



```
                group     g1,org $100
                group     g2,org $1000

                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:
```



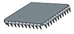

```
                group     g1,org $100
                group     g2,org $1000

                section   s1,g1
start:
                bsr       func1
                nop
                bsr       func2
                nop
                bsr       func3
                nop
                bsr       func4
                nop
func1
                move      #1,r0
                add       #4,r0
                rts
                nop

                section   s3,g2
func2
                move      #2,r0
                rts
                add       #4,r0        ; Delay slot

                section   s3,g1
func3
                move      #3,r0
                add       #4,r0
                bra       func1
                nop
                rts
                nop

                section   s4,g2
func4
                move      #4,r0
                add       #4,r0
                bra       func2
                nop
                rts
                nop

                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.ASM and TEST2.ASM for example, which could subsequently be linked together to achieve the same effect as the code in the original single file. TEST1.ASM would be as follows:



68000

```
          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
```


SH2

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

        section   s1
start:

        bsr       func1
        nop
        bsr       func2
        nop
        bsr       func3
        nop
        bsr       func4
        nop
func1
        move      #1,r0
        add       #4,r0
        rts
        nop

        section   s3
func2
        move      #2,r0
        add       #4,r0
        rts
        nop
```

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

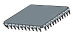
Note that the GLOBAL directive could have been used in place of IMPORT and EXPORT as shown below.

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

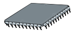
Hence, on finding that FUNC3 and FUNC4 had not been defined TEST1.ASM, the first GLOBAL directive would behave in the same way as the IMPORT directive. Similarly, as START, FUNC1 and FUNC2 are defined in TEST1.ASM the second GLOBAL directive would behave as it were an EXPORT directive.

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



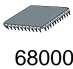

```
                export    func3,func4
                import    func1,func2
                section   s3
func3
                move.l    #3,d0
                addq      #4,d0
                bra       func1
                rts

                section   s4

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

                ds.b      $1000
stack:
```


SH2

```
              export    func3,func4
              import    func1,func2
              section   s3
func3
              move      #3,r0
              add       #4,r0
              bra       func1
              nop
              rts
              nop

              section   s4

func4
              move      #4,r0
              add       #4,r0
              bra       func2
              nop
              rts
              nop

              ds.b      $1000
stack:
```

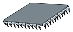
The two source files TEST1.ASM and TEST2.ASM now require assembling prior to linking. Although the linker is incorporated into the assembler, assembly and linking of the same source file cannot be performed at the same. This is because it would create a situation where the assembler would attempt to 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.



68000

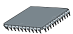
```
snasm68k /l test1.asm,test1
```



SH2

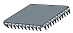
```
snasmsh2 /l test1.asm,test1
```

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



68000

```
snasm68k /l test1.asm+test2.asm,test
```



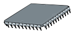
SH2

```
snasmsh2 /l test1.sh2+test2.sh2,test
```

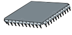
See also "SNMAKE" on page 9-1.

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,t4:
```



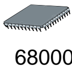

```
snasmsh2 /l test1.cof+test2.cof,t1:
```

This will link TEST1.COF and TEST2.COF together and download them to the appropriate target. 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.ASM and TEST2.ASM 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.ASM, shown below, includes the two object files TEST1.COF and TEST2.COF and orders groups and sections within groups.

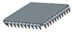
```
group     g1,org $8000
group     g2,org $9000

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:



68000

```
snasm68k test.asm,t4:test
```



SH2

```
snasmksh2 test.asm,t1:test
```

This example assembles TEST.ASM, 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.

5.14.6 The Command File

See also "Assembler Command Files" on page 3-14.

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 the appropriate target.

```
test.asm,t4
```

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
```

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.

6 Macros

Macros provide the ability 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 required and parameters passed to them, simplifying programming and improving code readability.

The assembler enables you to:

- 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

6.1 Introducing Macros

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

6.1.1 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 existing copy being automatically removed from memory. Explicitly removing macros from memory can be performed using the PURGE directive discussed later.

See also "Symbols and Periods" on page 4-20.

The macro name is defined in the label field of MACRO. The name can include periods but this is not recommended. 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, attempting to define a macro with the name of a current directive or processor instruction will cause the assembler to generate a warning stating that the macro cannot be called. This situation can be avoided by using ALIAS to alias the a directive or instruction, DISABLE to remove the old name and then defining 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 macros can be nested 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. See “Macro Parameters” on page 6-5.
MacroBody	is a block of code which can include nested macros and balanced control constructs. If a structure is initiated in a macro it must be terminated before the ENDM directive. Similarly, a structure that was not initiated within a particular macro cannot be terminated from within that macro. In both cases the assembler will generate an error.

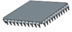
6.1.2 Invoking a Macro

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 consisting of a period followed immediately by 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.

6.1.3 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 macro call. MEXIT is supported for compatibility only as the conditional assembly macros render this directive redundant. If MEXIT *i*s

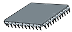
used, care should be taken when using MEXIT as it creates multiple exit points from the macro. Note that whilst both MEXIT and ENDM terminate a macro *expansion* only ENDM terminates a macro *definition*.



68000

Example 1

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



SH2

Example 1

```
BIOSCall    macro
            move.w   #\1,r0
            if       narg=2
             mov.l   #\2,r0
            endif
            jsr      _CDBIOS
            nop
            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 be better 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
```

6.2 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.

6.2.1 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
; joe 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
```

6.2.2 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
```

6.2.3 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

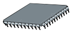
```
shift
```

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
```

6.2.4 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.

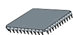


68000

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 ;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
```


SH2

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 ;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
          ...
          mov     #VarBase(pc),r6
          move.w  L1(r6),r0
```

6.2.5 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

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

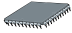
The '_' parameter returns a string containing the entire parameter string from the remainder of the macro invocation line up to but not including the end of line or a comment. This feature enables a macro to interpret its invocation line which is useful when you are invoking a macro from within a macro.

The \@ Parameter

The '\@' parameter construct is used to generate unique labels each time a macro is called. '\@' expands to a character string of the form 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

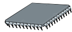
```
        inform  0,"Assembling \_filename"
```



68000

Example 2

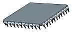
```
Inc     macro           ; increment register
        addq.\0    #1,\1
        endm
        ...
        Inc        d0 ; Expands to addq. #1,d0
        Inc.b      d1 ; Expands to addq.b #1,d1
        Inc.w      d0 ; Expands to addq.w #1,d0
        Inc.l      d7 ; Expands to addq.l #1,d7
```



SH2

Example 2

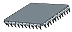
```
Inc     macro           ; increment register
        mov        #\1,r0
        mov.\0     (r0),r1
        add        #1,r0
        mov.\0     r1,(r0)
        endm
        ...
        Inc        addr0
        Inc.b      addr1
        Inc.w      addr0
        Inc.l      addr7

addr1   dc.l       0
addr2   dc.b       0
addr3   dc.w       0
addr4   dc.l       0
```



68000

Example 3

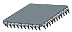
```
BraNz   macro           ; branch if register not
zero
        tst.w    \1
        bne.\0   \2
        endm
        ...
        BraNz    d0,Exit
        BraNz.s  d7,Again
```



SH2

Example 3

```
BraNz   macro           ; branch if register not
zero
        mov      \1,r0
        cmp/eq   #0,r0
        bf       \2
        endm
        ...
        BraNz    r1,Exit
        BraNz.s  r7,Again
```

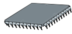


68000

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
        move.w   \1,\2
Loop\@  dbra     \2,Loop\@
        endm
        ...
        Delay    #3,d0
```


SH2

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
        move     \1,\2
        dt       \2
Loop\@  bf/s     Loop\@
        dt       \2
        endm
        ...
        Delay    #3,r0
```

6.3 Short Macros

See also "Conditional Assembly (IFxx) Macros" on page 6-16.

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 a macro may be required to imitate a control structure.

6.3.1 MACROS

The MACROS directive is used to define a short macro.

Syntax

```
macros
```
MacroLine

where:

MacroLine is a line of code.

Note

Short macros 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
```

6.3.2 Conditional Assembly (IFxx) Macros

See also "Switches" on page 3-8.

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 macro` `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 6-1. Conditional assembly macros.

6.4 Advanced Macro Features

6.4.1 Extended Parameters

The assembler provides the ability 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   equu      0
```

```
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

PolyListmacro

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
```

Example Continued

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

For compatibility with other assemblers it is possible to pass a list of parameters enclosed in angle brackets (‘<’ and ‘>’) instead of curly brackets. This requires code to be written 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'>
```

6.4.2 Local Labels in Macros

Local labels and modules can be used in macros. The scope of local labels defined in modules is not affected by nested macros. This means that when invoking a macro in a module you can reference any of the local labels defined in that module when the macro is expanded. To provide a macro with its own local labels you can use a module in the macro, use the '\@' parameter or use the LOCAL directive.

LOCAL

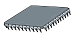
Use the LOCAL directive to declare local labels in macros. Labels declared in this way are defined to be local to the outermost macro nesting level, which for non-nested macros is the macro in which the label was defined. The LOCAL directive does not type the symbols it defines; they can be used as labels,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

Syntax

`local` [*Label*]

where:

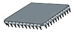
Label is any symbol defined by this statement.



68000

Example 1

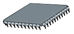
```
Delay       macro
            local     Loop
            move.w    \1,\2
Loop        dbra\     2,Loop
            endm
```



SH2

Example 1

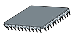
```
Delay       macro
            local       Loop
            move.w      \1,\2
            dt          \2
Loop        bf/s\       Loop
            dt          \2
            endm
```



68000

Example 2

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



SH2

Example 2

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

6.4.3 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 there are no restrictions 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
```

6.4.4 PURGE

The assembler stores macros very efficiently but they can still consume valuable memory. If amacro 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 used to store it.

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 existing 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, care should be taken 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.

7 Sections and Groups

7.1 Overview

In simple terms, sections provide a mechanism to structure programs into logical blocks of code known as sections. Groups provide a way of organising those sections in memory. Sections therefore encourage modular programming and 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 provide control over where a group of sections reside in memory.

The assembler provides a range of directives and functions to support the creation and manipulation of sections and groups. 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

7.2 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 necessarily 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 code into logical blocks, usually corresponding to the categories described above, although this scheme does not have to be strictly adhered to. There can be as many executable, initialised uninitialised sections as necessary.

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 to 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 7-1 below illustrates the relationship between an object file and a hypothetical target’s memory map.

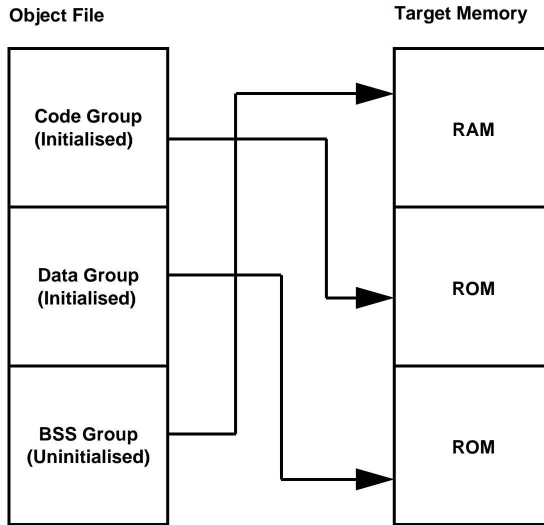


Figure 7-1. Partitioning target memory into logical blocks

7.2.1 Section and Group Directives

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

SECTION	This directive: defines a named section into which code and or data can be placed; enabled chainging betwen different sections.
GROUP	This directive defines a group and its various attributes.
PUSHS and POPS	These directives enable change between sections.

7.2.2 Section and Group Functions

The assembler provides eight functions to manipulate sections and groups, summarised below.

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.
OBJBASE(*Name*)	This function returns the logical starting address of the section or group specified by *Name*.
ORGBASE(*Name*)	This function returns the phyyscial stating base of the section of group specified by *Name*.
OBJLIMIT(*Name*)	This function returns the last logical address containing data from the section or group specified by *Name*.
ORGLIMIT(*Name*)	This function returns the last physical address containing data from the section of group spcified by *Name*.
SIZE(*Name*)	This function returns the current size of the section or group spcified by *Name*.

7.3 Sections

7.3.1 SECTION

The SECTION directive defines (opens for the first time) a new section or re-opens an existing one. When defining a new section, the name of the section and, optionally, 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.

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 7-15 See also “Section Attributes” on page 7-8. for more information on setting section attributes.

Example

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

Note The assembler provides 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.

7.3.2 Section Names

Each section must be defined with a section name. Once a section has been defined it can be re-opened as many times as required. Each time a section is opened any code following the SECTION directive will be concatenated to the end of that section.

A section may also be defined or re-opended with a group name; this allocates the section to a group. It is recommended that a section is defined with a group name. Note that a section should be allocatd to a group only once; if the code is to be linked then the allocation of section fragments to groups should be done at that time.

A section does not have to be defined with a group name. A section can be allocated to a group at any time by re-opening the section with a group name but this is not recommended. See also “Allocating Sections to Groups” on page 7-8.

Example

```
          section       Tables,BssGroup
; Tables section defined and allocated to
; BssGroup group

          section       Data,LowGroup
; Data section defined and allocated to LowGroup group

          section       Tables
; Tables section re-opened and concatonated to BssGroup
; group
```

7.3.3 Section Alignments

Sections can be opened with an optional alignment. If a section is defined without an alignment the assembler uses the default alignment for that processor. See also “SECTION Syntax”. If no alignment is specified the assembler treats the section as long aligned i.e. the same as opening the section with SECTION.L.

If a section is defined with a particular alignment and later re-opened with a different alignment then the section is treated as being aligned to the largest size. (The alignment of a group is taken to be the alignment of the largest section within that group.)

If a section is opened with a specified alignment then subsequent code or data will be aligned accordingly. If a section is re-opened without a specified alignment then subsequent code or data is placed in that section immediately after any previously emitted bytes.

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
```

7.3.4 Allocating Sections to Groups

A section is usually defined with a specified section name and group; this allocates the section to a group. If the code is to be linked then this should be the only time the section is opened with a group name. If the code is not going to be linked then a section should specify a group name each time it is re-opened.

Allocating a section to a group causes it to 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 that precedes all other groups. The section can subsequently be allocated to a group at a later time. If the section remains unallocated and the code is not going to be linked then the unallocated sections remain in the unnamed group; this unnamed group will be output first when asembling.

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
```

7.3.5 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 group 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 commas.

Example

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

7.3.6 Changing Sections

The PUSHS and POPS directives are used to change sections, working on a stack basis. The PUSHS directive saves the current section; the POPS directive restores the section previously saved by the last PUSHS. The PUSHS and POPS directives do not have to be balanced i.e there is no limit to the number of PUSHS's that may be outstanding.

Syntax

```
pushs
...
pops
```

Example 1

```
                  section       Data
LevelText         dc.b          0,11,19
                  ...
                  pushs
; Push current section (Data) onto stack

                  section       FastRAM
; Open
PicTexture        ds.w          1
Done1String       ds.w          100
Done1String       ds.w          100
                  pops
; Pop FastRAM section from stack and return to
; previously pushed section (Data)

; This code is in section Data:
                  dc.b          'Aggressive',0
                  dc.b          'Neutral',0
                  dc.b          'Passive',0
                  even
                  pops
```

Example 2

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
```

7.3.7 Section Functions

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 sections are assigned memory addresses.

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. To determine the offset of a symbol into a fragment place a label at the beginning of the fragment and then perform the subtraction manually. 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. The alignment type can be any power of two where :

2^0	means Byte aligned
2^1	means Word aligned
2^2	means Long aligned
2^3	means Double Long aligned
...	etc.

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.

Example

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

OBJBASE(*SectionName*)

The OBJBASE function returns the logical starting address of the section specified by *SectionName*, evaluated at link time.

ORGBASE(*SectionName*)

The ORGBASE function returns the physical starting address of the section specified by *SectionName*, evaluated at link time.

OBJLIMIT(*SectionName*)

The OBJLIMIT function returns the last logical address containing data from the section specified by *SectionName*.

ORGLIMIT(*SectionName*)

The ORGLIMIT function returns the last physical address containing data from the section specified by *SectionName*.

SIZE(*SectionName)*

The SIZE function returns the current size of the section specified by *SectionName*. It is evaluated immediately and so reflects the current section size not the final size.

LINKEDSIZE(*SectionName*)

The LINKEDSIZE function returns the final link time size of the section specied by *SectionName*.

7.4 Groups

7.4.1 GROUP

The GROUP directive is used to allocate several sections contiguously in memory. Group attributes are set 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` *Attributes*

where:

GroupName	is a label defined by this statement. It must not be the name of an existing section or group name.
Attributes	are one or more of the following:
`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` [*Expr*]	allows a group to use assembly with offset i.e. group does not run at the normal contiguous address but at the address specified by *Expr* .
`size` *Size*	specifies the maximum size, *Size*, of a group. Enabling the PAD attribute forces the assembler to output the group padded to the specified size. The assembler will generate an error if the group is larger than the size specified by *Size*.
`bss`	defines a uninitialised data group.

`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 FF's so the burn takes less time. Settin gthe PAD attribute is effective only 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 group fragment 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.

7.4.2 Group Starting Address

There are two ways to set the starting address of a group, which can be mixed 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 provides individual starting addresses for groups. This is done by setting the ORG attribute for each group with the required starting address. ORG cannot be used inside a section but if sections and groups are not used then the ORG directive can be used anywhere.

Groups are normally placed in memory in the order in which they are encountered in a program. Groups that do not have an explicit starting address are placed at the end of the previously encountered group. Sections within each group are placed in memory in the order in which they are specified. When linking, 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 start address
        ...
Code    group               ;Loaded at $1000
Data    group               ;Loaded immediately
        ...                 ;after the end of Code
```

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
        ...
```

7.4.3 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.l    Table1,BssGroup
;Table1 is now long so BssGroup is now long
```

7.4.4 Writing Groups to File

The assembler allows you to write 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 condjunction 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.

7.4.5 Overlaying Groups

The OVER attribute enables you overlay groups i.e. to have several groups starting at the same address. A group specified as an overlay has the same starting address as the previous group. Enough space will be left for the largest group.

Example

```
Overlay1     group
Overlay2     group    over
Overlay3     group    over
```

7.4.6 Group Functions

OBJBASE(*GroupName*)

See also "Group Functions" on page 7-20.

The OBJBASE function returns the logical starting address of the group specified by *GroupName*, evaluated at link time.

ORGBASE(*GroupName*)

See also "Group Functions" on page 7-20.

The ORGBASE function returns the physical starting address of the group specified by *GroupName*, evaluated at link time.

OBJLIMIT(*GroupName*)

The OBJLIMIT function returns the last logical address containing data from the group specified by *GroupName*.

ORGLIMIT(*GroupName*)

The ORGLIMIT function returns the last physical address containing data from the group specified by *GroupName*.

SIZE(*GroupName*

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

LINKEDSIZE(*GroupName*)

The LINKEDSIZE function returns the final link time size of the group specied by *GroupName*.

4

The Debugger

About the Debugger

Running the Debugger

The Debugger Interface

The Main Window

Code Windows

The Registers Window

The Memory Window

The Watch Window

The Program Window

The Breakpoints Window

The Log Window

The File Viewer Window

The Local Vars Window

Breakpoints

Expressions

Expression Formatting

8 The Debugger

8.1 About 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, 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.

8.2 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.

8.2.1 Command-line Syntax

The command-line format consists of the debugger executable name optionally followed by two 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:

`snbugsat` *Switches*

Files Used by the Debugger

The files used by the debugger and their default extensions are given in Table 8-1 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 SNBUGSAT.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 8-1. Files used by the debugger.

Command-line Switches

The debugger has eight optional command-line switches, described in Table 8-1 below.

Switch	Description
[-|/]?	*Command-line help.*
[-|/]a	*Use default colour scheme*. This switch causes the debugger to ignore the colour settings in the INI and CFG files.
[-|/]b	*Use Borderless Edit Controls*. This switch causes the debugger to use single line buttons and borderless edit controls.
[-|/]i=*SessionFile*	*Use Project Info*. This switch invokes the debugger using the session save file specified by *SessionFile*. The session file contains information on how to connect to targets, the object files in use for each target, update rates, breakpoints, watch expressions, log expressions and window positions and displays. If no memory ranges are specified the debugger will look for them in SNBUG.CFG.

Switch	Description
[-|/]m	*Use Block Mouse Cursor.* This switch forces the debugger to use the block (non-graphics) cursor when running under DOS or in a DOS box under Windows. See also the m swtich.
[-|/]s	Use Shadowed Window Environment.
[-|/]t#[b][n]:[*ObjectFile*]	*Specify Target and Object File.* This switch specifies the target and the object file to load. The target is identified by its processor ID # (0-7) where 1=SH2 Master; 2=SH2 Slave. The object file to load is specified by *ObjectFile*. 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.

Switch	Description
[-|/]v*VideoSetting*	*Configure Video Setting*. This switch configures the debugger's video display according the the setting specified by *VideoSetting*. *VideoSetting* can be one of the following: [b|B] - BIOS screen wites. [c|C] - Colour[1]. [d|D] - Direct screen writes[1]. [g|G] - Direct screen writes, no CGA snow[2]. [m|M] - Monochrome[3]. [1] Default. [2] Default on CGA cards. [3] Default on monochrome cards.

Table 8-1. Debugger command-line switches.

Example 1

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

```
snbugsat    -i=initfile
```

Example 2

This example invokes the debugger and restores the debugging session according to the default session file SNBUGSAT.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...`).

```
snbugsat    /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.

```
snbugsat    -t7n:test
```

8.2.2 Session Files

The debugger gets its startup state from a *session file* (.INI) 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.

Project Infomation

Invoking the debugger using a session file containing project specific information restores project items, such as breakpoints, if present. If the ‘`do_bin`’ entry in the project section of the session file is non-zero the COFF file binary is downloaded to the target. If the ‘`do_syms`’ entry in the project section of the session file is non-zero any available symbolic debug info will be loaded automatically. Note that the ‘`do_bin`’ and ‘`do_syms`’ entries are normally set automatically when a session file is saved. Note also 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

The debugger will search for session files first in the current directory then in the debugger executable directory. If a session file is found the debugger will use the settings it contains.

If a session file cannot be found or if some settings, such as memory ranges, are not specified in the session file the debugger will use the defaults in the template session file SNBUG.CFG. **This file must be in the same directory as the debugger and should not be modified**. SNBUG.CFG also contains window default parameters such as size, colour and update rate. The debugger always uses these parameter values when a new window is opened. To change the parameter values,

select a window and configure it as desired then choose Save as Default from the Display menu to save the new parameter values in SNBUG.CFG. The new values will be used each time a new window of the same type is opened.

Editing Session Files

Information in session files can only be changed using a text editor. Be careful to retain the format shown in this manual 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, readwrite and write 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.

The Memory Range Format

[memory_*TargetNumber*]
Read=*StartAddress*,*EndAddress*,[*Size*],[*Expression*]
Write=*StartAddress*,*EndAddress*,[*Size*],[*Expression*]
ReadWrite=*StartAddress*,*EndAddress*,[*Size*],[*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.
Size	is the access method specifier and can be either `byte`, `word`, `triple` or `long`. The default is byte access.
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 1. The expression sets the read range only if bit $80 is set in byte C2BF.

```
[memory_1]
Read=0x00001000,0x00007FFF,[C2BF]&$80
Write=0x00008000,0x0000BFFF
ReadWrite=0x0000C000,0x0000FFFF
```

Window Attributes

SNBUG.CFG has one or more super-sections that contain information about window attributes. To create a processor specific super-section for a window type choose Save as Default from the Display menu. There is a default super-section called "Default_Windows" that contains non-processor specific infomation that is used if no processor specific information is available. Each processor type can optionally have a super-section that contains default position, size and colour attributes for each window. The format of a processor specific super-section is shown below.

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

where:

ProcessorID	is the name used to identify the processor.
WindowType	is one of: 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 SH2 Main processor that defines the size and position for the Disassembly window.

```
[[SAT_SH2_MAIN_Windows]]
[dis_wind]
Row=10
Column=10
Width=35
Depth=12
[Reg_Wind]
...
[[]]
```

8.3 The Debugger Interface

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.

8.3.1 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.

The Current Target

The currently selected target is highlighted 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.There are three ways to select a target (and initialise it if it has not yet been connected).

Selecting a Target

In any window use Shift+Target Number to select a target. This also initialises the target if it has not yet been connected.

In the Main window (described later) choose Target|Select from the menu to display the Select Target dialog box. Select the radio button corresponding to the target you wish to select and click OK. This also initialises the target if it has not yet been connected.

In the Main window click the left mouse button on the target's status line to select a previously initialised target.

Discarding a Target

There two ways to discard a target. Choose Discard from the Target menu in the Main window or use Ctrl+Alt+Target Number.

8.3.2 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). Any active debugger window can be moved or resized and any value in the Memory and Register windows can be edited.

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). 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:*Processor_ID*-*WindowName*-*WindowNumber*

Example

```
1:SAT_SH2_MAIN-Mem{3}
```

Resizing Windows

Using the Mouse

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. Note that this feature is disabled when a scroll bar is visible.

Using the Keyboard

To resize an active window press Alt+Space to display the system menu and choose Size. Use the arrow keys to resize the window and press Enter when done.

Moving Windows

Using the Mouse

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.

Using the Keyboard

To move an active window press Alt+Space to display the system menu and choose Move. Use the arrow keys to move the window and press Enter when done.

Selecting Windows

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

8.4 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 8-2 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 in the target list.

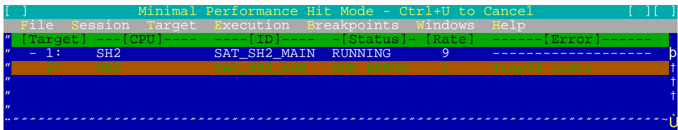


Figure 8-2. The Main debugger window.

Main Window Menus

The Main window has seven menus; File, Session, Target, Execution, Breakpoints, Windows and Help. 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 are now described in turn.

8.4.1 File Menu

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

File	
Load COFF with Debug Info...	Shift+Ctrl+C
L**o**ad Debug Info Only...	Ctrl+C
Reload Processor's Last COFF	
Reload All COFFs in Use	Ctrl+Alt+C
Send Binary...	Shift+S
Get Binary...	Shift+G
Prompt and Exit	F3
Save and **E**xit...	Ctrl+X
Quit (No Save)	Ctrl+Q

Loading Debug Info

Load COFF with Debug Info (Shift+Ctrl+C)

Selecting this command displays a dialog box requesting the name of the object file that will be used to send the binary code to the currently selected target. Any debugging information included in the file by the assembler will be loaded by the debugger. Line number information is discarded if multiple lines are reported for the same address (the last line number in the list is loaded). The target will be stopped after the COFF has been loaded.

Load Debug Info Only (Ctrl+C)

Selecting this command performs the same action as Load Binary & Debug Info except that it 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. The target will not be stopped after the debug info has been loaded.

Reloading COFFs

Reload Procesor's Last COFF

Selecting this command reloads the last loaded COFF file for the current target processor.

Reload All COFFs in Use

Selecting this command reloads all COFF files currently in use for all target processors.

Binary Transfers

Send Binary (Shift+S)

Selecting this command displays the Binary Transfer dialog box requesting the name of the file to send, the start address and the length of file. The length and end address are set automatically from the specified file. The default start address is the beginning of the file and the default length is the entire file. If the value for either the start address, end address or length is changed the other values will be adjusted automatically.

Get Binary (Shift+G)

Selecting this command displays the Binary Transfer dialog box requesting the name of the file to get, the start address and the length of file. The length and end address are set automatically from the specified file. The default start address is the beginning of the file and the default length is the entire file. If the value for either the start address, end address or length is changed the other values will be adjusted automatically.

Saving and Exiting

Prompt and Exit (F3)

Selecting this command displays a prompt requesting the name of the session file to save; the default file is the last saved or loaded session file or SNBUGSAT.INI if no custom session files have yet been saved. To enable the project to be restored to its current state, select the Save Project Information tick box. This saves: the COFF information in use; window positions and displays; and any breakpoints, watch expressions, log expressions that have been set.

Save and Exit (Ctrl+X)

Selecting this command exits the debugger and saves the session information to the default session file SNBUGSAT.INI, not the current (last saved or loaded) .INI file. Project information will be automtically saved if:

- it was restored from a session file (either SNBUGSAT.INI or a custom INI file)
- the Project Information box was ticked during a previous session save (using Ctrl+F3).
- a COFF was loaded or a breakpoint set during the current session.

Quit (no Save) (Ctrl+Q)

Selecting this meu item quits the debugger without any prompts or saving any session information.

8.4.2 Session Menu

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

Session	
Load...	F4
Save...	Ctrl+F3
Reload Last	Ctrl+F4
Minimal Performance Hit Mode	Ctrl+U

Loading Session Files

Load (F4)

Selecting this command loads a new session file, discards the current setup without saving it and restores the debugger settings from the file chosen in the Session Save Selector dialog box. 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

Reloading A Previous Session

Reload Last (Ctrl+F4)

Selecting this command starts a new session using the last saved or loaded session file.

Saving Session Files

Save (Ctrl+F3)

Selecting this comand saves the debugging session in its current state to a specified file, optionally including project information. The default file is the last saved or loaded session file or SNBUGSAT.INI if no session file has yet been saved or loaded.

Minimal Performance Hit Mode

Selecting this command reduces the debugger's influence on code running on the Saturn by stopping the debugger communicating with a processor whilst the monitored processor is executing code. Conditional breakpoints behave as normal. The Debugger uses Minimal Performance Hit Mode by default.

8.4.3 Target Menu

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

Target	
Select...	Shift+0..7
Discard...	Ctrll+Alt+0..7
Update Rate...	Shift+U
Monitoring...	Ctrl+M
Reset	Ctrl+Shift+R

Selecting Targets

Select (Shift+0..7)

Selecting this command displays the Target Select dialog box. This dialog box has eight radio buttons corresponding to possible target device numbers.

The default radio button is the current target. Buttons 1,2,4 select the Master SH2, Slave SH2 and 68000 targets respectively. Selecting any other button causes all three targets to be connected. Selecting a target will cause any windows associated with it to be brought to the front; the active window will be 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 performed by clicking the left mouse button on the target's information line in the Main window or by pressing Shift+Target Number from anywhere within the debugger.

Discarding Targets

Discard (Ctrl+Alt+0..7)

Selecting this command from the menu discards the current target. Selecting this command using the keyboard shortcut discards the specified target. Discarding a target will close all windows for the target and disconnect it.

Setting Update Rates

Update Rate (Shift+U)

Selecting this command displays the Update Rate dialog box. The Update Rate dialog box sets the foreground and background update rate for the currently selected target.

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.

If the Continuous check box is enabled the target will be continuously updated. Note that in this case the foreground and background update rates 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

Toggle Continuous Update Rate Ctrl+U

If the current target is running, selecting this command toggles continuous update on and off. If the current traget is not running, this command toggles Minimal Performance Hit Mode on and off.

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 ybetween breakpoints halting the execution of code on the target and the debugger detecting and processing them.

Monitoring (Selected Target) (Ctrl+M)

Selecting this command toggles monitoring of the selected target on and off. Turning monitoring off disables the connection, or ability to connect to, a target. No target updating or window refreshing takes place and no mouse clicks or keyboard presses are processed. 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 combined with ignoring requests for a forced update such as looking at a memory range. The Monitoring command can also be used to disable continuous attempts to reconnect to a target after a SCSI error without the need to discard the target.

Monitoring (All Targets) (Ctrl+U)

Selecting this command performs a similar action to Ctrl+M but toggles monitoring of all targets on and off. Unlike Ctrl+M this command allows keyboard presses and and tracing to force a window refresh. Use Ctrl+U monitor toggling to make the debugging session user triggered.

8.4.4 Execution Menu

See also "Code Windows" on page 8-30.

The Execution menu shown below provides a subset of the Execution menu found in Code (Source, Disassembly and Mixed windows).

Execution	
Run from PC	F9
Run to **A**ddress...	Shift+F9
Run All Targets	Ctrl+F9
Stop All Targets	
Single Ste**p**	F7
Step **I**nto	Shift+F7
Step **O**ver	F8
Unstep	Ctrl+F7
Halt	Esc
Halt (**D**MA & Interrupts)	Shift+Esc
Save Registers	Ctrl+S
Retrieve Registers	Ctrl+R

Running Code

Run from PC (F9)

Selecting this command starts the target executing code from the current position of the PC.

Run to Address (Shift+F9)

Selecting this command executes the current program until it reaches a specified address or executes a specified source file until it reaches a given line number.

Run All Targets (Ctrl+F9)

Selecting this command starts all targets executing code from the current position of the PC. This command is equivalent to selecting Run from PC for all targets.

Halt All Targets

Selecting this command stops code executing on all targets. This command is equivalent to selecting Stop for all targets.

Stepping Code

Single Step (F7)

In disassembled code, selecting this command causes the target to execute the instruction at the PC (using the current register values) and then stop.

In source code, selecting this command causes the target to execute the instruction at the PC (using the current register values). The target will stop when all low level assembly instructions generated by the single source instruction have been executed.

Step Into (Shift+F9)

In disassembled code, selecting this command causes the target to execute the instruction at the PC (using the current register values) and then stop.

In source code, selecting this command causes the target to execute the instruction at the PC (using the current register values) and then stop at each individually generated assembler instruction.

Step Over (F8)

In disassembled code, selecting this command causes the target to execute the instruction at the PC (using the current register values) and then stop.

In source code, selecting this command causes the target to execute the instruction at the PC (using the current register values) and then stop when the source file reference has changed.

Unstep (Ctrl+F7)

Selecting this command causes the target to untrace the action of the previous individually single stepped instruction.

Halting Code

Halt (Esc)

Selecting this command causes the target to stop executing code as soon as possible and leaves the PC at the start of the next instruction that would have been executed.

Halt (DMA & Interrupts) (Shift+Esc)

Selecting this command causes the target to stop in the same way as Stop but also disables all DMA accesses and stops all interrupts so that the target is forced into a safe state.

Reseting the Processor

Reset Processor (Shift+Ctrl+R)

Selecting this command causes the currently selected target to perform a processor reset.

Saving and Retrieving Registers

Save Registers (Ctrl+S)

Selecting this command saves the current contents of the target registers.

Retrieve Registers (Ctrl+R)

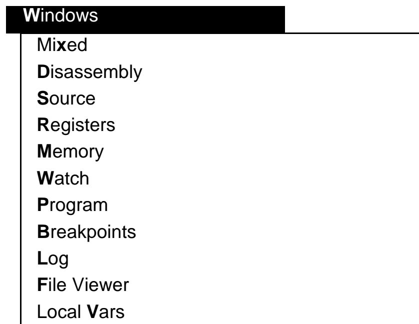
Selecting this command restores the previously saved register contents.

8.4.5 Breakpoint Menu

The Breakpoint menu provides the ability to remove all currently set breakpoints. There is one command on the Breakpoint menu: Remove All; this removes all currently set breakpoints for a target.

8.4.6 Windows Menu

The Windows menu creates a new window for a target. There are nine available window types: Mixed; Disassembly; Source; Registers; Memory; Watch; Program; Log; and File Viewer.



A target can have an unlimited number and mixture of windows with the restriction that only one Register and Log window is permitted per target. Mixed, Disassembly and Source windows are discussed together in “Code Windows” on page 8-30.

8.5 Code Windows

Types of Code Windows

The debugger supports three types of Code window: Mixed; Disassembly; and Source.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. The Mixed window combines the Source and Disassembly windows: source code is displayed in the upper region and the corresponding disassembled code in the lower region.

Running and Tracing Code

See also "Breakpoints" on page 8-57.
See also "Tracing" on page 8-65.

All Code windows provide the facility to run, trace and set breakpoints. For example, in a Mixed window to trace code at source level: select the source region of a mixed window. The assembly language statements corresponding to each executed source statement can be viewed 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.

8.5.1 Mixed Window

The Mixed window is a combination of the Source and Disassembly windows and so supports the features of both window types.

The Mixed Window Regions

The Mixed window 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. To start the display at a given line double click on the line in the left hand side of the display.

Click the left mouse button in the relevant region or use Space to toggle the focus between the two regions.

Marked Instructions

The left-hand column of the Disassembly region is used to display markers to identify slot instructions and the location of the target's PC.

The PC Marker

If the target's PC is at one of the instructions displayed in the disassembly region, the instruction is marked with a '<*>', in the left hand coloumn. known as the PC Marker. The PC marker is yellow for valid instructions and red for invalid instructions. The PC marker is also displayed in red when the disassembly region is the inactive region of the Mixed window. If the PC is currently on a slot instruction the maker is changed to display '<s>'

The Slot Instruction Marker

Slot instructions are marked with '-s-'. If the PC is currently on a slot instruction the maker is changed to display '<s>'.

Selecting Source Instructions

To select a line click the left mouse button on the line. 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.

Setting Breakpoints

Clicking in the left hand side of the display will select a line and set a breakpoint on it. Pressing Ctrl+F5 displays the Breakpoint configuration dialog box.

The Display Menu

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

Display	
Update Rate	Shift+U
Save As **D**efaults	
Zoom	Ctrl+Y
Switch **A**ctive	Space
Centre on **T**race	
Centre on **B**point	
Centre on Instr. **E**rror	

The Origin Menu

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

Origin	
Goto...	Ctrl+G
Go to **C**ursor	Home

The Format Menu

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

The Execution Menu

See also "Tracing" on page 8-65.

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 **A**ddress...	Shift+F9
Run to **C**ursor	F6
Run All Targets	Ctrl+F9
Stop All Targets	
Single Step	F7
Step **I**nto	Shift+F7
Step **O**ver	F8
Unstep	Ctrl+F7
Stop	Esc
Stop (**D**MA & Interrupts)	Shift+Esc
Rese**t** Processor	Shiftl+Ctrl+R
Save Registers	Ctrl+S
Retrieve Registers	Ctrl+R

Run All Targets (Ctrl+F9)

Selecting this command starts all targets executing code from the current position of the PC. This command is equivalent to selecting Run from PC for all targets.

Stop All Targets

Selecting this command stops code executing on all targets. This command is equivalent to selecting Stop for all targets.

The Breakpoints Menu

See also "Breakpoints" on page 8-57.

The Breakpoints menu provides the ability to: set and configure individual breakpoints; and to remove all breakpoints.

Breakpoints	
Toggle at Cursor	F5
C**o**nfigure...	Ctrl+F5
Remove All	Shift+F5

The Utils Menu

The Utils menu provides access to the expression calculator and search facilities.

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

Expression Calculator (Ctrl+E)
Selecting this command invokes the expression calculator.

Find (Ctrl+F)
Selecting this command invokes the Search dialog. The search is case-sensitive and uses the current Format settings for case, radix, labels and symbol display when performing the search.

To search for a text string or symbol:

1. Specify what to search for. To search for a symbol enter its name. To search for text enter a double quote followed by the search text. To include trailing spaces, close the search text with a double quote.
2. Specify the start address. The default value is the start address from the previous search. If no searches have been performed the address at the start of the currently displayed block of memory is used.

3. Specify either the length or end address; the corresponding Length or End Address field will be filled in automatically. The default values are the length or end address from the previous search. If no searches have been performed the address, and corresponding length, at the end of the currently displayed disassembly is used.

Example

A possible search string could be:

```
"A search string
```

A possible search string with trailing spaces could be:

```
"A search string with trailing spaces "
```

A possible symbol search could be:

```
Master_Start+20
```

8.5.2 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. See the Mixed window for a full description.

8.5.3 Source Window

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

8.6 The Registers Window

The Registers window displays the contents of the processor's general registers and the instruction at the PC. Choose Registers from the Windows menu to open the Registers window, shown in Figure 8-1 below. The window defaults to a horizontal, hexadecimal display.

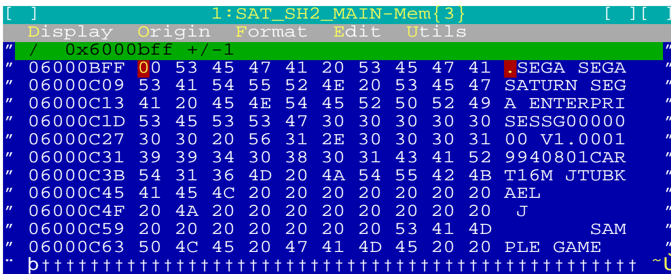


Figure 8-1. The Registers window.

The Status Bar

The status bar displays information about the current target.

Formating the Display

The Format Menu commands control the display of the Memory window.

Editing Register Values

You can edit the value of any of the registers by entering the new value at the cursor position. (For a flag register Press ‘1’ to select a bit and ‘0’ to clear it.) In addition, the Edit menu provides access to commands for changing register values. Use + and - to increment or decrement the value of a register, 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 invoke the expression evaluator use Ctrl+E.

Saving and Retrieving Register Values.

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 change the window refresh rate choose Update from the Display menu or use Shift+U.

8.7 The Memory Window

The Memory window shown in Figure 8-2 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.

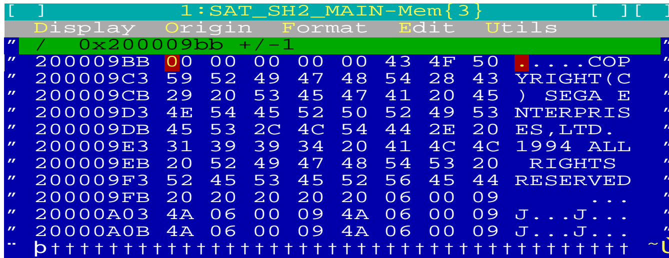


Figure 8-2. The Memory window.

Formatting the Display

The default display shows memory contents as bytes on the left-hand side 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. To display a specified number of bytes use Ctrl+B. 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 Ctrl+A.

Editing Memory Locations

To edit 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. The default value incremented or decremented by **+** and **-** is one. To change the amount incremented or decremented by **+** and **-** choose Inc/Dec Amount from the Edit menu.

Copying and Filling Memory

Memory Fill (Shift+F)

Selecting this command fills a range of memory with a specifed byte. The specified memory range can contain invalid areas but these areas will not be read from or written to.

Memory Copy (Shift+C)

Selecting this command 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.

8.7.1 Finding Memory

Find Memory (Ctrl+F)

Selecting this command allows you to define and search an area of memory for a specified pattern of data. If a match is found, the address of the match is displayed. The search will automatically skip over any sensitive areas such as invalid memory areas, write-only memory and memory reserved for the monitors.

Finding a Pattern in Memory

Memory searches are specified in the Search Target Data Block dialog box.

To find an area of memory:

1. Select the Mode as either Binary, Decimal, Hex or ASCII.
2. Select the Width as either Byte, Word or Long. Width specifies how to compare the search pattern with the memory contents by aligning the search pattern with the data in the target's memory.

   The allowable search width depends on the search mode chosen. The valid Mode and Width combinations are shown in Table 8-1 below.
3. Specify the pattern to search for. In binary, decimal and hex modes the search pattern can optionally be delimted by either commas or a semi-colons. The meanings of the comma and semi-colon delimiters are discussed below.
4. Specify the start address. The default value is the start address from the previous search. If no searches have been performed the address at the start of the currently displayed block of memory is used.

5. Specify either the length or end address; the corresponding Length or End Address field will be filled in automatically. The default values are the length or end address from the previous search. If no searches have been performed the address, and corresponding length, at the end of the currently displayed block of memory is used.

Mode	Valid Widths
Binary	Binary, Word, Long
Decimal	Byte
Hex	Binary, Word, Long
Text	n/a

Table 8-1. Mode and Width combinations for memory searches.

Example

The following patterns are treated as equivalent when the Hex and Byte widths are set:

```
FF,FF,FF,FF,34,DC
FF;FF;FF;FF;34;DC
FFFFFFFF34DC
```

The ? Wild Card

A special wild card character, '?', can be used in Binary and Hex modes. The '?' character specifies a nibble in Hex mode and a bit in Binary mode that always results in a successful match.

Example

In Hex mode, `FF?F` will match with `FF0F,FF1F,FF2F,...,FFFF`. In Binary mode, `????1111` will match with `00001111,00011111,...,11111111`.

Automatic Padding

The search pattern is automatically left-padded for the Binary, Decimal and Hex modes. The type of padding is either ‘0’ or ‘?’ depending on the delimiter used.

The Comma delimiter

Delimiting the search pattern with commas (the default) automatically left-pads it with zeroes. This means that in Hex mode with Byte width the search patterns, `FFFFFFFF34DC` and `FF,FF,FF,FF,34,DC` for example, perform the same sarch as the comma sepaerator is implied in the former. Further examples, in Hex mode and Word width, are:

`f,87d,a`	automatically pads to	`000F,087D, 000A`
`f87da`	automatically pads to	`000F,87DA`
`,`	automatically pads to	`0000`

Note that a single comma used on its own produces the pattern 0000. This can be useful but you should be careful when using this feature. For example,

`,7`	automatically pads to	`0000,0007`

which may not be the desired search pattern.

The Semi-colon Delimiter

Delimiting the search pattern with semi-colons automatically pads it with the ‘?’ wild card. Examples, in Hex mode and Word width, are:

`f;8d;a`	automatically pads to	`???F,?87D,???A`
`f;87da`	automatically pads to	`???F,87DA`
`;`	automatically pads to	`????`

Examples

You can specify the same same search pattern in different ways using the comma and semi-colon delimiters. The following patterns are treated as equivalent when the Hex and Byte widths are set:

```
FF,FF,FF,FF,34,DC
FF;FF;FF;FF;34;DC
FFFFFFFF34DC
```

The comma and semi-colon delimters can be mixed to produce precise search patterns. For example, in Hex mode and Word width:

```
f;f0f0f0f0,fffffff?,;7;
```

automatically pads to

```
???????F,F0F0F0F0,FFFFFFF?,???????F,???????7
```

8.8 The Watch Window

See also "Expression Formatting" on page 8-72.

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.

Using Watch Expressions

To add a watch expression use Ctrl+A. Alternatively, press Enter on an empty watch line to invoke the watch expression editor. A new expression can then be entered in the next free slot. The current expression can be edited by pressing Enter or deleted using Ctrl+D. To move between expression entries can use the cursor keys or click with the left mouse button.

8.8.1 Structure Browsing

The Watch window allows you to view and browse the data structures in the currently loaded COFF. Compound or derived data types can be expanded and contracted .

Displaying Structre Browse Information

To display structure browse information the currently loaded COFF must have been produced from C source files compiled, with debug enabled (`-g`), and linked by either the Sierra or GNU tools.

To display a data type and its associated values enter the variable name as the watch expression. Compound or derived data types which may be examined in greater depth have a '+' character in the leftmost column of the display.

To expand the structure of a compound or derived data type first highlight the required expression then press Spacebar to expand the type structure by one level. Note that the '+' symbol changes to a '-' indicating that the structure may be collapsed. Type expansion is possible until an integral type is displayed. To collapse an expanded type structure press Spacebar.

Individual lines may be deleted to allow only specific browse items to be watched. To delete a browse item first select the required item then use Crtl+D to delete it. Note that removing an expanded browse item also deletes all data it contains.

8.9 The Program Window

The debugger's Program window provides the ability to create new windows tailored to specific needs. This facility is made possible by extending the Tcl language used to construct the debugger’s standard windows. A standard Tcl interpreter is provided with SNASM2 specific extensions to allow; access to target memory, a display window, and debugger events.

How the Program Window Works

The Program window is initialised by a user’s Tcl program. This binds scripts and/or procedures to key and mouse events and to two timer driven debugger events: “refresh”, for accessing target memory; and “update”, for display changes.

Creating and Editing Tcl Programs

To open the Tcl program editor choose New Program from the Utils menu or use Alt+E. The program editor has a fixed size and position.

To save the program source use Alt+X; this also exits the editor. The modified code is run in a re-initialised interpreter.

To Find Out More About Tcl

A full and informative description of Tcl is given in “Tcl and the Tk Toolkit” by John K Ousterhout (the creator of Tcl). Published by Addison Wesley, 1994. ISBN 0-201-63337-X

8.9.1 SNASM2 Tcl Extensions

The SNASM2 extenstions to the Tcl language are described below.

put [*Row Column*] *Text*

This displays the string *Text* at the current cursor position or at (*Row*,*Column*), if specified.

clear

This clears the window and places the viewport origin and cursor position at (0,0).

setrc *Row Column*

This sets the cursor position to (Row,Column).

where {Cursor|Mouse}

This returns the position of the cursor or mouse as "*Column Row*" in decimal.

readmem [{Byte|Word|Long}] *Address Count*

This returns a list of *Count* hex numbers of the specified size (the default is Byte) from *Address*.

sendmem [{Byte|Word|Long}] *Address Values*

This writes *Values* (a list of numbers) in the specified size (default Byte), to *Address*.

sym *Name*

This returns the value, in hex, of *Name* in the current target's symbol table. If no match can be found the message "Symbol not found" is returned.

tosym *Value* [{Exact|Before|After}]

This returns the name of the symbol with value *Value*, according to the specified search mode (default Exact). The search modes are:

Exact	This returns a symbol found with value *Value*.
Before	This returns the symbol with value next smaller than *Value*.
After	This returns the symbol with the next greater value than *Value*.

If no match is found the unmatched value *Value* is returned. Multiple symbols with the same value can be obtained by repeated calls using the exact match mode.

firstsym *Name Pattern*

This initialises a scan of the symbol table for symbol names matching the 'glob' style pattern *Pattern*. *Name* is the name of the Tcl variable to use for storing internal scan progress information. Returns the first match found or "Not Found" if no match was made.

nextsym *Name*

This continues the symbol table scan initiated by a call to firstsym with variable name *Name*. Returns the next matching symbol name or "Search Completed" if no further match can be found.

getvalue *Title*

This opens a dialog box, with title *Title* to prompt for a string value. Returns the entered text. This command may not be invoked during a refresh event (see below).

bind [Event [Script]]

This creates bindings to events or returns details of bindings made depending on the arguments specified. Event specifiers are enclosed in ‘<’ and ‘>’ and scripts are quoted with { }. bind with no arguments returns a list of all events with defined bindings. bind with the Event argument alone returns the script, if any, bound to Event. bind with both Event and Script arguments creates a new binding, or replaces an existing one such that Script is evaluated whenever Event occurs.

Supported events are:

<refresh>	This refreshes communication with the target. Data reads and writes should be done here but not displayed as it slows down the target.
<update>	This updates the window. Changes to the display of data are done here. Reading or writing target memory should be avoided.
<sequence>	This specifies a keypress or mouse click. Keys are specified by name or letter which matches any unbound key combination. Mouse clicks are <Button1>, <Button2> or <B1> and <B2> etc.

The Event type may be qualified by a number of hyphen separated modifiers: Shift, Ctrl and Alt are keyboard modifiers for key or mouse event; Single, Double and Release represent the type of mouse event; Any is any combination, including none, of the above. Note that more explicit bindings are matched in preference to the general variety. Sequence specification is case insensitive.

readbin @[~]*Filename Length Address* [report]

This reads binary data from a target using the specified file *Filename*. Filenames must be prefixed with ‘@’. The optional ‘~’ character will prefix the Debugger’s executable path to the specified filename. The optional report parameter causes the

Debugger to report'in progress' information, and a success/ failure summary.

The target is defined as the processor associated with the 'Program Window' running the .PRG file.

sendbin @[~]*Filename Length Address* [report]

This sends binary data to a target using the specifed file Filename. Filenames must be prefixed with '@'. The optional '~' character will prefix the Debugger's executable path to the specified filename. The optional report parameter causes the Debugger to report'in progress' information, and a success/ failure summary.

The target is defined as the processor associated with the 'Program Window' running the .PRG file.

Examples

This example reads 1Mbyte of data from the target commencing at address 0x6100000. The data is read into the file TEST.BIN located in the Debugger executable directory. Full 'in progress' reporting is requested.

```
readbin @~TEST.BIN 0x100000 0x6100000 report
```

To send 1Mbyte of data to the target address 0x6100000 from the file TEST.BIN located in the current directory using 'quiet' mode (no reporting other than error conditions)

```
sendbin @TEST.BIN 0x100000 0x6100000
```

8.10 The Breakpoints Window

See also "Breakpoints" on page 8-57.

The Breakpoints window has a similar display to the Watch window. The Breakpoints window displays a list of all breakpoints set and is dynamically updated. Manipulating breakpoints from within the Breakpoints window affects all Code windows.

Using Breakpoints

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 configured by pressing Enter or removed using Ctrl+D. To move between expression entries use the cursor keys or click with the left mouse button.

Breakpoints in Source Displays

Some source instructions generate several assembly instructions. In a Disassembly window or disassembly region of the Mixed window, to set a breakpoint for such source instructions requires setting the breakpoint on the first assembly instruction generated by the source instruction. The breakpoint cannot be set on the second or subsequently generated assembly instructions.

8.11 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 or Breakpoint 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.

8.12 The File Viewer Window

The File Viewer window displays ASCII text files, usually source code. Opening a File Viewer window for a new file displays the file selector dialog box. The title bar of the file selector dialog box displays the name and date of the file being viewed. Note that files can be viewed but not modified in this window.

8.13 The Local Vars Window

The Local Vars window shows all variables in the current scope. The Local Vars window is similar to the Watch window except that the variables that are displayed change as the scope changes.

Individual lines may be deleted to allow only specific variables be shown. To delete a variable first select it then use Crtl+D to delete it.

The values of local variables can be modified. Choose Edit (Add) from the Utils menu or press Enter to modify the value of a variable.

8.14 Breakpoints

The debugger provides a powerful and flexible breakpointing facility, from simple single-shot to complex conditional breakpoints, and can log the output to monitor the state of variables and registers as the program runs.

The debugger supports the hardware breakpoint facilities available on the SH2. Hardware breakpoints are implemented by setting breakpoint conditions in the SH2's User Break Controller (UBC). A user break interrupt request is sent to the SH2 when these conditions are met.

The next sections describe how to use breakpoints, how to configure software and hardware breakpoints

8.14.1 Using Breakpoints

This section describes how to use breakpoints in the debugger. It shows you how to set, configure and clear breakpoints. It also shows you how to view breakpoints and what breakpoint information is saved in session files.

Setting Breakpoints

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 breakpoints can be set whilst the target is running and take effect immediately.

Configuring Breakpoints

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.

Clearing Breakpoints

To clear a breakpoint, first highlight the breakpoint and than use F5 or click the left mouse button in the left-hand margin to remove it.

See also “The Breakpoints Window” on page 8-53.

Viewing All Defined Breakpoints

The Breakpoints window provides a global view of all defined breakpoints for each target. The breakpoint window also provides the ability to define, remove or configure breakpoints.

See also “Session Files” on page 8-8.

Breakpoints and Project Information

Breakpoints can be saved in a session file by ticking the Project Information tick box. Note that breakpoints will only be restored as part of a session restore if the same underlying binary code is present. If the instruction code at the breakpoint address is different to that present when the breakpoint was specified, the breakpoint is discarded.

8.14.2 Configuring Software Breakpoints

Breakpoint characteristics are controlled through the Breakpoint Configuration dialog box shown in Figure 8-2 below. The Fname:Line# field displays the filename and line number on which the breakpoint will be set. The Address field shows the address of the instruction on which the breakpoint will be set. The Hardware tick box specifies whether the breakpoint will use the hardware breakpoint facilites available on the SH2. The debugger uses software breakpoints by default; hardware breakpoints must be specifically enabled.

There are two sets of check boxes for configuring the type of breakpoints and the action they take: Condition Flags and Action Flags. The Condition Flags check boxes set the conditions under which a breakpoint applies, the Action Flags check boxes set the action to take when the breakpoint is encountered.

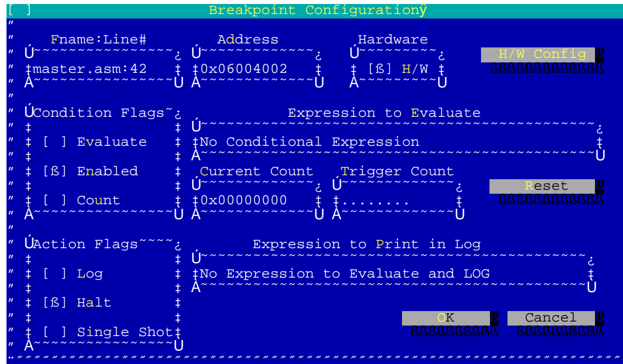


Figure 8-2. The Breakpoint Configuration dialog box.

Setting Condition Flags

Condition Flags specify the type of breakpoint condition.

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

Use Enabled to set a breakpoint. Clearing this check box disables a set breakpoint. The breakpoint is not discarded and mthe current settings are kept. 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.

Setting Action Flags

Action Flags specify what will happen when the breakpoint is encountered. There are three possible actions; Log, Halt and Single Shot.

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.

Reset

The Reset button sets the current count to zero.

8.14.3 Configuring Hardware Breakpoints

The debugger supports the hardware breakpoint facilities available on the SH2. Hardware breakpoints are implemented by setting breakpoint conditions in the SH2's User Break Controller (UBC). A user break interrupt request is sent to the SH2 when these conditions are met.

Channel Management

The UBC has two channels, Channel A and Channel B. Channel B is more sophisticated as it allows you to set breakpoints on data bus conditions. The debugger automatically allocates channel resources according to the configuration of the currently enabled hardware breakpoints.

The User Breakpoint Config Dialog Box

Hardware breakpoints are configured through the User Breakpoint Config dialog box. Options in this dialog box represent the break compare conditions that can be set in the UBC. The options are each described in turn below.

Address

This is the address in hex at which the breakpoint is set.

Mask

The is the bit mask for the Address, entered in hex; the bit mask is generated automatically.

Break Cycle

This is the Bus master where:

CPU	is the CPU cycle condition.
Periph	is the peripheral cycle condition.
ChipExt	is the chip-external cycle condition.

Break Access

This is the Bus master where:

InsFetc — is the Instruction fetch condition. Set this to trigger the breakpoint immediately before execution of the instruction at the address specified by Address.

BrkAfte — is the Instruction After condition. Set this to trigger the breakpoint immediately after the instruction at the address specified by Address.

Data — is the Data access condition. Set this to trigger the breakpoint on data access.

Access Cycle

Read — is the Read condition. Set this to trigger the breakpoint on a read access cycle.

Write — is the Write condition, Set this to trigger the breakpoint on a write access cycle.

Operand Size

Byte — is the Byte condition. Set this to trigger the breakpoint if the size of the instruction's operand is Byte.

Word — is the Word condition. Set this to trigger the breakpoint if the size of the instruction's operand is Word.

Long — is the Long condition. Set this to trigger the breakpoint if the size of the instruction's operand is Long.

Channel B Specifics

Data
The data value condition in hex. This is a single 32-bit value representing the two combined 16-bit break data registers.

Masks
Bitmask for the data value condition. The bitmask is entered in hex as a single 32-bit value, representing the two combined 16-bit break data mask registers. Set bits to 0 to include the corresponding data value condition bit; set bits to 1 to mask the corresponding data value condition bit. The bitmask is displayed automatically.

Inlcude Data Bus in Conds.

Tick box to include the data bus in the breakpoint condition.

8.14.4 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

This forces individual asssembly instructions to be traced one at a time.

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 JSRs 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. In the case of a single source instruction generating many assembly instructions you will need to press Shift+F7 several times on the source instruction before progressing to the next source instruction. In Mixed windows your progress through the source instruction trace is displayed in the lower disassembly region. In the disassembly region the PC moves one assembly instruction for each key press. This enables you to debug macro and complex source instructions at a detailed level.

Unstep Ctrl+F7

All instructions traced using Step Into may be untraced. 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 or Line-F's encountered whilst single stepping. 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.

8.15 Expressions

The debugger uses a similar expression evaluator to the assembler with the addition of powerful expression formatting facility. Use Ctrl+E to invoke the expression calculator.

Default Base

In contrast to the assembler, the default base is hexadecimal 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`.

Expressions enclosed in square brackets return the word at the memory location given by the result of the expression. The square brackets can optionally be suffixed with @B, @W, or @L to return the byte, word or long respectively.

8.15.1 GNU C++ Qualified Function Names

The expression evaluator accepts GNU C++ qualified function names, with support for class functions and overloaded operators. Global symbols (i.e static members) within a class can be viewed using a fully qualifed name in the Watch window.

Names are decoded according to scheme described in "The Annotated C++ Reference Manual" by Margaret A. Ellis and Bjarne Stroustrup.

Syntax

The fully qualified name must be used so that the symbol reference can be resolved:

`[`*ClassName*`::][,`*ClassName*`::]...`*SymbolName*

where:

ClassName	is the name or comma delimited list of names required to form the qualifed function name.
SymbolName	is the name of a function or operator.

Note that white space is ignored. For example, the following expressions will be considered as identical:

```
foo :: operator !
foo::operator!
```

Examples

Consider the following section of code for a class definition:

```
void *operator new (size_t size)// global overloaded operator
{
  return ::new char [size];
}
class foo // dummy class foo
{
    // internal variables
    int foo_a, foo_b;
  public:
    // static variable
    static int static_var;
    // constructors & destructor
    foo() { foo_a = 0; foo_b = 0; };
    foo(int a, int b = 0) {foo_a = a; foo_b = b};
    virtual ~foo() {};
    // class functions
    virtual void display();
    void *operator new(size_t size) {return ::new char
[size];};
    foo &operator !(foo &a);
    // conversion operators
    operator int();
    operator foo2();
};
int foo::static_var;// storage space for static
```

Valid Expression Qualifiers

The valid expression qualifiers produced from this class is:

`operator new`	Global New
`foo::operator::new`	Class foo overload of operator new
`foo::static_var`	Static variable
`foo::~foo`	Class destructor
`foo::operator int`	Conversion operator from integer
`foo::operator foo2`	Conversion from another class type foo2

Invalid Symbols

In the expression evaluator, entering an invalid symbol name, such as ‘`foo::operator |`’ instead of ‘`foo::operator !`’ will generate an error at the ‘::’ token not the ‘|’. This is because the expression evaluator is unable to evaluate which of the tokens is incorrect and so reports the first of the tokens it is unable to decide on.

Considerations and Limitations

- Overloaded functions cannot be distinguished between using only the qualified name e.g. `foo::foo`.

- Global constructors or destructors are created if a class variable is defined within file scope. This situation normally defines two special labels on the constructor or destructor:

  __GLOBAL_I*String*
  __GLOBAL_D*String*

  where:

  *String*is a string from an unknown class function name.

  These labels are not decoded as no information about how they were generated is available.

- Symbol names are limited to 127 characters.

- Do not use compiler generated default constructors or destructors as a Goto in the Source window or source region of a Mixed window. This will cause incorrect line references to be displayed in the Disassembly window or disassembly region of the Mixed window. This happens because no source code exists for the constructor or destructor; the line number that was assigned just before the compiler created the code is displayed instead. This line number is usually the last line of a class definition.

- When browsing C++ sub-structures and sub-classes in the Watch window you have to follow two sets of indirection, not the single set provided by a C compiler.

- In the Watch window some references a virtual table e.g. **$vf, **$vb are displayed as pointers to an unknown class. This is because these variables exist in within classes that that were derived from virtually from parent classes or contain virtual classes.

8.15.2 Symbol Completion

The Expression Evaluator has a symbol completion facility. In the Expression evaluator dialog, to complete a partially entered symbol press the Symbol button. If a unique symbol name can be identified the name will be completed automatically. If more than one possible name exists you will be presented with a list of symbol names to choose from. The list will be sorted in alphabetical order, beginning with the character at the end of the input text. Pressing the Symbol without first entering part of a symbol name will present you with a list of all symbols in the currently loaded COFF. Valid symbol characters are A-Z, a-z, 0-9, '_', '.', ':' and '~'.

8.16 Expression Formatting

The debugger provides a powerful expression formatting facility for controlling the display of expressions in Log and Watch windows. Formatting is controlled by the use of *formatting expressions* which work in a similar way to the C ‘printf’ 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*]...

8.16.1 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.

Each expression must have the same number of operands as format specifications ('%' characters). Insufficient operands will cause the debugger to generate an error.

All expression operands must evaluate. If an operand evaluates to a section relative address the string is displayed as "*SectionName*`:`*Value*".

8.16.2 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 Table 8-4 below.

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 8-4. Format specifier characters and their effects.

Examples

`%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’).

8.16.3 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.

Syntax

[+|-]*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.

Examples

The following example shows a formatting string that displays its three parameters as decimal signed integers 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"
```

8.16.4 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.

Syntax

[-][*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

`%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.

8.16.5 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.

Syntax

`@`[*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.

Utilities

SNMAKE

SNLIB

SN2G

9 SNMAKE

The SNMAKE utility enables the SNASM2 development system to be easily used from within a text editor. SNMAKE works on the common make utility principle of reading a file containing user defined 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 dependent 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.

9.1 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.

See also "Command-line Syntax" on page 9-12.

Use Alt+F9 to invoke the SNASM2 main menu from within one of the supported text editors. Selecting 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 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.

9.2 Project Files

9.2.1 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.sh2 to t1:
[snmake]

t1:;prog.sh2
  snasmsh2 $! /sdb prog.sh2,t1:prog

[debug]
  snbugsh2 -t1:prog

[eval]
  evalsym /v$$$ prog
```

9.2.2 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;
t1;
e:netwrkt7;
```

Note that white space in target declarations is stripped out.

9.2.3 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

See also "Example 3" on page 9-6.

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 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.

Example 1

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;                dep1
  somecmd  dep1 ... very long command line > 128 chars
```

SNMAKE will automatically detecting an over-long command-line and when this occurs create a temporary response file, calling SOMECMD as follows:

```
somecmd @tmp1.$$$
```

Example 3

```
[SnMake]
.SNRES;         snasmsh2

t1:;            testmain.sh
                snasmsh2 $! testmain.sh,t1:

t2:;            testsub.sh
                snasmsh2 $! /K testsub.sh,t2:

!ifdef(debugstr)
                snbugsh2
!endif

[debug]
                snbugsh2
```

9.2.4 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.SH and SRC2.SH. SNMAKE will attempt to invoke any rules defined for TARGET1.COF if either SRC1.SH or SRC2.SH have been updated since TARGET1.COF was last created.

```
target1.cof;    src1.sh src2.sh
```

9.2.5 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

This example defines a rule telling SNMAKE to issue the command `snasmsh2 /l dep1 dep2,target1.cof` if TARGET1.COF is younger than either DEP1 or DEP2.

```
target1.cof;      dep1 dep2
  snasmsh2 /l dep1 dep2,target1.cof
```

9.2.6 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 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 '.SH' extension as follows:

```
.cof,.sh
```

The second part of the declaration must begin with a '.' character, unless a path name is specified as below:

```
.cof,e:temp\.sh
```

This tells SNMAKE that any targets with a '.COF' extension are to be created from a dependency of the same name but with a '.SH' extension in directory E:\TEMP.

9.2.7 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 $- where $+ specifies the target and $- its dependency. Invoking the following implicit rule

```
.cof,.sh
  snasmsh2 $+,$-
```

on target PROG1.COF will result in the following command:

```
  snasmsh2 prog1.sh,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,.sh
```

will create any suitable targets in this area from files of the same name with a ‘.SH’ extension.

9.2.8 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 1. Without the continuation mark SNMAKE would truncate the dependency list at `dep9`.

```
target1;dep1 dep2 dep3 dep4 dep5 dep6 dep7 dep8 dep9 \
     dep10 dep11
```

9.2.9 Comments

Comment lines begin with a ‘#’ character and can start at any position on the line.

9.2.10 Macros

See also "Command-line Syntax" on page 9-12.

Macros can be passed into SNMAKE from the command-line using the `e` switch. 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 9-1. 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'
```

9.2.11 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 SNASMSH2 command lines in the project file as shown below:

```
targ1;          dep1 dep2
  snasmsh2 $!  /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:

```
snasmsh2 /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.

9.2.12 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)
t1:;         prog1.sh
  snasmsh2 $! /sdb prog1.sh,t1:prog1
  snbugsh2 -t1:prog1
!else
t1:;         prog1.sh
  snasmsh2 $! prog1.sh,t1:
!endif
```

Example 2

This is a more efficient implementation of Example 1.

```
t1:;prog1.sh
  snasmsh2 $! $(SRC_DB) prog1.sh,t1:prog1
!ifdef(debugstr)
  snbugsh2 -t1:prog1
!endif
```

9.3 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.

9.3.1 Switches

SNMAKE accepts five switches from the command-line.

Switch	Description
`b`	*Build all.* All rules carried out regardless.
`d`	*Set debug mode*. Sets the special macro $!. Only of use if invoking SNASM2 from the project file.
`e` *Name=Exp*	*Pass a macro definition into SNMAKE*. Sets up a macro *Name* which will expand to *Exp*.
`i`	*Set info mode.* As above.
`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.
`q`	*Quiet mode*. No echoing is done as SNMAKE proceeds.

Table 9-2. SNMAKE command-line switches.

9.3.2 Example

The following example produces a file called TEST1.COF from the source files E:SRC1.SH E:SRC2.SH . *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.sh
  snasmsh2 $! /l /sdb e:\src1.sh,src1.cof

src2.cof;    e:\src2.sh
  snasmsh2 $! /l /sdb e:\src2.sh,src2.cof

test1.cof;   src1.cof src2.cof
  snasmsh2 $! src1.cof+src2.cof,test1.cof
!ifdef(debugstr)
  snbugsh2 -t1b:test1.cof
!endif

[Debug]
```
SNMAKE stops reading here.
```
  snbugsh2 -t1b: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.

10 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.

10.1 Running SNLIB

10.1.1 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 10-1. SNLIB command-line switches.

This is the only information this page contains.

11 SN2G

12.1 About SN2G

SN2G is a utility program for converting SNASM2 COFF object modules into GNU format COFF object modules. Due to some of the extensions that the SNASM2 assembler uses not all objects can be converted. In such cases the converter will provide as much information as it can as to why the object could not be converted.

The converter exists because the GNU linker, ld, cannot link SNASM2 object files. This prevents you from using the SNASM2 assembler with the GNU linker. Furthermore, if you use the SNASM2 linker with GNU object files, you will lose your C structure browse information. This situation is undesirable as this information can be vital for debugging C programs. To ease this situation, use SN2G.EXE to convert your assembler objects into GNU ld format and link using ld. This provides the best of both worlds by enabling you to use the SNASM2 assembler with the GNU linker.

12.2 Command-line Syntax

The command-line syntax for SN2G is as follows:

`s2g` [[-|/]*Switch*] *Infile Outfile*

where:

Infile	is the SNASM2 object file to convert.
Outfile	is the converted object file.
Switch	is one of

- `v` Verbose mode.
- `d` Produce error information.
- `h` Provide this help message.

Example

As an example, you may use the converter in the following manner

```
c:\>gcc -g -c test.c -o test.o [return]

c:\>snasmsh2 /l test2.asm,test2.cof [return]

c:\>s2g test2.cof test2.o [return]

c:\>ld -g test.o test2.o -o test.x -Ttest.cmd [return]
```

12.3 Considerations and Limitations

SN2G can convert relocations of four byte quantities. This means that most symbol patches work. SN2G has to convert SNASM2's 'section relative' relocates into symbol relative relocates. This is achieved by putting a symbol at the start of each section, and relocating with that symbol.

SN2G also converts symbols. All symbols are converted to address labels. They contain no specific type information.

There are several features that cannot be converted because they are either not supported by the GNU linker or have no meaning in that context. These features are described below.

Complex Expressions

Complex expressions cannot be converted. For example, if an expression involving two symbols cannot be evaluated by the assembler then SN2G will output an expression for the relocation. Currently, these expressions are not converted. It is not known if it is possible to convert complex ecpressions. The work around is to re-write the code so that the symbols are no longer one expression. For example, consider the following:

```
mov.l  #extern_1+extern_2,r1;this cannot be converted
```

The work around is to change the code to:

```
mov.l  #extern_1,r1
mov.l  #extern_2,r2
add    r2,r1
```

Groups

All groups are discarded completely during conversion. This is becuase GNU has no concept of groups in GNU. It may be necessary to pass on some of the group attributes to their relevant sections at a later date. This means that assigning ORGs etc. to groups has no meaning. Register setting information, and file information is also discarded.

SNASM2 Specifics

Various facilities within SNASM2 will no longer work and will cause conversion to fail. OBJLIMIT and OBJBASE objbase are resolved as special expressions, and therefore cannot be converted.

Also, when a complex expression appears in a piece of source such as

```
mov.l  #objlimit,r1
...
lits
```

then the error is reported as being on the ‘LITS’ line. This is actually correct, as this is where the complex expression is actually generated by the assembler. At the time of writing there is nothing that can be done about this message. The information about the original line is not present.

This is the only information this page contains.

Appendix

Hitachi Assembler Compatibility

A Hitachi Assembler Compatibility

A.1 Introduction

This appendix describes the differences between the Hitachi syntax and the syntax supported by the SNASM2 assembler using the HITACHI.MAC compatibility file.

Where differences exist between Hitachi and SNASM2 syntax, cross references to the relevant places the rest of this manual. Where no reference is made to Hitachi syntax it can be assumed the syntax is fully supported by the SNASM2 assembler

To use Hitachi syntax first make sure that the HITACHI.MAC file is in the same directory as the SNASM2 SH2 assembler and then specify the `[-|/]hitachi` command-line switch each time you invoke the assembler.

Note This appendix does not attempt to teach you how to use the SNASM2 assembler and assumes that you are familiar with the Hitachi assembler and its syntax.

For information concerning the Hitachi syntax see the supplied SH Series Cross Assembler User's Manual (P/N SH0700ASCU1SE).

A.1.1 Using Hitachi Syntax

See also section 3.1, "Command-line Use".

The SNASM2 assembler provides support for Hitachi syntax both in the assembler itself and via a compatibility file, HITACHI.MAC. The SNASM2 assembler provides support for e.g. the Hitachi integer constant syntax e.g. H'123 for hexadecimal numbers.

The HITACHI.MAC file contains a set of macro definitions and aliases for SNASM2 directives. To use the facilities in the HITACHI.MAC file, use the command-line switch: `[-|/]hitachi`; this causes the SNASM2 assembler to pre-read the Hitachi macros.

A.1.2 Porting Hitachi Code to SNASM2

Complete projects built using the Hitachi assembler and linker will need some source level changes to allow them to be built using the SNASM2 assembler and to generate a new SNASM2 link file. The compatibility features aim to minimise the changes needed to source files that have been successfully assembled using the Hitachi tools; they do not attempt to provide 100% compatibility.

A.2 Overview of Syntax Differences

There are three major areas where the syntax of the Hitachi and SNASM2 assemblers differ: Automatic literal pool generation; the line continuation character; and conditional assembly functions. These areas are now described in turn.

A.2.1 Automatic literal pool generation

The Hitachi assembler automatically emits the literal pool after the slot instruction following a BRA, JMP, RTS or RTE instruction. The SNASM2 assembler does not emit the literal pool until explicitly requested to do so. To emit the literal pool issue a .LITS directive immediately after the slot instruction of a BRA, JMP, RTS or RTE instruction. Note that .LITS will always emit the literal pool regardless of where it is invoked from.

A.2.2 The Line Continuation Character

See also Section 4.2, “Statement Format”

In the Hitachi assembler, long lines can be continued on the next source line by placing a ‘+’ character as the first character on the continuation line. In contrast, SNASM2 requires a ‘&’ character as the last character of a line that is to be continued.

A.2.3 Conditional Assembly Functions

See also Section 6.3.2, “Conditional Assembly (IFxx) Macros”

The Hitachi assembler uses a preprocessor to handle the conditional assembly functions. The preprocessor requires that all variables used in such constructs to be prefixed with ‘\&’. In contrast, the SNASM2 conditional assembly functions are part of the assembler itself and so variables used in them do not need any prefix.

A.3 Program Elements

This section describes in detail the differences between Hitachi and SNASM2 syntax. Where no reference is made to Hitachi syntax it can be assumed the syntax is fully supported by the SNASM2 assembler.

A.3.1 Continuation Lines

Continuation lines started with '+' are not supported by the SNASM2 assembler. Hitachi source statements must be changed so that either they do not require continuation lines or the '&' character placed at the end of a line to signify that the next line is a continuation of the previous one. The maximum length of a source statement in SNASM2, including all continuation lines is 1024 characters. Note also that in SNASM2, comments cannot be embedded in a source line that is to be continued. For example, the following Hitachi source statements:

```
      .DATA.L    H'FFFF0000
+                H'FF00FF00    ; Comments allowed here
+                H'FFFFFFFF
```

should be recoded as

```
      .DATA.L    H'FFFF0000 &
                 H'FF00FF00 &
                 H'FFFFFFFF
```

or, to allow the use of comments, as

```
      .DATA.L    H'FFFF0000
      .DATA.L    H'FF00FF00    ; Comments allowed here
      .DATA.L    H'FFFFFFFF
```

A.3.2 Reserved Words

The STARTOF and SIZEOF operators are not implemetend.In SNASM2, use the SECT and SECTSIZE functions to obtain a sections' start address and size respectively. For more information, see "Expressions" below.

A.3.3 Coding of Symbols

Do not use the '$' character in symbols.

A.3.4 Expressions

Exclusive OR Operator

In SNASM2, the Exclusive OR (XOR) operator is represented by the ‘^’ character; Hitachi syntax uses the ‘~’ character.

STARTOF and SIZEOF

See also Section 7.3.7, “Section Functions”

In SNASM2, the Hitachi STARTOF and SIZEOF operators are available as the SECT and SECTSIZE functions respectively. Source statements containing the STARTOF operator, such as:

```
.DATA.L    startof code
```

should be recoded as

```
.DATA.L    sect(code)
```

Similarly, statements containing the SIZEOF operator, such as:

```
.DATA.L    sizeof code
```

should be recoded as

```
.DATA.L    sizeof(code)
```

A.3.5 Sections

See also Section 7.3, “Sections”

The SNASM2 assembler does not support the common section facility. Programs using this facility will need to be recoded to run under SNASM2.

The HITACHI.MAC compatibility file provides facilities to support all variants of the. SECTION directive with the exception of the COMMON section type.

The SNASM2 assembler does not validate the constructs allowed in each of the section types.

In DUMMY sections, which the Hitachi assembler uses to define structures, .RES directives are redefined to generate the SNASM2 RS constructs. No checks are made to ensure that no other assembler constructs are used inside DUMMY sections.

A.3.6 The .REG directive

In SNASM2 syntax, do not parenthesise the register name operand in the .REG directive.

A.3.7 Data Definition and Reservation.

DC.L

See also Section 5.5.1, “DC”

HITACHI.MAC replaces the default SNASM2 DC.L directive with one that aligns the data being defined on a long word boundary.

Out of Range Parameters for DC, DCB, DB and DW

The SNASM2 assembler has a *Truncate* option that, when On (`t+`), will truncate out of range parameters for the DB, DC, DCB and DW directives. If this option is not enabled (`t-`, the default), out of range parameters for these directives will generate an error.

Maximum Data Size for .DATB

The SNASM2 assembler has a command-line switch that controls the maximum data size that can be generated by a single data definition directive. The purpose of this switch is to prevent coding errors generating vast amounts of data. The syntax is:

`[-|/]dmax` *Num*

Num is in the range 1-32 where `dmax`=2^{Num}. By default *Num* is set to 16 (i.e. `dmax`=65536) allowing up to 64K of space to be reserved by one data definition statement. The assembler will generate an error if the size exceeds 2^{dmax}.

Appending Control Characters to Strings

In SNASM2 syntax, do not use angle brackets to enclose a control character when appending it to a string in, for example, the .SDATA directive. For example, the following Hitachi syntax statement:

```
.SDATA     "abab"<H'07>
```

should be recoded using SNASM2 syntax as follows:

```
.SDATA     "abab",H'07
```

A.3.8 Object Module Assembler Directives

.OUTPUT

See also Section3.1, "Command-line Use"

The SNASM2 assembler does not support the Hitachi .OUTPUT directive and its use will cause the assembler to generate an error. In SNASM2, specification of the output file is performed on the command-line when invoking the assembler.

.DEBUG

The SNASM2 assembler does not support the Hitachi .DEBUG directive and its use will cause the assembler to generate an error. In SNASM2, use the `[-|/]sdb` command-line switch to cause source debug information to be output.

A.3.9 Assembly Listing Directives

.PRINT, .LIST, .FORM, .HEADING, .PAGE, .SPACE

The .PRINT, .LIST, .FORM, .HEADING, .PAGE and .SPACE directives are not supported and generate errors if they are encountered.

A.3.10 Object Module Name Setting

.PROGRAM

The .PROGRAM directive is not supported; the name of the module or object file is specified on the SNASM2 command-line.

A.3.11 File Inclusion Function

In SNASM2, if a filename is specified with a root name but no extension the assembler will search for a file of that name.

If such a file cannot be found the assembler will search for a file with the specified root name and one of the extensions given below in the following order.

```
.ASM
.S
.COF
.O
.LIB
```

The SNASM2 assembler will use the first file found; the extension specifies the type of file.

A.3.12 Conditional Assembly Functions

Prefixing Conditional Assembly Functions

See also Section 6.3.2, “Conditional Assembly (IFxx) Macros”.

The Hitachi assembler uses a preprocessor to implement conditional assembly functions; this requires preprocessor symbols to be prefixed with ‘/&’. In SNASM2, conditional assembly is implemented in the assembler removing the requirement for such prefixes.

The SNASM2 assembler, via the HITACHI.MAC file, translates the Hitachi conditional assembly directives but will not rename any prefixed symbols used in such constructs; these must be recoded to use unique symbols not prefixed with ‘/&’.

.ASSIGNA and .ASSIGNC

The Hitachi .ASSIGNA and .ASSIGNC directives are not implemented and will cause the assembler to generate an error if used.

A.3.13 Macro Function

See also Section 6, “Macros”.

The macro facilities of the Hitachi and SNASM2 assemblers are similar but there are two Hitachi features that SNASM2 does not support: the ability to specify a default value for macro parameters; and the ability to specify parameter values by name. Using either construct will cause the SNASM2 assembler to generate an error.

A.3.14 Character String Manipulation Functions

.LEN, .INSTR, and .INSTR

See also Section 5.11, “Manipulating Strings”.

The Hitachi macro character string manipulation functions .LEN, .INSTR and .SUBSTR are supported with different syntax in SNASM2. The Hitachi .LEN, .INSTR and .SUBSTR functions should be recoded using the SNASM2 functions STRLEN, INSTR and SUBSTR respectively.

A.3.15 Automatic Literal Pool Generation Function

SNASM2 treats literals and literal pools in a similar way to the Hitachi assembler.

Literals that can be evaluated and whose values are within the available bounds are placed in the instruction itself. Literals whose value is not within the available bounds are placed in the next literal pool.

Literals which do not evaluate, because they forward reference a symbol for example, are placed in the literal pool unless the literal is introduced with a ‘##’. In this case they are forced into the instruction and will generate an error if the eventual literal value exceeds the bounds available.

Automatic Emission of the Literal Pool

The Hitachi assembler automatically emits the literal pool after the slot instruction following a BRA, JMP, RTS or RTE instruction. The SNASM2 assembler does not emit the literal pool until explicitly requested to do so. To emit the literal pool issue a .LITS directive immediately after the slot instruction of a BRA, JMP, RTS or RTE instruction. Note that .LITS will always emit the literal pool regardless of where it is invoked from.

.POOL

The Hitachi .POOL directive is supported via a macro definition and causes a branch, a NOP and the current literal pool to be output. Note that is the literal pool is empty a branch and a NOP will still be generated by the SNASM2 assembler.

For those Hitachi directives not supported , the HITACHI.MAC compatibility file generates an error using the _.MSG1 macro. If desired, this macro can be edited so that it issues a warning instead.

Index

Symbols

* 4-26, 4-27, 6-8
@ 4-26, 4-27
_ 6-10
\# 4-31
\$ 4-31
* 6-8
\@ 6-10
\0 6-10

A

Addressing modes 4-36
ADDRMODE function 4-36
  addressing modes 4-36
ALIAS directive 5-3
.ALIGN directive 5-27
ALIGNMENT function 4-37, 7-13
Assembler
  command file 3-13
  command-line syntax 3-3
  filenames 3-6 to 3-7
    default extensions 3-6, 3-7
    ignoring specified files 3-8
  quirks 3-12
  running 3-3
  Running from within an editor 2-2
  switches 3-8 to 3-11
.ASSIGN directive
  See SET directive

B

Breakpoints 8-57 to 8-66
  clearing 8-57
  conditional 8-60
  configuring 8-57
  as counters 8-60
  halting 8-61
  logging 8-61
  setting 8-57
  single step 8-65
  step into 8-66
  step over 8-65
  suspending 8-60
  tracing 8-65
  unconditional 8-60
  unstep 8-66
Breakpoints window 8-53

C

CASE and ENDCASE directives 5-46
CNOP directive 5-28
Code windows 8-30 to 8-37
Command file
  linking 5-76, 5-86
Conditional assembly 5-41 to 5-54
  CASE and ENDCASE directives 5-46
  DO and UNTIL directives 5-53
  .END directive
    See END directive
  END directive 5-42
  IF...ELSE...ELSEIF and ENDIF directives 5-43
  IFxx macros 6-16
  REPT and ENDR directives 5-48
  WHILE and ENDW directives 5-51
Constants 4-21 to 4-29
  assembly location counter. *See* current location counter
  assembly time 4-25
  character 4-24
  current location counter 4-26
  integer 4-22
  pre-defined 4-27 to 4-29
  strings 4-30
  turning numbers into strings 4-31
_CURRENT_FILE 4-28
_CURRENT_LINE 4-28
Current location counter 4-26

D

Data
  defining 5-16 to 5-24
  defining initialised 5-16 to 5-23
    DATA directive 5-22
    DATASIZE directive 5-21

DCB directive 5-18
DC directive 5-16
HEX directive 5-20
IEEE32 directive 5-23
IEEE64 directive 5-23
out of range parameters 5-19
reserving space
DS directive 5-24
DATA directive 5-22
DATASIZE directive 5-21
_DAY 4-28
DCB directive 5-18
out of range parameters 5-19
DC directive 5-16
out of range parameters 5-19
Debugger
about 8-1
Breakpoints window 8-53
Code windows 8-30 to 8-37
command-line syntax 8-2
Disassembly window 8-36
exiting 8-19
File Viewer window 8-55
interface 8-13 to 8-15
Log window 8-54
Main window 8-16 to 8-29
Execution menu 8-25
File menu 8-17
Session menu 8-20
Target menu 8-22
Windows menu 8-29
Memory window 8-40 to 8-41
Mixed window 8-31
Program window 8-48
Registers window 8-38
running 8-2 to 8-12
session files 8-8
memory ranges 8-9
window attributes 8-11
Source window 8-37
target
discarding 8-22
monitoring 8-24
resetting processor 8-27
restoring registers 8-27
saving registers 8-27
selecting 8-13
stopping 8-27
updating 8-23
Watch window 8-46
structure browsing 8-47
windows
working with 8-14
DEF function 4-37
Directives
changing names 5-3
See also individual entries
DISABLE directive 5-3
Disassembly window 8-36
DO and UNTIL directives 5-53
DS directive 5-24

E

.END directive
See END directive
END directive 5-42
ENDM directive 6-2
Equates 5-5 to 5-15
.ASSIGN directive
See SET directive
.EQU directive
See EQU directive
EQU directive 5-5
EQUR directive 5-13
EQUS directive 5-7
permanent 5-5
forward references 5-6, 5-21, 5-24, 5-28
redefinable 5-6
.REG directive
See EQUR directive
REG directive 5-15
register 5-13 to 5-15
RS 5-10 to 5-13
SET directive 5-6
string 5-7
.EQU directive
See EQU directive
EQU directive 4-19, 4-25, 5-5
EQUR directive 4-19, 5-13
EQUS directive 4-19, 5-7
Errors

FAIL directive 5-71
INFORM directive 5-70
user generated 5-70 to 5-71
EVEN directive 5-26
.EXPORT directive
See EXPORT directive
EXPORT directive 5-73
Expressions 4-33 to 4-43, 8-67 to 8-78
C++ 8-67
format specification 8-73
format specifier character 8-74
formatting 8-72
functions 4-36 to 4-43
operator precedence 4-34
pointer modifier 8-75
repeat modifier 8-78
width modifier 8-76

F

FAIL directive 5-71
_FILENAME 4-28
Files
including 5-34 to 5-38
INCBIN 5-37
INCLUDE directive 5-34
FILESIZE function 4-37
File Viewer window 8-55
Forward references
permanent equates 5-6, 5-21, 5-24, 5-28
Functions 4-36 to 4-43
ADDRMODE 4-36
addressing modes 4-36
ALIGNMENT 4-37
DEF 4-37
FILESIZE 4-37
INSTR 4-40
INSTRI 4-40
LINKEDSIZE 4-40, 7-14, 7-20
NARG 4-40
OBJBASE 4-38, 7-13, 7-20
OBJLIMIT 4-38, 7-14, 7-20
OFFSET 4-40
ORGBASE 4-38, 7-13, 7-20
ORGLIMIT 4-38, 7-14, 7-20
REF 4-41
for sections and groups 7-4
SIZE 4-39, 7-14, 7-20
SQRT 4-41
STRCMP 4-41
STRICMP 4-41
STRLEN 4-41
TYPE 4-42

G

.GLOBAL directive
See GLOBAL directive
GLOBAL directive 5-76
Groups 7-1
alignment of 7-18
associated directives
summary 7-3
associated functions
summary 7-4
attributes of 7-15
current size
See SIZE function
defining 7-3
functions 7-20
initialised 7-2
linked size
See LINKEDSIZE function
logical end address
See OBJLIMIT function
logical starting address
See OBJBASE function
overlaying 7-19
physical end address
See ORGLIMIT function
physical starting address
See ORGBASE function
and sections 7-1 to 7-20
starting address of 7-16
uninitialised 7-2
writing to file 7-18

H

HEX directive 5-20
_HOURS 4-28

I

IEEE32 directive 5-23
IEEE64 directive 5-23
IF...ELSE..ELSEIF and ENDIF directives 5-43
IFxx macros 6-16
.IMPORT directive
  See IMPORT directive
IMPORT directive 5-74
INCBIN directive 5-37
INCLUDE directive 5-34
INFORM directive 5-70
INSTR function 4-40, 5-56
INSTRI function 4-40, 5-56

L

Labels 4-16 to 4-20
  and symbols 4-16 to 4-20
  local 4-16, 4-17
    in macros 6-21
    scope 4-17, 4-19
LINKEDSIZE function 4-40, 7-14, 7-20
Linking 5-72 to 5-86
  command file 5-86
  .EXPORT directive
    See EXPORT directive
  EXPORT directive 5-73
  .GLOBAL directive
    See GLOBAL directive
  GLOBAL directive 5-76
  guide to 5-76
  .IMPORT directive
    See IMPORT directive
  IMPORT directive 5-74
  PUBLIC directive 5-75
LIST directive 5-32
Listings 5-32 to 5-33
  LIST directive 5-32
  NOLIST directive 5-32
Literal pools 4-6, A-9 to A-10
LOCAL directive 6-21
Local labels 4-16, 4-17
  in macros 6-21
  scope 4-17, 4-19
  scope in modules 5-58
Log window 8-54

M

MACRO directive 6-2
Macros 6-1 to 6-25
  advanced features 6-17 to 6-25
    extended parameters 6-17
    local lables 6-21
    PUSHP and POPP directives 6-23
  defining 6-2
    ENDM directive 6-2
    MACRO directive 6-2
  editor macros for SNMAKE 9-2
  ENDM directive 6-2
  expanding 6-3
    MEXIT diective 6-3
  IFxx macros 6-16
  importing labels 6-8
  introducing 6-2 to 6-4
  invoking 6-3
  local labels in 6-21
    LOCAL directive 6-21
  MACRO directive 6-2
  MACROS directive 6-14
  memory management 6-24
  MEXIT directive 6-3
  parameters 6-5 to 6-13
    extended 6-17
    labels as 6-8
    named 6-6
    numbered 6-5
    special 6-10
    variable numbers of 6-7
  PURGE directive 6-24
  short macros 6-14 to 6-16
    MACROS directive 6-14
MACROS directive 6-14
Make file. See Project file
Memory ranges 8-9
Memory window 8-40 to 8-41
MEXIT directive 6-3
_MINUTES 4-28
Mixed window 8-31
MODEND directive 5-58
MODULE directive 5-58
Modules 5-58 to 5-61

MODEND directive 5-58
MODULE directive 5-58
scope of local labels in 5-58
_MONTH 4-28

N

NARG function 4-40
NARG symbol 4-28, 6-7, 6-17
NOLIST directive 5-32
Numbers
turning into strings 4-31

O

OBJBASE function 4-38, 7-13, 7-20
OBJ directive 5-29
OBJEND directive 5-29
OBJLIMIT function 4-38, 7-14, 7-20
OBJLMIT function 7-20
OFFSET function 4-40, 7-12
Operator precedence 4-34
OPT directive 5-68
Optimisations 5-67
changing in source code 5-69
OPT directive 5-68
PUSHO and POPO directives 5-69
setting in source code 5-68
Options 5-62 to 5-65
changing in source code 5-69
setting in source code 5-68
ORGBASE function 4-38, 7-13, 7-20
ORG directive 5-25
ORGLIMIT function 4-38, 7-14, 7-20

P

POPO directive 5-69
POPP directive 6-23
POPS directive 7-10
Program counter
changing 5-25 to 5-29
.ALIGN directive 5-27
CNOP directive 5-28
EVEN directive 5-26
OBJ directive 5-29
OBJEND directive 5-29
ORG directive 5-25
Program window 8-48
Project files 2-1, 9-1, 9-3 to 9-11
comments 9-8
creating 9-3
defining dependencies 9-6
defining explicit rules 9-6
defining implicit rules 9-7
defining targets 9-4
implicit targets
defining rules for 9-8
line continuation 9-8
macros 9-9
special targets 9-4
PUBLIC directive 5-75
PURGE directive 6-24
PUSHO directive 5-69
PUSHP directive 6-23
PUSHS directive 7-10

Q

Quirks 3-12

R

_RADIX 4-27
_RCOUNT 4-27
REF function 4-41
.REG directive
See EQUR directive
REG directive 5-15
Registers window 8-38
REGS directive 5-39
REPT and ENDR directives 5-48
REPT directive 5-48
__RS variable 4-27, 5-10
RS directive 5-10
RSRESET directive 5-10
RSSET directive 5-10

S

_SECONDS 4-28
SECT function 7-12
SECTION directive 7-5
Sections 7-5 to 7-20

ALIGNMENT function 7-13
alignment of 7-7
allocating to groups 7-8
and groups
introduction to 7-2 to 7-4
associated directives
summary 7-3
associated functions 7-14 to 7-20
summary 7-4
attributes of 7-8
base address of 7-12
changing between 7-10
current size
See SIZE function
Fragments 7-12
and groups 7-1 to 7-20
linked size
See LINKEDSIZE function
logical end address
See OBJLIMIT function
logical starting address
See OBJBASE function
name of 7-6
OFFSET function 7-12
offset of symbols in 7-12
physical end address
See ORGLIMIT function
physical starting address
See ORGBASE function
SECT function 7-12
symbol offset from alignment of 7-13
Session files 8-8
memory ranges 8-9
window attributes 8-11
SET directive 4-19, 5-6
SHIFT directive 6-7, 6-17
SIZE function 4-39, 7-14, 7-20
SNASM2
main menu 2-2
SNLIB utility 10-1
running 10-1
command-line syntax 10-1
SNMAKE utility 9-1
command-line syntax 9-12 to 9-13
switches 9-12
editor macros for 9-2
project files 9-3 to 9-11
Source window 8-37
SQRT function 4-41
STRCMP function 4-41, 5-55
STRICMP function 4-41, 5-55
Strings 4-30
comparing 5-55
determining the length of 5-55
equating substrings to a symbol 5-57
INSTR function 5-56
INTRI function 5-56
manipulating 5-55 to 5-57
STRCMP function 5-55
STRICMP function 5-55
STRLEN function 5-55
SUBSTR function 5-57
sub-strings 5-56
STRLEN function 4-41, 5-55
Structure browsing 8-47
SUBSTR function 5-57
Symbols
and periods 4-20
Syntax
source code 4-1
statement format 4-13

T

Target
discarding 8-22
monitoring 8-24
registers
restoring 8-27
saving 8-27
resetting processor 8-27
selecting 8-13
setting parameters for 5-39
REGS directive 5-39
stopping 8-27
updating 8-23
Tracing 8-65
single step 8-65
step into 8-66
step over 8-65
unstep 8-66
TYPE function 4-42

U

Utilities
SNLIB 10-1
SNMAKE 9-1

W

Warnings
FAIL directive 5-71
INFORM directive 5-70
user generated 5-70 to 5-71
Watch window 8-46
Structure browsing 8-47
_WEEKDAY 4-28
WHILE and ENDW directives 5-51

Y

_YEAR 4-27