SEGA STUDIOS

CROSS
PRODUCTS

LIMITED

SNASM68K

8/8/91

CROSS PRODUCTS LIMITED
23 THE CALLS
LEEDS
LS2 7EH
Tel: 0532 429814
Fax: 0532 426163

Contents

Introduction

The Cross Products 68000 cross assembler system is a high performance 68000 assembly language development system running on MS-DOS and PC-DOS computers.

Components of the System

Assembler

SNASM68K converts programs written in 68000 assembly language into either absolute code which can be sent directly to the target machine, CPE (Cross Products Executable) files or object modules suitable for linking.

Debugger

SNBUG68K allows 68000 programs running on the target machine to be remotely debugged from the development PC. It has access to the symbols used in the source code allowing full symbolic debugging.

Linker

SNLINK combines object modules created by SNASM68K, evaluates any expressions which could not be resolved earlier. It can generate output in a large variety of formats including direct transmission down the SCSI link.

Librarian

An object module library is a collection of object modules that reside in one file. SNLIB is a utility program that builds and maintains object module libraries for the linker.

Hardware Link

To speed program development, the development PC is connected to the target machine using a SCSI (small computer systems interface) bus. The assembler, linker and debugger use this interface to send code to the target machine.

Environment

A program can be assembled and debugged without exiting the Brief editor which greatly speeds the Edit/Assemble/Debug cycle.

Ways of using SNASM

SNASM is an extremely flexible system that can be used in many different ways. A few of these approaches are given here to give an overview of the product.

Absolute code without sections, no linking

The code is be assembled at an absolute address, specified with one or more ORG directives, and then either sent directly to the target machine, placed into a CPE file or output in pure binary format.

Absolute code with sections, no linking

The ORG address of the program must be specified before any sections are opened. If groups are not used BSS type sections are not available, and sections are placed in memory in the order they occur. If groups are used, they define the ordering of sections in memory, and allow BSS type and absolute word addressable sections.

Linking

Each source file is assembled using the /L switch to produce an object module. Source modules cannot contain ORG directives. The modules are linked to produce any of the output types supported by the assembler plus some machine specific relocatable formats.

There is no need to use sections or groups when producing linkable output, but many more functions are available if they are used.

Hardware Requirements

SNASM68K requires the following minimum configuration :-

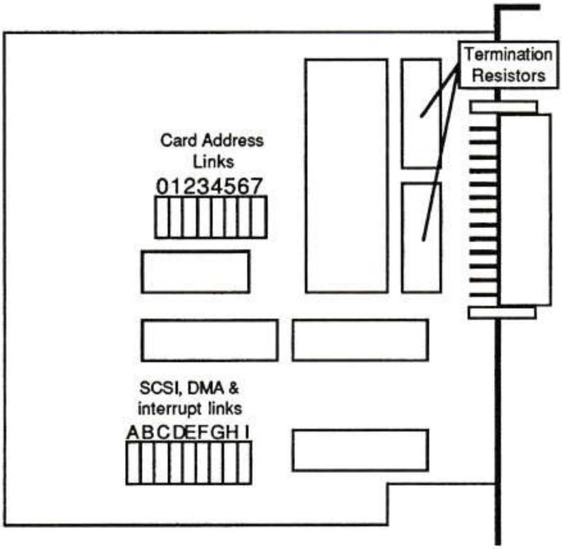
- IBM-PC or compatible computer
- One free 8 or 16 bit slot
- 256K of memory
- MS-DOS 2.0 or later
- A hard disk

we strongly recommend :-

- 8MHz IBM PC/AT or faster
- A fast hard disk
- 640K of main memory and a 1Mb RAM disk
- DOS 3.0 or above
- VGA graphics

Hardware Installation

Configuring the PC card



The card can reside at one of several addresses in the PC's port map. The links labelled 0-7 are used to set the address as follows :-

Link	Address
0	300-307
1	308-30F
2	310-317
3	318-31F
4	380-387
5	388-38F
6	390-397
7	398-39F

The card is supplied with link 6 connected so it is at address 390 hex.

The SCSI bus supports up to eight devices, each of which must have a different SCSI device number.

The links labelled A-I are used to set the SCSI device number of the card and the DMA and Interrupt channels the card will use.

Link	Function	
A	IRQ5	
B	IRQ7	
C	DRQ1	
D	DACK1	
E	DRQ3	
F	DACK3	
G	Bit 2	SCSI ID of card
H	Bit 1	
I	Bit 0	

Interrupts are not currently used so links A and B should be left unconnected. The card is supplied with links C and D connected so the card uses DMA channel 1. If you want to use DMA channel 3 move both jumpers to E and F, or if you don't want to use hardware DMA at all remove both links.

The card is supplied with links G and H connected giving the card a SCSI device number of 6. This will only need changing if it clashes with another device on the SCSI bus.

Fitting the Card

1) Verify that no power is being supplied to the computer.
2) Remove the computer's case.
3) Check that the address and DMA settings of the card will not cause it to clash with any other cards in the computer.
4) Locate an empty slot and remove the backplate cover.
5) Install the SCSI card by firmly but carefully pressing it down into the slot.
6) Secure the card with the screw you removed from the backplate.
7) Replace the computer's case.

If you changed the settings of the card make sure you know the address and DMA channel you selected as you will need this information to complete installation.

Software Installation

Note: The SNASM install program is designed to customise the Brief text editor's environment to enable SNASM to work from within Brief. If you intend to work in this manner you should install Brief **before** you install SNASM.

The install program provided on your SNASM distribution disk will automatically install the whole SNASM system . (All queries from the program asking for user input are terminated "[y/n]". Hitting the ESCAPE key at this prompt will abort immediately.)

Install will ask for the following information:

1. Install first checks to see if it can find the default directory to copy the SNASM executable files to. This is C:\SNASM. Install will give the user the option to specify any other directory.

2. The install program will now ask for the system boot drive and check that it can find an autoexec.bat file there. It will persist in asking for this information until it can find an autoexec.bat file.

3. Next it requires the name of the disk drive in which the SNASM distribution disk is residing.

4. The install program will now ask if the user requires to backup old files. If the answer is yes any files in the target directory of the same name as a new one about to be copied onto it will be saved (in that directory) with their file extension changed to ".old". This option takes effect throughout the install process and if it is not selected the copy process will stop for every file to ask "filename exists Overwrite/Backup/overwrite All[O/B/A] "

These options have the following effects:

O: overwrite this file repeat query for next file.
B: save old file with ".old" extension repeat query.
A: overwrite all files of same names without asking
(throughout the whole of the install process).

5. Should Install detect that the directory holding the executable files is not on the search path of the users machine it will ask if it should add it.

6.Install will next look for the BPATH and BHELP environmental variables to determine where to copy the Brief macros and menu files to. It will display a message asking if this is desirable: as Brief expects to find them in this location it is advisable to do so.

7. Before editing the autoexec.bat file, Install will ask the user if a backup should be created(as autoexec.old). It will then proceed to set several flags within the Brief environmental variables and add the command to install the SCSILINK software on boot-up. Install will always ask before editing any

portion of the autoexec.bat file.

8. If Install leaves a message on exit saying that the old SNASM macro start-up code is present in the users 'startup.cb' macro file this is due to a change in the manner in which the SNASM Brief macros are initiated on starting an editing session. The line 'load_macro("snasm")' should be removed from the file startup.cb (residing in the directory pointed to by the BPATH environmental variable - \brief\macros by default) and the file re-compiled.

The communications between the PC and the target machine are handled by a program called SCSILINK which must be installed before any of the other programs can be run. Install will add a line to do this your autoexec.bat file. The information below will be of use if you wish to install it at a non-default address.

Usage:

```
        SCSILINK address[,D?][,I?]
```

the default is

```
        SCSILINK 390,D1
```

The address is whatever was set using links 0-7 on the SNASM PC card and the number after the D should be the DMA channel you are using.

If no DMA channel is specified hardware DMA will not be used and unless you have a fast computer data transmission speed will be reduced.

The I parameter is used to give the card a different SCSI device number without opening the computer and changing the links, however this is not often required.

SCSILINK checks for the existence of the PC card as it installs itself, and reports any errors at this time.

9.You should re-set your computer when Install finishes to benefit from the new SNASM environment.

Running the Assembler

If you intend using SNASM68K as a stand alone assembler instead of running it from within BRIEF you must be familiar with the command line syntax.

Command line Syntax

```
SNASM68k /switches sourcefile,objectfile,symbolfile,listfile,tempfile
```

Switches	/b number	Set size of internal buffers (Kbytes 2-64)
	/c	enable listing of lines that are in failed conditional structures
	/d	Debug mode - do not run
	/e symb=value	Equate symbol to value
	/g	Write non-global symbols to linker object file
	/i	Display information window during assembly
	/j path	specify include file search path
	/k	Allow use of ifeq etc.
	/l	produce linkable output file
	/m	Expand macros in listings
	/o opts	Set assembler options/optimisations
	/p	Produce pure binary instead of .CPE
	/w	write equates to symbol/linker file
	/z	include line number info. in linker file

See the section on Options later in this manual for a full list of the assembler options and optimisations.

Sourcefile File containing 68000 source to be assembled. If this is omitted SNASM68K will print the above information in a help message.If no extension is given for the source file it is assumed to be .68K.

Objectfile File to receive object code output. If this is omitted no object code will be generated. If you want the object code to be sent to a target computer use T?: as the objectfile where ? is the SCSI device number of the target.

Symbolfile File to receive symbol table information for debugger.

Listfile File to receive any listing output.

Tempfile File to be used for any data that will not fit in memory whilst assembling. If this is omitted the file will be called SNASM.TMP or a unique name if your DOS version is high enough.

e.g.

```
SNASM68K /o ow+,an+,e+ test.68k,t7:,test.sym,,e:
```

Assembles test.68k with warnings reported, alternate numerics enabled and error line printing suppressed. Object code will be sent to the target with SCSI device number 7. No listing will be generated, symbols will be written to test.sym and the temporary file will be generated on disk E:.

Each time SNASM68K is run it checks for an optional environment variable called SNASM68K which contains default switches, options and filespecs.

eg.

```
set snasm68k=/c- ,t7:,,,e:\tmp
```

Disables case sensitivity, makes default output go to target 7 and sets the default temporary file to be e:\tmp. Anything set in the environment variable can be over-ridden on the command line.

Assembly can be stopped by pressing Control-C, or if you are assembling from within Brief and you are using the information window, by pressing Esc.

If the first item on the command line starts with an at sign (@) it is taken to be the name of a file containing the assembler options.

eg.

```
SNASM68K @snasm.cmd
```

where snasm.cmd contains :-

```
/j  c:\includes
/e z=1
test.68k,
t7:,
,
test.lst
```

will act as -

```
SNASM68K /j c:\include /e z=1 test.68k,t7:,,test.lst
```

Running from within Brief

1. The Brief editor macros that interface SNASM to the editor have been revised to work with the Cross Products make utility SnMake and hence provide a better working environment for the SNASM user. In practical terms this means that within Brief the dialogue-box interface to SNASM has been completely scrapped and now relies on using SnMake and its associated 'project files' for the setting up and executing of external commands. Whilst this involves slightly more work in initial setting-up it offers longer-term time savings and much more flexibility.

Using SnMake is detailed in the section 'Using SnMake' and will not be gone into in great detail here. The SNASM environment now relies on the creation of a 'project file' for each project the user is undertaking. These are files utilised by SnMake which hold information about the file dependencies for that project and the rules by which the output file(s) can be created according to those dependencies. (See 'Using SnMake'). This means that commands and their options are put into the project file as they would be entered from the command line, instead of being set up in the dialogue-box as previously.

The Snasm main menu is invoked by pressing Alt-F9. The menu options that are available are discussed in sequence below. Navigation through the menus is done using the up and down arrow keys, select items using the return key, escape will exit all menus.

Make. Keyboard equivalent Alt-F10.

This option invokes SnMake on the currently selected project file.

Select Project File. Keyboard equivalent Ctrl-F9.

When this is selected it brings up a window showing all the files with a '.prj' extension in the current directory, if one is currently selected it will be highlighted. These are the project files. The first line of the project files is displayed in this window as an aid to the users memory if multiple project files are present in the current directory. See the sub-menu item 'Show comment lines' for more information on putting such comments into project files. Selecting one of these filenames will bring up a sub-menu offering four options.
1. **Select this file.** Selecting this option make the highlighted project file 'current', i.e. invoking the make option will start SnMake with the highlighted project file.

2. **Select and Make.** Make the highlighted project file current and invoke the make utility at the same time.

3. **Show comment lines.** This displays all the text in the project file from the top of the file to the [SnMake] label. This is a required label in a project file and SnMake does not begin to parse the file for input until it encounters this label. Thus any text can be placed before this label without causing an error.

4. **Edit this file.** Edit the highlighted project file.

Note: only sub-menu options 1 and 2 alter the setting of the current project file.

Debug. Keyboard equivalent Ctrl-F10.

This option enters the debugger specified in the current project file.(See 'Using SnMake' for more information on setting the debugger).

Set Debug Mode. No keyboard equivalent.

Selecting this option brings up a sub-menu with only two options: On and Off, one of which will be highlighted. These options control the setting of the special macro '$!' in the project file. This macro expands to the settings of the debug and info switches on the Snasm command line. Info mode is always on when SnMake is invoked from within Brief, making Snasm and Snlink bring up status windows allowing the user to monitor their progress. Debug mode can be set from this menu option, determining whether the program being made, (assuming it is being downloaded to a target machine) is run immediately (debug mode Off) or waits with the program counter set to the value specified by the user with the 'regs' directive (debug mode - On). It must be stressed that this control only stems from the correct use of the special macro '$!' in the project file. (See 'Using SnMake').

Evaluate. Keyboard equivalent Ctrl-E.

This option invokes the expression evaluator specified in the current project file.(See 'Using SnMake' for more information on setting the expression evaluator). If any text in the current window is highlighted this function will attempt to pass it to the expression evaluator, if not it will prompt for input on the Brief status line.

Jump to label. Keyboard equivalent Ctrl-G.

This option examines the current cursor position and determines if it is on a valid label(SNASM 68000 assembly language syntax). If so it jumps to that label, if not it prompts for a label name to look for.

Undo last label. Keyboard equivalent Ctrl-F.

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

Save all buffers. Keyboard equivalent Alt-S.

This option saves all buffers currently being edited.

Error Window. Keyboard equivalent Ctrl-Q.

This option opens an error window to display the current contents of the error file 'errors.err' 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(i.e. similar to Snasm and Snlink) the user can step through the errors using the up and down arrow keys. Hitting enter on a highlighted error will take the user to the reported error position in the relevant source file. The 'Home' and 'End' keys can beused 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. Keyboard equivalent 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. On finding no more errors the function will display a message to that effect on the status line.

SCSI errors whilst assembling

If you are sending the object output down the SCSI hardware you may occasionally see an error message and be given the option to Abort, Retry or Bus Reset and Retry. The message will tell you where the error was detected (Locally or by the Target) and what caused the error.

Examples

LOCAL: Target not available
LOCAL: Bus not available

If the error was caused by a power spike or by the target being temporarily unavailable **Retry** will usually get things going again. If this does not work **Bus Reset and Retry** may be effective as it resets all devices on the SCSI bus and reconnects with the target before having another go. **Abort** should be used if neither of the above work, or if the error was caused by something like the target running out of memory, as this causes the assembler to stop assembling and exit.

Debugging

SNBUG68K is a full-featured debugger which uses the SCSI hardware to enable you to remotely debug a program running on your target machine. The debugger has access to symbols within your program and its expression evaluator has all the features of SNASM68K which allows full symbolic debugging.

The SNBUG68K debugger lets you :-

. Trace your code instruction by instruction
. See the values of your variables in a variety of formats
. Change the value of any variable or memory location
. Set breakpoints to stop when an instruction is executed or when
  a certain condition is met.

To make it easier to move between Brief for editing and SNBUG68K for debugging, where possible we have tried to use Brief's control keys and window concepts.

The Link Software

The Link Software usually resides in ROM on the target machine interface and it communicates with the assembler and debugger. To enable the target machine to be debugged the Link Software uses several of the target machines vectors. These are :-

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

The debugger can gain control of the target whenever one of the above exceptions occurs. If your program is running normally none of the above should happen, so to ensure that the debugger can gain control if it requires you should execute a Trap0 instruction at regular intervals. This can, if you want, be within an interrupt routine such as vertical flyback.

Command line Syntax

SNBUG68K /switches symbolfile

The switches can be after the symbol file name if you wish, and you are also allowed to specify several symbol files.

Switches	/sFilename	Load configuration file
	/vExpression	Evaluate expression This is to allow expressions to be evaluated from within Brief, you will not normally use it.
	/eFilename	Load CPE file into target
	/rNumber	Override video bios data screen rows
	/u+ or /u-	Turn continuous update on or off
	/iNumber	Specify rate of update (See Alt-I)
	/lNumber	Set label level
	/tNumber	Set target ID
	/gNumber	Use Hercules graphics drivers
	/h	Halt target at debugger startup

Example

snbug68k /m128 main.sym diskutil.sym

General Concepts

The debugger's display consists of a number of windows of five basic types. These are Register, Disassembly, Memory,Watch and File. Windows can be split to form two windows, the border between two windows can be moved or removed and the type of windows can be changed.

You can have as many Disassembly, Memory, Watch and file windows as you want but you are restricted to only one register window. You can move between windows at will, with the currently active window being highlighted. Within each window the keyboard controls allowe you to change the window's state or send commands to the target machine.

If continuous update mode is on, the windows are refreshed at whatever rate is set for the update interval. If this mode is not enabled windows are only updated when some thing happens, i.e. a key is pressed or a breakpoint occurs.

Register window

The register window displays the contents of all of the processor's general registers. The cursor can be moved around the window and the contents of any of the registers changed either by typing the new value directly or by pressing Return and entering an expression.

Disassembly window

This window displays your code in disassembled format. Where possible symbols are used instead of hex addresses to make the 'code' more readable. If the target's PC is at one of the lines in the disassembly it is marked with a greater-than sign (>). If a breakpoint is set the colour of the line changes and the breakpoint expression and count are displayed after the instruction.

If the debugger has been told the name of the symbol table it replaces numeric values with their symbolic equivalents. Exactly where this is done is controlled using the label level. Local labels will be seen in the disassembly window if they have been enabled at assembly time using the v+/- option.

Level	Effect
0	No symbols except in address field
1	As above plus non-immediate or offset symbols
2	All the above plus long immediate
3	All the above plus offset(An)
4	All the above plus offset(An,Dn)

Memory window

Memory is displayed in hex format as either bytes, words or long words. As in the Register window you can directly enter new data or press return to enter an expression.

Watch window

The Watch window is used to display the contents of memory at an address given by an expression which can, of course, contain symbols. It is used to monitor the value of variables and tables as your program runs.

File Window

This window is used to view text files, usually source code. You can browse around the file and do simple searches, but you cannot make changes to the file.

Configuration Files

When the debugger starts up it gets its initial state from a configuration file called something like -SNBUG68K.CF? where ? is the SCSI ID of your chosen target. The /s switch can be used on the command line to cause a different config file to be loaded. These files can be saved and loaded at any time during the debugging session, so you can have a few of your most useful configurations saved on disk. If a configuration file refers to a symbol which no longer exists then that part of the configuration will be ignored. If a configuration file was saved with the debugger in a video mode with a resolution different to the current one, you will be given the choice of either switching into the new mode, or ignoring the file altogether.

The only entries in the config file that you will usually need to change are those which specify the areas of the targets memory that can be read and written. This information is used to prevent the debugger trying to access memory areas which would cause an address error; if the link to the target goes down when you look at certain memory areas or if some areas appear to contain zeros then these entries need to be changed.

You can edit the config file, using a text editor such as Brief, and change the memory areas under the headings #readram and #writeram, the format is a

[ADDRESS]@W&1=MA

@W=SIZE

simple list of start and end addresses of valid areas, and you can have as few or as many entries as you wish. Please keep the format of the config file identical to the original as otherwise the debugger can have problems reading it.

Prompt History

Whenever the debugger prompts for a reply such as an expression or a breakpoint condition, you can step back through old replies to that prompt using cursor up and down. The prompt line can be edited with cursor left and right, with home, end and backspace performing as you would expect.

Symbol Completion

Whenever you are entering an expression you can get the debugger to try to complete the name of the symbol you are typing using Alt-N. If there are several symbols which start with the text you have entered, then you can cycle through these by pressing Alt-N again.

Expression Evaluator

The debugger uses the same expression evaluator as the assembler with the following changes :-

- The default base is Hexadecimal. Decimal numbers are preceded by a hash (#)
- Processor registers can be used in expressions
- Square brackets around an expression are used to perform indirection. The debugger fetches a byte, word or long word from the target, and sign extends it before using it in the expression. Long words are fetched by default with the @ operator being used to change this as shown below.

Examples

100	100 hex, 256 decimal
#100	100 decimal
[a0]	Long word at A0
[a0+d0.w]@w+a1	Fetch the word from a0+d0, sign extend it and add it to a1

The .b, .w and .l operators which sign extend their operators from the specified length to a long word can clash with dots in label names. If this occurs you can either bracket the expression and put the .w outside the brackets or use \w instead just as in the assembler.

Keyboard Controls

General (work in all windows)

F1	Move to adjacent window
F2	Move a window edge
F3	Split a window
F4	Remove a window edge

Shift-Arrows	Move to adjacent window (These may not work on all keyboards)
Home	Move cursor to the top of the window. In the Register and Watch windows only the cursor moves whereas in the Disassembly and Memory windows the window address is changed so that the item under the cursor moves to the start of the window.
Ctrl-Z	Toggle Full screen display on current window
Shift-F1	Change window type
Ctrl-F2	Restart debugging session (when using a CPE file)
Alt-L	Set window lock expression and turn on lock.
Ctrl-L	Toggle window lock on/off. (Prompts for expression if one hasn't been entered)
Alt-H	Hex calculator
Alt-D	Display creation date of debugger
Alt-X	Exit debugger (Automatically saves current state to SNBUG68K.CF?)
Ctrl-X	Exit without saving state
Alt-Z	DOS gateway. Type 'exit' to return to debugger
Alt-S	Save target registers
Alt-R	Restore saved registers
Ctrl-R	Reset. SSP and PC are loaded from locations 0 and 4, SR is loaded with $2700.
Esc	Halt target machine as soon as possible
Shift-Esc	Halt target machine and turn off interrupts/DMA
Alt-U	Toggle continual update mode on/off. (Default off)
Alt-I	Set interval for continual update in 18ths of a second. (0=Flat out, 2=9 times per second, 18=once per second)
<	Upload bytes from target into a file
>	Download bytes from a file to target memory
Ctrl-F	Memory fill
F10	Save current configuration to a file
Shift-F10	Load configuration file
Alt-0..7	Change target number. The current setup is saved to the config file and a new config file for the new target is loaded.

Register window

Arrows	Move cursor
Return	Change registers value to result of an expression
0-9/A-F	Enter new values at cursor

Memory window

Arrows	Move cursor
Page up/down	Move cursor up/down a page at a time
Return	Change memory value to result of an expression
0-9/A-F	Enter new values at cursor
Alt-G	Goto expression. Changes window start address to result of expression
Alt-W	Switch between byte, word and long word display
Alt-F	Set window start to value of long word under cursor
+/-	Increment/decrement value under cursor

Watch window

Ins	Add a new watch expression
Del	Remove a watch expression

Disassembly window

up/down	Line up/down
left/right	Move a word at a time
Page up/down	Move cursor up/down a page at a time
Alt-G	Goto expression. Moves cursor to result of expression
Tab	Move cursor to PC
Shift-Tab	Move PC to cursor
Alt-C	Attach condition to breakpoint at cursor
Ctrl-C	Attach count to breakpoint at cursor
Alt-F5	Reset all breakpoint counts

F5	Toggle breakpoint at cursor
F6	Run code at target until instruction at cursor
F7	Traces. Steps over traps and line A/F
F8	Step over. Steps over subroutine calls and dbras
F9	Start target executing code from current PC

Shift-F5	Clear all breakpoints
Shift-F6	Reset all breakpoint counts
Shift-F7	Forced trace. Traces down traps and line A/F
Shift-F9	Run to specified address

+/-	Increment/decrement label level

File Window

up/down	Line up/down
left/right	Character left/right
Page up/down	
	Move cursor up/down a page at a time
Home/End	Beginning/End of file
Enter	Change file name
Alt-G	Goto line
Alt-S	Search
Alt-N	Next occurance of search string

Hercules Drivers

This switch, which should only be used with hercules mono cards, causes the debugger to use its built-in hi-resolution hercules drivers to give you many more rows of characters on the screen. The /g0 uses 80x50 mode and /g1 90x43.

Source Code Syntax

The statements in 68000 source code are either instructions which can be translated into 68000 machine code, or directives. Directives are used to control the operation of the assembler and how it interprets the source code.

SNASM68K supports the Motorola standard 68000 mnemonics. These mnemonics are not explained in this manual as it is not intended to be a 68000 tutorial.

Statement Format

Statements have the following general form :-

label	operation	operand	comment

Notes

. Labels always start in the first column unless they end with a colon (:).
. Fields are separated by tabs or spaces.
. Anything on a line after the operation and operand is taken as a comment but to avoid confusion it is recommended that they begin with a semi-colon.[1]
. Blank lines and lines containing only tabs and spaces are comments.
. Lines starting with a semi-colon or an asterisk are comments.
. Lines starting with an equals sign are case statement selectors.

Labels and Symbols

Labels and Symbols must obey the following rules

. A symbol contains characters from the set:

A-Z	uppercase letters
a-z	lowercase letters
0-9	digits
Underscore (_) and Dot (.)	special characters

. Digits may not be the first character of a symbol except in local labels.

. A symbol may optionally be followed by a colon when it is defined but not when it is referenced

. Local labels begin with an at (@) or optionally a dot(.) These are explained later in this manual.

As labels can contain dots it is possible for there sometimes to be confusion about whether a dot is part of a label or a size modifier. If this happens you

[1] The WS option makes the assembler insist on a semi-colon before comments. See the section on options for more information.

can either bracket the expression or use backslash (\) in place of the dot on the size modifier.

Examples of symbols

MySymbol	A	A135	Last_One
@Local	@2		

The assembler has various pre-defined constants available which usually start with an underscore. If you want to avoid clashes with any future ones of these then avoid the use of underscore as the first character of a label.

Integer Constants

The default base for constants is decimal unless the RADIX operator is used to change it. Hexadecimal numbers are preceded by a dollar sign and binary numbers by a percent sign. If you want to use the ASCII value of a character as a constant place it in quotes. If a character is preceded by caret (^) the corresponding control character is substituted.

Constant	Decimal Value
1234	1234
$A0	160
%101010	42
'A'	65
"?"	63
^M	13

RADIX

Syntax

```
                    RADIX      constant
```

Unless you tell it otherwise the assembler assumes that the radix of a number not preceded by $ or % is decimal. With the RADIX operator you can change this default to any value between 2 (binary) and 16 (hexadecimal). The argument to the RADIX operator is always evaluated using decimal as the radix.

If the AN (alternate numeric) option is enabled, you can also define constants in the Zilog/Intel fashion. The number, which must begin with a decimal digit, is followed by H,D,Q or B to specify the base as Hex, Decimal, Octal or Binary respectively. This feature does not work if the default radix is >10 as B and D are valid hex digits.

Location Counter

An asterisk (*) when used as a constant substitutes the value of the current location counter.

Example

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

Expressions

SNASM68K uses a signed 32-bit integer expression evaluator which has the following operators :-

Operators

Token	Syntax	Meaning
+	+X	positive(unary)
-	-X	negative(unary)
+	X+Y	addition
-	X-Y	subtraction
*	X*Y	multiplication
/	X/Y	quotient of division
%	X%Y	remainder of division
&	X&Y	logical and
!	X!Y	logical inclusive or
~	~X	logical compliment
^	X^Y	logical exclusive or
<<	X<<Y	shift X left (Y times)
>>	X>>Y	shift X right (Y times)
()	(X)	parenthesis

Comparison

=	X=Y	equals
<	X<Y	less than
>	X>Y	greater than
<=	X<=Y	less than or equals
>=	X>=Y	greater than or equals
<>	X<>Y	doesn't equal

Functions

def()	**DEF**(X)	symbol defined
ref()	**REF**(X)	symbol referenced
narg()	**NARG**(X)	parameters argument count
strcmp()	**STRCMP**(S1,S2)	string comparison
strlen()	**STRLEN**(S1)	string length
instr()	**INSTR**([X,]S1,S2)	substring location
type()	**TYPE**(X)	type of a symbol
sqrt()	**SQRT**(X)	square root
sect()	**SECT**(X)	base of section
offset()	**OFFSET**(X)	offset into section
filesize()	**FILESIZE**(S1)	Size of a file

Special constants

narg	**NARG**	parameters passed to macro
*	*	current value of PC
__rs	**__RS**	current value of rs counter
_year, _month, _day, _weekday, _hours, _minutes, _seconds		time and date constants

For a true result comparison operators return -1, for false they return 0.

Some of the above functions and constants may be unfamiliar but they are explained later in the manual.

Note

Unlike some assemblers SNASM68K uses a full 32-bit expression evaluator, so you must be careful when using large numbers. A number needs bit 31 set to be negative, not just bit 23.

Example

```
Pal0            equ     $FF8240     ; positive number!
; the above is NOT the same as
Pal0            equ     $FFFF8240   ; negative number!
```

Operator Precedence

SNASM68K evaluates expressions using the following rules :

- Operators with higher precedence are performed before ones with lower precedence
- Operators with the same precedence are performed left to right
- Expressions in parenthesis are always evaluated first as they have the highest priority

The table below shows operator precedence in descending order.

```
()
+, -, ~ (unary)
 <<, >>
&, !, ^
*, /, %
+, - (binary)
=, <, >, <=, >=, <>
```

It is usually best to parenthesize an expression to make it clear to both yourself and the assembler what you mean.

ALIAS and DISABLE

If any of the assembler's pre-defined constants or functions clash with symbols in your program or if you simply don't like the current names you can remove the definitions with the DISABLE command. As this would make the function inaccessable you can first rename it using ALIAS.

Example

```
_Type           alias       type
                disable     type
                ...
Type            dc.w        0       ; Type of account
```

ALIAS and DISABLE can be used on any name (symbol, function macro etc) that the assembler has already encountered but it cannot be used to remove or rename assembler instructions or directives.

Equates

Equates are a way of attaching a symbolic name to a constant or variable numeric value. This improves code readability and means only having to change one line to change the value of the constant.

Permanent Equates

EQU

This is the most common type of equate. The symbol on the left hand side of the line on which the EQU directive occurs is assigned the result of the expression on the right.

Examples

```
True            equu    -1
```

Once a symbol is given a value with the EQU directive any attempt to assign a new value will generate an error .

It is perfectly legal to re-equate a symbol to the same value as it already holds,. These are called benign redefinitions and will usually occur when an include file defines hardware locations for itself that the main program also uses.

Most assembler insist that the expression to which the symbol is equated contains no forward references. With SNASM68K however, as much as possible of the expression is evaluated, and the remainder of the work is done at the end of the first pass. You should be careful when forward referencing local symbols in this way, as they will probably not exist when the evaluation is done.

Redefinable Equates

SET and =

Whereas EQU is used for defining constants SET and = are used for defining variables. A symbol defined with either of these directives can have its value changed as often as you like. These variables are frequently used as loop counters and scratch variables in macros.

Example

```
FreeSpace       set     0                   ; zero free space total
                ...
FreeSpace       set     FreeSpace+1024      ; 1K more free here
```

Set and = are synonymous.

e.g.

```
Total           =           0
                ...
Total           =           Total+1
```

SNASM68K generates a warning if a variable is referenced before it is defined. If you do forward reference a variable, the value substituted will be the value of the variable at the end of the first pass.

Unlike EQU the argument of SET must not contain forward references. Also no information about the type of the expression is inherited by the symbol, which means they are not always the best choice in code using sections.

eg.

```
                section     Code,Text
Marker          =           *
                ...
; This next line will give an error as * is relative to the start of the section and
; Marker is just a number.
; If Marker is defined with an EQU the expression will evaluate.
                dcb.b       *-Marker,63
```

String Equates

EQUS

EQUS is used to assign a value to a string variable. String variables are used for things like version and copyright messages or as scratch variables in macros.

As symbols equated with the EQUS directive can be used anywhere in your code they must be preceded by a backslash (\) character so the assembler knows that a string equate is following. If there could be confusion as to where the name of the string variable ends use a second backslash as a delimiter. The only exception to this rule is in expressions, as if the assembler finds a symbol which is a string variable whilst evaluating an expression, then the string is substituted automatically.

The parameter to the EQUS directive will usually be delimited by quotes in which case that text is assigned to the string equate. If the quotes are ommitted the assembler expects the parameter to be the name of an already defined string equate, the contents of which are transfered to the new string equate.

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

Examples

```
1)
Version         equs        'Demo Version 0.21 22/05/89'
                ...
                dc.b        '\Version'
; expands to
;               dc.b        'Demo Version 0.21 22/05/89'


2)
IDA             equs        'IntDispAddr'
                ...
                dc.l        \IDA\1
; expands to
;               dc.l        IntDispAddr1

                dc.l        \IDA\2
; expands to
;               dc.l        IntDispAddr2
; Note use of second backslash as delimiter


3)
HexStr          equs        'FF00'
                ...
                hex         \HexStr\\HexStr     ; Note two backslashes
; expands to
;               hex         FF00FF00
```

Strings are delimited with quotes, single (') or double ("). The quotes which are used to delimit the string after the EQUS directive do not become part of the symbols contents. If you want to put a single quote in the string you can either double up the single quote or delimit the string with double quotes . Exactly the same applies to double quotes: either delimit with single quotes or use two doubles.

Examples

```
Single1         equs        'It''s great'
Single2         equs        "It's great"
Double1         equs        'They shouted "Yes" together'
Double2         equs        "They shouted ""Yes"" together"
```

There is a predefined string equate called _filename which holds the name of the file being assembled. This is the file on which SNASM was invoked not the current include file.

Example

```
                        dc.b          'To rebuild this assemble \_filename',0
```

RS Equates

RS, RSSET, RSRESET

Syntax

```
                        RS.size     Count
                        RSSET       Value
                        RSRESET     [Value]
```

RS equates are used when you want to define a set of symbols as offsets into some data structure but do not want to keep track of the offsets yourself.

The assembler has an internal variable called __RS which is used to keep track of the current offset. When you define a symbol using the RS directive __RS's value is assigned to the symbol and the counter is advanced by the specified number of bytes, words or long words. The RSRESET directive zeros __RS and is used at the start of each new structure. RSSET puts any value you like into the __RS variable, it is used when you want to start the offsets at something other than zero.

For compatibility with other assemblers, the RSRESET directive can take an optional parameter, which if present causes it to behave exactly like the RSSET directive.

Examples

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.
; or we could have used
FileSpecSize            equ           __RS
```

2)

```
                        rsset         -8      ; -ve as address register points
                                              ; eight bytes into data structure
; We could have used :-
;                       rsreset       -8
; but this is for compatibility and its use is discouraged
ObjXPos                 rs.l          1
ObjYpos                 rs.l          1
ObjFlags                rs.w          1
ObjSpeed                rs.w          1
```

If the auto-even option is enabled the word and long forms of RS force the __RS variable to be set to the next even boundary before the operation is performed .

Use of RS and associated directives results in data structure definitions that, in addition to being easy to read, allow you to add or subtract fields without having to count on your fingers!

Pre-defined Constants

To help you keep track of which versions of your programs were made when, SNASM68K provides the following pre-defined numeric constants.

Name	Contents
_YEAR	Gregorian (e.g. 1989)
_MONTH	1=January ... 12=December
_DAY	1=lst day of month
_WEEKDAY	0=Sunday .. 6=Saturday
_HOURS	Time in 24 Hour format
_MINUTES	
_SECONDS	

Example

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

Notes

The above constants hold the date and time at the start of assembly: their values do not change during assembly.

To put the date and time into a string you can use the \# parameter which is described later in the chapter on Macros.

Register equates

EQUR

EQUR is used to define a symbol as a synonym for a data or address register to improve code readability.

Example

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

Dots are not permitted in the names of register equates so that the assembler knows that CurIndex.w means the lower word of (d2) rather than a register equate called CurIndex.w .

REG

Similarly REG is used to define a symbol as a synonym for a list of data or address registers.

Example

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

Defining Data

Defining Initialised Data

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 word format.

Examples

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

If you want to put a single quote (') in a string you can either double up the single quote or delimit the string with double quotes ("). Exactly the same applies to double quotes: either delimit with single quotes or use two doubles.

Examples

```
Single1         dc.b        'It''s great'
Single2         dc.b        "It's great"
Double1         dc.b        'They shouted "Yes" together'
Double2         dc.b        "They shouted ""Yes"" together"
```

DCB

Syntax

```
                DCB[.size]  count,value
```

The DCB directive is used to generate a block of memory containing a specified number of instances of the same byte, word or long value.

Examples

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

If the auto-even option is enabled the word and long forms of DC and DCB force the program counter to the next even boundary before the operation is performed .

Whereas some assemblers truncate out of range parameters to force them into range, SNASM68K gives an error

e.g. All of the following cause an error:-

```
                        dc.w        70000       ; greater than 65535
                        dc.w        -40000      ; less than -32768
                        dc.b        260         ; greater the 256
                        dc.b        -130        ; less than -128
```

Handy tip

If you define Word and Byte as shown below you can use them as a quick and readable way of truncating a number to a word or byte.

```
Word            equs        '$FFFF&'
Byte            equs        '$FF&'

BigNumber       equ         70000
                dc.w        \Word\-40000
                dc.w        \Word\BigNumber
                dc.b        \Byte\(-80*9)       ; Brackets are important!
                dc.b        \Byte\BigNumber
```

HEX

The HEX directive is followed by a stream of hex nibbles which are paired-up to give bytes.

```
e.g.
MaskTab1        dc.b        $01,$02,$04,$08,$10,$20,$40,$80
; could be written as ...
MaskTab1        hex         0102040810204080
```

The HEX directive should only be used for small amounts of data. Data stored as hex is very unreadable, takes twice the space of binary data and is slower to load and assemble. Rather than having vast quantities of hex data you may prefer to put the data into a file as raw bytes and use the INCBIN directive described later.

DATA and DATASIZE

The DC directive can only be used for constants that can be contained within 32 bits. It is sometimes necessary to place larger constants within code and the DATA and DATASIZE directives have been provided for this purpose.

DATASIZE is followed by a single parameter which specifies how many bytes are to be used for constants defined using the DATA directive. The DATA directive has a variable number of parameters, which are decimal by default but which can be hex if preceeded by a dollar ($). Binary numbers and the alternate numeric form cannot be used.

Example

```
                    datasize    8     ; 256 bit numbers
                    data        10000,1000000
                    data        $100,-200
                    data        200000000
```

Reserving Space

DS

The DS directive is used to reserve, and initialise to zero, a block of memory.

e.g.

```
ScratchBuffer       ds.b        1000        ; space for 1000 bytes
PointerList         ds.w        16          ; space for 16 words
```

If the auto-even option is enabled the word and long word forms of DS force the program counter to the next even boundary before the operation is performed.

The DS directive is used in BSS type sections to reserve space. Like everything in BSS sections no initialisation is performed, so don't rely on there being zeros in memory!

Changing the Program Counter

ORG

ORG is used to tell the assembler where in the target machine the code is to reside. You should not use the ORG directive if you are producing linkable output as the linker decides where the various parts of the program are to go. If you are not using sections in your program you are free to use as many ORG directives as you wish.

Example

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

If the argument to ORG starts with a question mark (?) it is interpreted as the amount of RAM the program needs and the target is asked to reserve that much memory. The target then returns the address at which it managed to reserve the RAM and SNASM68K assembles the program to run at that address.

Example

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

This facility is useful when developing for a machine with the operating system resident. This form of the ORG directive can take an optional second parameter which indicates the type of memory to be reserved. The value of this parameter is specific to the version of the target software being used.

EVEN

The EVEN directive forces the program counter to the next even address.

Example

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

CNOP

CNOP sets the program counter to a given offset from any size boundary.

Examples

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

When using sections it is not possible to align the program counter to a larger boundary than the alignment of the current section, i.e. EVEN cannot be used in a byte aligned section and CNOP to a 4 bytes boundary cannot be used in a word aligned section.

OBJ and OBJEND

All code generated after the OBJ directive but before the OBJEND directive will still be placed at the same place in memory but the code will have all offsets set for it to run at the address specified after the OBJ directive.

This is sometimes refered to as assembly with offset.

Example

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

RelocCode
                obj         RunAddr
; Everything within this section will be set to run at RunAddr
                jmp         Startup
                ...
Startup         move.w      #$2700,sr
                ...
                objend
RelocEnd
```

OBJ and OBJEND should always be paired correctly if you need to revert to 'normal' assembly. Don't ommit an OBJEND directive or use two of them, or attempt to nest OBJ/OBJEND pairs, as the assembler will loose track of where the PC actually is, and no doubt you will too!

Including Source and Binary

INCLUDE

You will almost certainly want to break your source code into several smaller files either to make it more manageable or so you can use some of the parts in more than one program.

The INCLUDE directive tells the assembler to process another file before continuing with the current one. You will normally have one 'root' file which includes all the other parts of your code. If you want, these included files can include files of their own.

Example

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

If the text following the backslash can be confused with a string equate use either two backslashes or a forward slash. You may enclose the file spec. in quotes if you wish but they are optional.

```
e.g.
                include     c:\\general\\maths.asm
; or
                include     c:/general/maths.asm
```

If the file cannot be found in the current directory it is searched for in all directories specified with the /j switch.

INCBIN

If you have a lot of data in raw binary format such as graphics or music data, you can use the INCBIN directive to include this into your program. As the data is just bytes the assembler knows nothing about its internal structure and you will have to put a label on the data and handle offsets into it yourself.

Example

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

See the above notes on using backslashes in path names.

If the file cannot be found in the current directory it is searched for in all directories specified with the /j switch.

If you need to know the size of a binary file before you include it then you can use the FILESIZE function. This returns the size of a file in bytes or minus one if the file cannot be found.

Example

```
BinHeader        dc.l        filesize('sintab.bin')
                 incbin      sintab.bin
```

DEF and REF

If you have a general purpose piece of source code that will be included in quite a few projects you often want to have control over exactly which routines are included. The REF operator returns true if the symbol following it has already been referenced.

Example

```
                 if          ref(Printf)
; This is only assembled if Printf has been referenced
Printf           movem.l     d0/a0,-(sp)
                 ...
                 rts
; end of printf
                 endif
```

The DEF operator returns true if the symbol following it has already been defined. You can use the DEF function to check if the variable has already been defined and then define it only if necessary.

Example

```
                 if          ~def(StringBuffer)
; This is only assembled if StringBuffer hasn't been defined
StringBuffer     ds.b        64
                 endif
```

TYPE

The type function provides information about a symbol. It enables a macro to determine exactly what it has been passed as a parameter.

The value returned is a word, with the bits having the following meanings :-

0 - Set if symbol has absolute value
1 - Set if symbol is relative to start of a section
2 - Set if symbol was defined using 'SET' directive
3 - Set if symbol is a macro
4 - Set if symbol is a string equate
5 - Set if symbol was defined using an 'EQU' directive
6 - Set if symbol was specified in an 'XREF' statement
7 - Set if symbol was specified in an 'XDEF' statement
8 - Set if symbol is a function (STRCMP, NARG etc.)
9 - Set if symbol is a group name

10 - Set if symbol is a macro parameter
11 - Set if symbol is a short macro (MACROS)
12 - Set if symbol is a section name
13 - Set if symbol is absolute word addressable
14 - Set if symbol is a register equate
15 - Set if symbol is a register list equate

To check specific bits returned by the TYPE function use the bitwise and operator (&).

Example

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

Setting Target Parameters

REGS

If you are either sending code straight to the target machine or if you are producing a CPE file, it is possible to specify the values that you want the 68000 registers to have when your code is executed. Usually this feature is used to set the PC at which you want execution to begin, and the value of SR at this time.

Example

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

As the 68000 has two stack pointers you must use be specific about which you mean by using USP and SSP.

This feature cannot be used if you are producing machine specific relocatable or pure binary formats.

WORKSPACE

The target sofware uses about 1K of memory for its own workspace. With most versions of the target software it is possible to change the address of this using the WORKSPACE command.

Example

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

Assembly Flow Control

END

END tells the assembler to immediately stop processing lines. The use of END is entirely optional as SNASM68K 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. Use of this feature is discouraged as the REGS directive can be used to achieve that same thing.

Examples

1)

```
StartUpCode     lea         MyStack,sp
                ...
                rts
; End of program
                end         ; No start address
```

No error here as this line is never reached.

2)

```
StartUpCode     lea         MyStack,sp
                ...
                jmp         MainLoop
                end         StartUpCode        ; Start at StartUpCode
```

IF ELSE ELSEIF and ENDIF

These are used to control exactly which lines of code get assembled. Conditional assembly is useful if you need to generate several versions of the program, but it is mainly used in macros to cause them to expand differently under different conditions.

Examples

1)

```
False           equ         0
True            equ         -1
DebugMode       equ         False
                ...
; The following code will be skipped if DebugMode is false
                if          DebugMode
                 move.w     XPosition,d0
                 jsr        Printf
                endif
```

```
2)
English         equu    0
French          equ     1
German          equ     2
Language        equ     English
                ...
; Assemble correct drink ordering string according to current language.
                if          Language=English
                 dc.b       'Two beers please',0
                else
                 if         Language=French
                  dc.b      'Deux beir sil vous plait',0
                 else
                  if        Language=German
                   dc.b     'Zwei bier bitte',0
                  endif
                 endif
                endif
```

```
3)
English         equ     0
French          equ     1
German          equ     2
Language        equ     English
                ...
; Assemble correct drink ordering string according to current language.
                if          Language=English
                 dc.b       'Two beers please',0
                elseif      Language=French
                 dc.b       'Deux beir sil vous plait',0
                elseif      Language=German
                 dc.b       'Zwei bier bitte',0
                endc
```

Notes

In example three the ELSEIF directive has been used to make the code more readable. In the same example ENDC is used instead of ENDIF to show that they are synonymous.

For compatibility with other assemblers the ELSEIF directive can be used without any parameters in which case it acts exactly like an ELSE directive.

Indenting of code has been used in these examples to make the code more readable. You may wish to do the same thing but it is entirely optional.

The logical not (~) operator can be used to branch on the opposite of a condition but you must be careful to parenthesize the expression correctly.

e.g.

```
; It may be tempting to write
                        if          ~Language=German
; but you probably wanted
                        if          ~(Language=German)
```

CASE and ENDCASE

If you have a symbol which you want to use to select between several pieces of code as in example three above, the CASE directive can be used. This directive can be used in place of the IF directive as shown in example one below, but it really comes into its own for multi-way choices.

Syntax

```
                        CASE        Expression
=Expression{,Expression}
                        ...
=?
                        ...
                        ENDCASE
```

Examples

1)

```
False                   equ         0
True                    equ         -1
LargeBuffer             equ         True
                        ...
                        case        LargeBuffer
=False                  ds.b        256
=True                   ds.b        1024
                        endcase
```

2)

```
; Assemble correct drink ordering string according to current language.
English                 equ         0
French                  equ         1
German                  equ         2
Language                equ         English
                        ...
                        case        Language
=English                dc.b        'Two beers please',0
=French                 dc.b        'Deaux beir sil vous plait',0
=German                 dc.b        'Zwei bier bitte',0
                        endcase
```

3)

```
ExecuteMode       equ         0
DebugMode         equ         1
SaveMode          equ         2
ProgMode          equ         SaveMode
                  ...
                  case        ProgMode
=ExecuteMode,DebugMode                   ; Select on either value
                  lea         MyStack,sp
                  jmp         CodeStart

=SaveMode         lea         CodeStart,a0
                  moveq       #0,d0
                  move.w      #CodeSects,d1
                  jsr         WriteSectors
                  rts
=?                inform      3,'Bad value for ProgMode'
; See later in this manual for explanation of INFORM
                  endcase
```

Notes

The =? case is used if none of the other cases succeed. If there is no =? case and all the other cases fail then none of the code will be assembled.

REPT and ENDR

The REPT directive is used to repeat a short section of code a predetermined number of times.

Syntax

```
                  REPT        Expression
                  ...
                  ENDR
```

Examples

1)

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

2)

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

WHILE and ENDW

The WHILE directive is used to repeat a short section of code whilst an expression evaluates true.

Syntax

```
                WHILE       Expression
                ...
                ENDW
```

Example

```
Factor          equ         4

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

DO and UNTIL

The DO/UNTIL loop is similar to the WHILE/ENDWHILE loop except the condition is checked at the end of the loop and the looping finishes once the condition becomes true.

Syntax

```
                DO
                ...
                UNTIL       Expression
```

Example

```
Factor          equ         4

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

Macros

A macro is a way of giving a sequence of assembler lines a symbolic name so they can be assembled later as many times as you require. A macro is invoked as if it where a new directive and like directives parameters can be passed to them. They are used to extend the features of the assembler or just save a bit of typing.

Introducing Macros

MACRO and ENDM

The lines between the MACRO and the ENDM directives are stored in memory and can be referenced using the label preceding the MACRO directive.

Example

```
; Expands to two NOP's
TwoNops         macro
                nop
                nop
                endm
                ...
Delay           TwoNops     ; assemble two NOP's
                TwoNops     ; two more
```

Whenever TwoNops is used as if it were a directive the macro is expanded.

MEXIT

The MEXIT directive causes expansion of the current macro to stop immediately.

Example

```
; Macro which repeat a string a certain number of times.
NewDs           macro
                if          narg<>2
                inform      2,'Wrong number of parameters'
                mexit       ; exit macro now
                endif

                rept        \2
                dc.b        \1
                endr

                endm
                ...
                NewDs       'Hello',2
```

The MEXIT directive is supported for compatibility reasons as the above could be written much more neatly using the IF..ELSE..ENDIF construct.

Macro Parameters

A macro can take parameters which can be used anywhere in the macro just as if they were string equates, i.e. preceded by a backslash (\). Again if there could be any confusion the parameter can be terminated with another backslash.

There can be up to thirty two parameters, \0 to \31 with \0 being the size of the macro i.e. the text following the dot (.) ,if any, when the macro was invoked.

Example

1)

```
; Macro to increment register
Inc             macro
                addq.\0     #1,\1
                endm
                ...
                Inc         d0      ; expands to addq.  #1,d0
                                    ; the assembler ignores the dot
                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
```

2)

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

3)

```
; If you enclose an argument in angle brackets (<>) you can use spaces and
; commas in it.
Format          macro
                dc.b        13,13,'  \1...',0
                endm
                ...
                Format      <Stop, Press a key>
```

SHIFT and NARG

It is often useful to have a macro which takes a variable number of parameters. The predefined NARG symbol and the SHIFT directive are used to determine how many parameters there are and step through them. Shift causes \1 to be lost and shifts the rest of the parameters down so the \1 becomes what \2 was etc.

Example

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

See the description of extended macro parameters for more uses of SHIFT and NARG.

Named Macro Parameters

If you prefer you can assign symbolic names to the macro parameters to be used instead of \1 to \31.

Example

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

Short Macros

MACROS

Normally control structures must be properly nested within macros. If you start a structure in a macro you must finish it before the ENDM and equally you cannot terminate a structure that you didn't start in that macro.

You may occasionally when porting code from other assemblers, need to define macros to imitate control structures. Short macros contain only a single line of code, but this line can be a control structure directive if you wish.

Examples

1)

```
; Macro to implement the IFEQ (if equals) conditional
ifeq            macros
                if          \1=0
; Note: Short macros don't have an ENDM
                ifeq        DebugMode
                ...
                endif
```

2)

```
; Macro to implement the IFND (if not defined) conditional
ifnd            macros
                if          ~def(\1)

                ifnd        Count
Count           dc.w        0
                endif
```

Note

If the /k command line option is used the above macros are automatically defined along with several others.

Extended Parameters

SNASM68K allows you to pass a list of items enclosed in curly brackets ({}) to a macro parameter.

The NARG symbol has been extended so that it can report how many items have been assigned to a parameter and similarly SHIFT can now be used the shift those items.

Example

```
Black           equu    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

PolygonList     macro

polygons\@      =       narg/2
; Check narg was even
                if      polygons\@*2<>narg
                inform  2,'Bad parameter list'
                else
; Handle all polygons
                rept    polygons\@

                dc.b    \1
points\@        =       narg(2)
                dc.b    points\@
                rept    points\@
                dc.b    \2
                shift   2
                endr

                shift
                shift

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

The above example may look complex but note how all the complexity is hidden away inside the macro and how neat the main code looks.

For compatibility with other assemblers it is possible to equate a symbol to a list of parameters using EQUS and use NARG and SHIFT on the string equate exactly as they were used on the macro parameter above. This feature allows the macro to accept lists of arguments using angle brackets instead of curly brackets.

Example

```
DefItems          macro
Day               equs       {\1}
                  rept       narg(Day)
                  dc.b       \Day,0
                  shift      Day
                  endr
                  ...
                  DefItems   <'Mon','Tue','Wed','Thu','Fri','Sat','Sun'>
; normally we would have used :-
                  DefItems   {'Mon','Tue','Wed','Thu','Fri','Sat','Sun'}
```

Continuation Lines

If when you are invoking a macro, the line becomes very long, you can terminate the line with a backslash (\), and continue the list of parameters on the next line.

Example

```
Double7           macro
                  dc.w       \1*2,\2*2,\3*2,\4*2,\5*2,\6*2,\7*2
                  endm
                  ...
                  Double7    1111,2222,3333,4444,5555,\
                             6666,7777
```

Label Importing

A macro can import the label on the line on which it was invoked and use it just like any other parameter. The label is not defined to be at the current PC as usually happens, in fact, unless the macro specifically assigns a value to the symbol in some way it is undefined.

To tell SNASM68K that you would like to use label importing you must specify use asterisk (*) as the first named macro parameter, then you can use * to substitute for the label.

Example

```
; A Macro that assigns labels relative to the start of a data table.
RC                macro       *,Data
\*                equ         *-VarBase
                  rept        narg(Data)
                  dc.\0       \Data
                  shift       Data
                  endr
                  endm
                  ...
VarBase           equ         *
L1                rc.w        {1,2,3,4}
L2                rc.w        {5,6,7,8}
                  ...
                  lea         VarBase(pc),a6
                  move.w      L1(a6),d0
```

Advanced Macro Features

PUSHP and POPP

SNASM68K lets you push some text and later pop it into any string variable.

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
```

You are not restricted to poping the parameter in the same macro that pushed it. This allows some very flexible macros to be written to handle self-referencing data structures.

Turning numbers into strings

The \# and \$ parameters substitute the decimal or hex value of a symbol into your code at any point. They are used to turn numbers into strings for formatting data or building arrays of symbols.

Examples

```
1)
; Put the data and time into a string
AsmDate             dc.b  '\#_day/\#_month/\#_year'
; expands to
AsmDate             dc.b  '13/12/1989'

2)
Col0                equu        $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.
```

Unique Labels

There is a special macro parameter \@ which expands to an underscore followed by a decimal number which increments upon each macro invocation to guarantee uniqueness.

Example

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

Each time the macro is expanded a different label such as Loop_000 or Loop_300 will be generated.

PURGE

Macros can take up a large amount of memory and even though SNASM68K stores macros very efficiently they might as well be removed if they are no longer needed. The PURGE directive removes a macro from the symbol table and frees up the memory it was using.

Example

```
BigMacro        macro
                ....
                endm

; Code that uses BigMacro
                BigMacro    1,2,3

                purge       BigMacro
; BigMacro now no longer exists and can be redefined if we want.
```

If a macro purges itself, the definition of the macro is not removed until the macro exits.

If you just want to redefine a macro there is no need to purge it: when a new definition of the macro is encountered the old macro is automatically purged. A macro can, if you really want, redefine itself! The new definition will be effective next time the macro is invoked but does not interfere with the expansion of the current invocation.

Example

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

String Handling

SNASM68K provides some useful string handling functions and directives which are used (usually in macros) for comparing, searching and slicing strings.

STRLEN

STRLEN is a function that can be used anywhere in an expression and which returns the length in characters of its string parameter.

Example

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

STRCMP

Syntax

symbol = **strcmp(**string1,string1**)**

STRCMP is a function that returns a boolean value (0 for false, -1 for true) which is the result of comparing its two string parameters.

Example

```
Language        equs        'English'
                ...
; Assemble correct drink ordering string according to current language.
                if          strcmp('\Language','English')
                 dc.b       'Two beers please',0
                else
                 if         strcmp('\Language','French')
                  dc.b      'Deux beir sil vous plait',0
                 else
                  if        strcmp('\Language','German')
                   dc.b     'Zwei bier bitte',0
                  endif
                 endif
     endif
```

INSTR

Syntax

symbol = **instr(**[start,]string,sub-string**)**

INSTR is used to search a string to see if it contains a sub-string. If the sub-string cannot be found the result is zero otherwise it is the offset into the string that the sub-string occurred. Here as in all of the string commands the first character in the string is character number 1.

Example

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

INSTR's optional first parameter is the position from which to start the search.

SUBSTR

Syntax

```
symbol              substr      [start],[end],string
```

SUBSTR is similar to EQUS in that it is a way of doing a string equate however it allows the start and end characters of the string to be specified.

Example

```
TestStr             equs        'What does this do?'
Temp1               substr      6,9,'\TestStr'
; Temp1 will equal 'does' (without the quotes of course)
Temp2               substr      ,4,'\TestStr'
; Temp2 will equal 'What'
Temp3               substr      6,,'\TestStr'
; Temp3 will equal 'does this do?'
```

Local Labels and Modules

When you are writing a large program it becomes very hard to think of an informative and unique name for each label . Local labels help solve this problem as they only exist within an area called their 'scope' and they can be re-used outside this area. They are mainly used within routines as 'place markers' for skips and loops where there is no need for other routines to be able to access them.

Local labels have an at sign (@) as their first character and this is followed by an valid label characters.

Example

```
; These are all local labels
@Label
@L1
@123
```

SNASM68K has two ways of controlling the scope of local labels. In the first of these a local label's scope extends between two non-local labels.

Example

```
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
```

The local label @NoInc can be re-used as it's scope is between IncNzD0 and IncNzD1.

This form of scoping is supported for compatibility reasons and it is not recommended.

MODULE and MODEND

SNASM68K has another scoping system which can be used freely with the above scheme and which gives the programmer much more control over the scope of a label.

Code which is between MODULE and MODEND directives is said to comprise a module. A local label defined in a module cannot be referenced outside that module, and can be re-used freely. Equally any local labels defined outside the module cannot be referenced within it.

Example

```
ClearData       module

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

                rts

@ClearIt        module

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

                modend
; end of @ClearIt

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

A label is assumed to be inside a module if it is declared on any line after the MODULE directive up to and including the line on which the MODEND occurs.

Outside modules the 'between non-locals' scoping is in force but it is worth noting that a module is treated like a non-local and default scoping is blocked by it.

e.g.

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

                movem.w     (sp)+,d7
                dbra        d7,@Loop
```

The assembler will give an error on the last line of the above example as the DBRA is not within @Loop's scope.

Macro expansion has no effect on the scoping of local labels. Any local labels which exists in a module can be referenced within a macro expanded in that module. If you want a macro to have its own local labels you can either use a module within the macro, use the \@ parameter or use the LOCAL directive described below

The use of modules is strongly recommended as they serve to neatly 'encapsulate' routines and rigidly define where labels local to routines can and can't be referenced.

LOCAL

The LOCAL directive is another way of declring labels within macros. Any symbols declared with the local directive can be used as if they were normal symbols but their scope is limted to the current macro.

Example

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

Local does not type the symbols it defines. You can go ahead and declare them as labels, text equates or whatever you like.

e.g.

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

Options

SNASM68K has several options which can be controlled either within the program or from the command line. See the section on command line syntax for information on setting assembler options on the command line.

OPT

The OPT directive is used to set the state of the assemblers options from within the source code.

Assembler options:

Abbreviation	Default	Description
ae	On	Automatic even
an	Off	Alternate numeric format
c	Off	Case sensitivity
d	Off	Descope locals on equ and set
e	On	Print source line which caused error
l	Off	Use dot (.) instead of at (@) for locals
s	Off	Treat equated symbols like labels
w	On	Print warnings
ws	Off	Allow white space in operands
v	Off	Write local labels to symbol file
Optimisations:		
op	Off	PC relative optimisation
os	Off	Short branch optimisation
ow	Off	Absolute word optimisation
oz	Off	Zero displacement optimisation
oaq	Off	Add quick optimisation
osq	Off	Subtract quick optimisation
omq	Off	Move quick optimisation

Example

```
; Enable all optimisations
                        opt         oz+,os+,ow+
; Turn off auto-even and warnings
                        opt         ae-,w-
```

Alternatively the /O switch can be used to set these options on the command line, see the earlier section on command line syntax.

AE - Automatic Even

When this option is enabled word length data directivess force the program counter to the next word boundary before they are executed. This applies to the word and long word forms of DC,DCB,DS and RS.

AN - Alternate Numeric

Enables use of Zilog/Intel form of constants. See the section on RADIX for a full description of this facility.

C - Case Sensitivity

When this is enabled the case of the letters in a label's name becomes important. For instance you could have one label called BigLabel and another called biglabel.

D - Descope Locals

Outside modules the EQU and SET directives do not usually affect the scope of local labels. Set this option if you want EQU and SET to descope locals.

E - Error Line Printing

When the assembler detects an error it is reported along with the file name and line number where it occured. If this option is enabled the text of the line that caused the error will also be printed.

L - Local Label Character

SNASM uses the at sign (@) as its default local label character. This option changes this to dot (.)

S - Treat equates like labels

When this option is enabled the assembler treats equated symbols like labels and writes them to the symbol table as such.

W - Give warnings

There are several things that even though they are not errors are unusual enough for it to be worth the assembler reporting them. If you don't want warnings reporting then disable this option.

WS - Allow white space

Normally in 68000 source code comments follow any white space in the operand field. This prevents the use of tab and space to increase code readability to the WS option has been added, which causes the assembler to ignore white space in operands and to insist on the use of semi-colon to start a comment.

V - Write locals to symbol table

If you want to see local labels in the debugger then enable this option. To keep the size of the symbol table down you may prefer to only turn on this option for the part of the code you are currently debugging.

OP - PC Relative Optimisation

Changes absoulte long addressing to PC relative addressing wherever possible and legal.

OS - Short Branch Optimisation

Backwards relative branches that could use the short form even though you have not specified it are used automatically if you enable this option.

OW - Absolute Word Optimisation

If you use the absolute long addressing mode and the address will fit into a word the assembler will use the shorter mode if this option is enabled.

OZ - Zero Displacement Optimisation

If you use the address register indirect with displacement addressing mode and the displacement is zero, the assembler will use the address register indirect mode if this option is enabled.

OAQ. OSQ and OMQ Quick Instruction Optimisation

This option causes all add, sub and move instructions that could be coded as the quick forms to be coded as such. Only move.l is changed to moveq not move.w!

All of the above optimisations can only be performed on backward references.

PUSHO and POPO

If you want to change the setting of an option briefly and then return it to its previous value you can use PUSHO to save the current state of all the options and later use POPO to return them to their previous state.

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
```

Errors and Warnings

Errors and warnings are generated by the assembler whenever it detects anything wrong. SNASM provides a means for the programmer to raise errors if an error condition occurs which the assembler can't detect, i.e.. a data table becoming too large.

INFORM

INFORM can generate errors of four different severities and display messages with embedded parameters to describe the error in detail.

Syntax

INFORM severity,string[,operands]

Severity is a number from 0 to 3 where 0 causes the message to be printed but no action taken, 1 gives a warning, 2 gives an error and 3 causes a fatal error where assembly stops immediately.

If the string contains %d, %h and %s these are substituted with the decimal, hex or string values of the operands in the order that they occur.

Example

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

FAIL

Fail is supported for compatibility and is the equivalent of :-

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

Listings

If a listing file name is given on the command line or if one is set by default in the SNASM68K environment variable then the assembler will generate a listing of the program during the first pass.

LIST and NOLIST

As listings as usually used to check how macros are expanded it is very rare that you will want the whole of your program listed. Listing is always on at the start of the program (if a listing file is specified). The NOLIST directive is used to turn off listing generation whilst the LIST directive turns it back on again.

Alternatively you can use the LIST directive with either plus (+) or minus (-) as a parameter to turn listing mode on and off. The assembler has an internal listing state which starts at 0 and listing output is only generated when this variable is positive. LIST without a parameter sets this variable to 0, NOLIST sets it to -1, LIST plus increments it and LIST minus decrements it.

Example

```
                NOLIST              ; state=-1, no listing
; Not listed
                LIST                ; state=0, listing
; Listed
                LIST        -       ; state=-1, no listing
; Not listed
                LIST        -       ; state=-2, no listing
; Not listed
                LIST        +       ; state=-1, no listing
; Not listed
                LIST        +       ; state=0, listing
; Listed
```

Normally the assembler turns off listing generation whenever it is expanding a macro. If you need to see how your macros are expanding use the /M switch on the command line. Similarly code which is being ignored due to conditional assembly does not go into the listing unless the /c switch is used.

Sections and Groups

Introduction to Sections and Groups

Sections are used when you want parts of your program that are in different places in the source code to be placed next to each other in memory. If you want all of your variables at the bottom of memory followed by strings followed by executable code then you should be using sections.

Sections are also used when you want to produce output in a machine specific relocatable format.

In addition to sections SNASM supports the concept of groups which are a way of making sure certain sections are together in memory and of assigning them various attributes.

Syntax

Declaring a group :-

```
GroupName        GROUP     [Group Attributes]
```

If a group has multiple attributes they are all listed after the group directive separated by commas.

Attribute	Function
ORG	Specify address in memory at which to place group
OBJ	Allows a whole group to use assembly with offset
SIZE	Specifies maximum size for group
BSS	BSS groups do not contain any initialised data
WORD	Tells the assembler/linker that a group can be absolute word addressed
FILE	Used to write contents of a group to a binary file
OVER	Causes two or more groups to start at the same address in memory

Opening a section :-

```
                 SECTION   SectionName[,GroupName]
```

If the group name is omitted then the section will be placed in a default unnamed group unless the section has been previously assigned to a group, in which case that group will be used. It is possible to write a program that doesn't use groups at all but BSS type sections will not be available.

The SECTION directive can be optionally followed by a size specifying its alignment. The default alignment is word which means that all parts of the section will start on a word boundary. If a section is byte aligned the EVEN directive cannot be used within it. The CNOP directive cannot be used to align the PC to a larger value than the alignment of the current section.

Example using sections and groups

Aim

So that variables can be absolute word addressable they must be at the start of the program. The variable are to be followed by two sections containing strings and one containing executable code. Uninitialised data is to reside in a section of its own above all the other sections.

Code

```
LowGroup        group       word
StringGroup     group
CodeGroup       group
BssGroup        group       bss

                org         $1000
; When using sections only one ORG is allowed and it must be before
; section definitions

                section     Data,LowGroup
Var1            dc.w        2
Var2            dc.w        89

                section     Tables,BssGroup
Scratch         ds.w        256
Temp            ds.l        100

                section.b   String1,StringGroup
CRLF            dc.b        13,10,0
Prompt          dc.b        '>',0

                section     Tables,BssGroup
FaceList        ds.b        64

                section.b   String2,StringGroup
Hello           dc.b        'Welcome to SNASM',0

                section     Code,CodeGroup
                lea         MyStack,a7
                nop
```

Groups are placed in memory in the order in which they are declared with sections placed within groups in the order in which they are opened.

PUSHS and POPS

If you need to temporarily open a new section (in a macro for instance), you can use PUSHS to save the current section, open a new section, and then use POPS to revert to the old section.

Example

```
; macro to place the current PC into a separate section
MarkPlace          macro
                   local        Temp
Temp               equ          *
                   pushs
                   section      MarkSection
                   dc.l         Temp
                   pops
                   endm
```

SECT and OFFSET

The SECT function returns the base of the section in which its parameter is defined. This cannot be evaluated until the end of the second pass if you aren't linking, and if you are linking it cannot be evaluated until link time.

The OFFSET function returns the offset of a symbol in its section, which can always be evaluated on the first pass. If you are linking OFFSET returns the offset of the symbol from the base of the current module's contribution to the section, so sect(x)+offset(x) will not equal x. This has been done to allow OFFSET to be evaluated at assemble time, which is very useful for things like :-

```
                   if offset(*)&1          ; If current PC is odd
                   dc.b          -1        ; add a minus one
                   endif
```

Note the use of offset(*) to get offset into section of current PC. If you want the true offset into section whilst linking use x-section(x).

Setting Group ORG's

It is possible to set the ORG address of a group totally independantly of the addresses of the other groups.

Example

```
Code               group
Data               group
Bss                group        Bss
Debug              group        org($80000)        ; Debug code 512K
```

The groups without the org attributewill be placed sequentally at the address given in the org directive as usual.

Overlaying groups

It is possible to have several groups starting at the same address using the OVER attribute. All of the groups will have the same start address and enough room will be left for the largest group.

Example

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

Writing Groups to Files

It is possible to write groups to seperate pure binary files whilst leaving the other groups to be written to the normal output. This facility is used for overlay files and loadable data files.

Example

```
Code            group
Data            group
Overlay1        group       org($80000),file('Overlay1.bin')
Overlay1b       group       ; This also goes into Overlay1.bin
Overlay2        group       org($80000),file('Overlay2.bin')
```

Note that all groups declared after a group that has a file attribute are also written to this file.

The FILE attribute can be used in conjunction with the OVER attribute to put overlays into seperate files with the start addresses of the overlays all being the same.

Machine Specific Relocatable Formats

ST

If you are using the linker to produce .TOS, .PRG or .TTP files you should declare the following groups.

```
Text            group
Data            group
Bss             group       bss
```

The order is not important as the linker declares them implicitly.

If you do not need to have multiple sections within each group you can omit the group declarations and use the default group names to identify the sections.

Example

```
                section     Text
                move.w      #234,d0

                section     Data
                dc.w        99
```

If yo do this you loose the ability to have multiple sections in each group and the linker has to perform the checks for initialised data in BSS groups.

Amiga

If you are using the linker to produce an Amiga executable file you should declare the following groups.

```
Code            group
Data            group
Bss             group       bss
Code_f          group
Data_f          group
Bss_f           group       bss
Code_c          group
Data_c          group
Bss_c           group       bss
```

The order is not important as the linker declares them implicitly.

If you do not need to have multiple sections within each group you can omit the group declarations and use the default group names to identify the sections..

Example

```
                section     Code
                trap        #0

                section     Data_c
Sprite0         dc.w        34
```

If yo do this you loose the ability to have multiple sections in each group and the linker has to perform the checks for initialised data in BSS groups.

Linking

SNLINK is a fully featured linker which allows you to write your program in several separate modules and then link these modules together to produce the final program.

Command line syntax

```
SNLINK /switches source(s),outputfile,symbolfile,mapfile,libraries
```

Switches	/d	Debug mode - do not run
	/e symb=value	Equate symbol
	/i	Information window
	/m	List external symbols to map file
	/o Number	Set org address
	/o ?Number	Ask target for memory for org address
	/p	Produce pure binary output
	/r Format	Produce machine specific relocatable file
	/x Number	Set execute address

Relocatable formats are currently ST for .TOS files or AM for Amiga hunk format files.

SourceFile One or more source files (produced by the assembler) separated by either spaces or plus signs (+) or the name of a linker command file preceded by an at sign (@)

Outputfile File to receive object code output. If this is omitted no object code will be generated. If you want the object code to be sent to a target computer use T?: as the objectfile where ? is the SCSI device number of the target.

Symbolfile File to receive symbol table information for debugger.

MapFile File to receive map information

Libraries Library files to search

Linker command files

Linker command files contain instructions that tell the linker which object files to read, where to org them and information about groups. These instructions have a very similar format to the instructions in the assembler for consistency.

The following instructions can be used:-

```
; Comment         Comment line
                  include     Filename         ; read object file
                  inclib      Filename         ; search library file
                  org         Addr             ; ORG address of program
                  org         ?size[,type]     ; ask target for memory
                  regs        pc=addr          ; set execution address
name              group       [attributes]     ; define group
                  section     name[,group]     ; define section
                  workspace   addr             ; move workspace on target
name              equ         value            ; equate symbol
```

Groups are placed in memory in the order in which they are declared, if a group is declared in the program which is not declared in the command file it is placed on the end of the defined groups.

Sections within each group are placed in memory in the order in which they are specified, if a section is used in the program which is not declared in the command file it is placed on the end of the defined sections in the appropriate group.

Parts of a section from different source files are concatenated in the order in which the source files are specified.

When groups are declared in your source code there is no need to specify any attributes other than WORD and BSS; all other attributes are ignored when linking. If these attributes are speified in the source they must also be specified in the linker command file and vice versa.

Example linker command file

```
; Command file for Amiga Widget sorter
            include    Input.obj
            include    Sorter.obj
            include    Output.obj
            org        1024
            regs       pc=ProgStart
LowGroup    group      Word
CodeGroup   group
BssGroup    group      Bss
            section    Data1,LowGroup
            section    Data2,LowGroup
            section    Code1,CodeGroup
            section    Code2,CodeGroup
            section    Tables,BssGroup
            section    Buffers,BssGroup
```

The groups and sections will be in the following order :-

```
; Start of executable at 1024
LowGroup    Data1
            Data2
```

```
CodeGroup  Code1
           Code2
BssGroup   Tables
           Buffers
; end of executable
```

The same group attributes are allowed as for the GROUP directive in the assembler.

XDEF, XREF and PUBLIC

When you are linking several modules together you will want to refer to symbols defined in one module in different module. To do this you must declare symbols as external in the module in which they are defined using XDEF, and then in the module in which they are used you must use XREF to tell the assembler that the symbol is in a different module. Any expression that contains an XREF'd symbol will not be fully evaluated by the assembler and will, instead, be resolved by the linker.

You can tell the assembler that a symbol can be accessed using absolute word addressing by specifying the word size on the XREF directive.

Example of XREF and XDEF

Source module 1

```
                xref.w      LargeTable
                xdef        ARoutine,ANOther
                section     Code1,Text
ARoutine        lea         LargeTable,a0
                ...
ANOther         mulu        d0,d0
```

Source module 2

```
                xdef        LargeTable
                xref        ARoutine,ANOther
                section     Tables,BssGroup
LargeTable      ds.w        100
                section     Code1,Text
                jsr         ARoutine
                jsr         ANOther
```

If you want to declare a large number of symbols as external you can use the PUBLIC directive to tell the assembler that all further symbols should automatically be XDEF'd.

Example

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

Libraries

A object module library is a file containing several object modules. These libraries can be searched by the linker if it cannot find a symbol in the object files. If the 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.

SNLIB is a utility program for creating and maintaining object module libraries.

Command line syntax

SNLIB /switch libraryfile modules

Switches	/a	Add modules to library
	/d	Delete modules from library
	/u	Update modules in library
	/x	Extract modules from library
	/l	List modules in library

The SCSI Link

Installing the software

SCSILink is installed either from the command line or from a batch file (usually autoexec.bat), and it must be installed before any of the assemblers or debuggers can be used.

SYNTAX:
SCSILink CardAddr[,D?][,I?]

The address is whatever address you set using links 0-7 on the PC card, and the number after the D should be the DMA channel you are using.

If you have left the links as supplied just insert :-

```
SCSILink 390,D1
```

If no DMA channel is specified hardware DMA will not be used and unless you have a fast computer data transmission speed will be reduced.

The I parameter is used to give the card a different SCSI device number without opening the computer and changing the links; this will not usually be necessary.

Interface to SCSILINK

The SCSILink software is accessed using int 7dh. AH is used to select the required function as shown below.

AH=0 Reset SCSI Bus

Resets all devices on the SCSI bus.

In	-
Out	-
Errors	-

AH=1 Connect to target

Arbitrates for use of bus and selects target.

In	AL=Target ID
Out	-
Errors	CF set if error, AL=Initiator error, AH=Target error

AH=2 Send command

Sends the command block to the target and performs and related I/O.

In	ES:BX = pointer to parameter block shown below
Out	-
Errors	CF set if error, AL=Initiator error, AH=Target error

Parameter block: (All with 8086 byte ordering)

Size of command block	Dword
Offset of command block	Word
Segment of command block	Word
Size of buffer	Dword
Offset of buffer	Word
Segment of buffer	Word

The contents of the command block which are described later have most significant bytes and words first!

AH=3 Set TimeOut

Change one of SCSILink's internal timeouts to allow communication with very slow targets.

In	AL=TimeOut Number
	BX=New value (55ms ticks)

Out	-
Errors	-

Number	Function	Default
0	Time to wait for bus	18
1	Time to wait for new phase	5
2	Max time to send/receive block	18
3	Time to wait for reselect	180

AH=4 Get Error String

Get text to print when an error is reported.

In	AL=Error number
Out	ES:BX=pointer to zero terminated string
Errors	-

AH=5 Get Address

Get information about current hardware configuration.

In	-
Out	AL=Dma Channel, AH=Initiator ID, BX=Card Addr
Errors	-

AH=6 Put Data

Save data for later retrieval. (Stored in PSP of SCSILink)

In	CX=Byte count (1-256), ES:BX=Data to save
Out	-
Errors	CF set if error, AL=1 Too much data

AH=7 Get Data

Retrieve data stored using Function 6

In	CX=Byte count (1-256), ES:BX=Buffer to fill
Out	-
Errors	CF set if error, AL=1 Too much data

AH=8 Terminate session

If a program has made any use of these link commands it should issue a Terminate Session command as it exits even if it is exiting due to a SCSI error. This command exists so that any software that intercepts int 7Dh knows when other programs have finished with the link.

AH=9 Poll Request

Whenever a program performs a GetRegs command and the exception that has occured is not one that it specifically handles it should perform a Poll Request call to see if any driver that has chained onto int 7Dh handles it. This service exists so that things such as resident disk servers can continue to run even whilst debuggers and profilers are running. The resident driver assumes that a connect has been performed and tries to leave the target connected.

In	AL=Target ID
Out	AL=-1 if a resident driver handled event AL=-2 if a resident driver experienced a SCSI error
Errors	-

SCSILINK Command protocol

The following is a list of the commands currently defined. Not all of these commands are currently supported - the SendImpl command lets you check what commands the software at the other end supports. Some of these commands will only be handled by downloaders which leave the machines OS resident, and these downloaders usually support some extra commands to support file handling and multi-tasking.

If the top bit of a command byte is set the target will disconnect after executing the command and reporting the status. This means the software can support up to128 commands.

Commands		Bytes	Data	Action
0	Noop	2	-	Do nothing except report status
1	SendImpl	2	In	Send binary array of valid commands
2	SendID	2	In	Send machine/processor ID
3	SendAddr	2	In	Send start and length of link workspace
4	RecvAddr	10	-	Change address of link workspace
5	FindRam	10	In	Find how much memory is free
6	ReserveRam	10	In	Ask OS for some memory
7	FreeRam	10	-	Hand memory back to OS
8	ReBoot	2	-	Reboot target
9	MakeSafe	2	-	Prepare for Re-send
10	Return	2	-	Return to OS
32	SendMem	10	In	Send a block of memory

33	RecvMem	10	Out	Receive a block of memory
34	VerifyMem	10	Out	Verify a block of memory
35	CheckSum	10	In	Generate checksum of a block of memory
40	SendRegs	2	In	Send register block
41	RecvRegs	2	Out	Receive register block
48	GoPC	2	-	Invoke code at PC without return addr
68	GetOSErr	2	In	Get result of last OS function

Format of 2 byte commands

Byte	Function
0	Command byte
1	Modifier (set to zero if unused)

Format of 10 byte commands

Byte	Function
0	Command byte
1	Modifier (set to zero if unused)
2-5	Length field
6-9	Address field

Commands in detail

Noop

Command	0
Modifier	0

Function

Does nothing. However this can be used to disconnect the target by setting the top bit as usual.

SendImpl

Command	1
Modifier	0

Data Transfered	Offset	Length
Binary array of commands	0	16

Function

Target sends 16 byte binary array indicating which of the 128 commands it supports. The data is bigendian so the first byte is for commands 127-120.

SendID

Command	2
Modifier	0

Data Transfered	Offset	Length
Processor ID	0	12
Machine id	12	12

Function

Target sends string giving processor and machine ID.

SendAddr

Command	3
Modifier	0

Data Transfered	Offset	Length
Start of link memory	0	4
Length of link memory	4	4

Function

The link software reports the address and length of the block of memory it is using. If the link software doesn't use any ram the length will be zero.

RecvAddr

Command	4
Modifier	-
Length	-
Address	New address for link memory

Data Transfered	Offset	Length
New address	0	4

Function

Causes link software to relocate itself.

FindRam

Command	5
Modifier	0
Length	-
Address	Type of RAM to reserve (Target specific)

Data Transfered	Offset	Length
Ram free	0	4

Function

Asks the targets OS the size of the largest block of contiguous memory that it can allocate.

ReserveRam

Command	6
Modifier	0
Length	Amount of ram to reserve
Address	Type of RAM to reserve (Target specific)

Data Transfered	Offset	Length
Memory handle	0	4
Address of ram	4	4

Function

Target machine asks OS for ram and passes back a handle which will be required to later free the memory and the base address of the block of memory.

FreeRam

Command	7
Modifier	0
Length	-
Address	Memory handle

Function

Hand memory back to OS

ReBoot

Command	8
Modifier	0

Function

Target machine reboots and goes through normal boot process

MakeSafe

Command	9
Modifier	0

Function

Target machines puts the hardware into the safest configuration possible. This usually involves disabling all DMA and interrupts, going out of polled mode and freeing any reserved ram.

Return

Command	10
Modifier	0

Function

Target returns to OS

SendMem

Command	32
Modifier	0
Length	Size of block in bytes
Address	Address of block in target

Data Transfered	Offset	Length
Stream of bytes	0	length given in command

Function

Target sends a block of ram to initiator

RecvMem

Command	33
Modifier	0
Length	Size of block in bytes
Address	Address of block in target

Data Transfered	Offset	Length
Stream of bytes	0	length given in command

Function

Target receives a block of ram from initiator

VerifyMem

Command	34
Modifier	0
Length	Size of block in bytes
Address	Address of block in target

Data Transfered	Offset	Length
Stream of bytes	0	length given in command

Function

Target receives a block of ram from initiator but just checks it against current ram contents.

CheckSum

Command	35
Modifier	0
Length	Size of block
Address	Address of block in target

Data Transfered	Offset	Length
Additive checksum of bytes	0	4

Function

Target adds together all the bytes in the defined range and returns the long word result

SendRegs

Command	40
Modifier	0

Data Transfered	Offset	Length
Register block	0	82

Function

The target sends its copy of the 68000 registers to the initiator.

Register block format is :-

```
regd0        ds.l      1
regd1        ds.l      1
regd2        ds.l      1
regd3        ds.l      1
regd4        ds.l      1
regd5        ds.l      1
regd6        ds.l      1
```

```
regd7          ds.l     1
rega0          ds.l     1
rega1          ds.l     1
rega2          ds.l     1
rega3          ds.l     1
rega4          ds.l     1
rega5          ds.l     1
rega6          ds.l     1
regssp         ds.l     1
regusp         ds.l     1
regpc          ds.l     1
regsr          ds.w     1
regextype      ds.b     1      ; Exception type (-1=Startup,
                                 1=Running)
regfunccode    ds.b     1      ; Exception function code
regerroraddr   ds.l     1      ; Address that caused error
reginst        ds.w     1      ; Instruction that caused error
```

RecvRegs

Command 41
Modifier 0

Data Transfered	Offset	Length
Register block	0	82

Function

Initiator sends new register block to target.

GoPC

Command 48
Modifier 0

Function

Target restores all register from register block and jumps to PC.

GetOSErr

Command 68
Modifier 0

Data Transfered	Offset	Length
OS error code	0	4

Function

Returns the error code for the last OS function the link software used.

Using SnMake

SnMake was designed to allow Snasm and associated Cross Products software tools to be used from within the Brief editor with the maximum of ease.

SnMake works on the time honoured make utility principle of reading a file supplied by the user in which are defined the relationships between the target(s) the user wishes to create and the source files from which that target is to be created. The target will be said to be dependant upon its source files which are generically known as dependencies. The file in which these relationships are defined is known as the 'makefile', although in discussing SnMake we shall also refer to them as 'project files' as will become clearer later. Once SnMake has read this file it determines which targets have dependants which have been updated since the target file was created, and thus which targets must be recreated from their dependants. SnMake discovers how to recreate the targets from rules specified in the makefile as discussed below. SnMake has a number of features that allow it to work in close conjunction with Snasm and other Cross Products development tools but which mean it differs from the more normal make utility syntax in a number of areas.

SnMake and Brief

Cross Products have produced a number of macros for the Brief editor to enhance the environment for the Snasm user. Some of those macros interface the editor with SnMake and allow the user to invoke the utility from within the editor. Note that the new macros do away with the dialog-box interface to Snasm and any user wishing to use Snasm from within Brief using the standard macros will have to use SnMake as described below. This is not, however, a complex matter.

The following description assumes that the Brief macros are being used as supplied, without any reallocation of key-bindings.

The Snasm menu is invoked within Brief with the Alt-F9 keys. The menu item 'Select Project File' will bring up a window displaying all the files in the current directory which have a '.prj' extension. The first line of these files is displayed to help the user remember what function they perform. This is the mandatory extension for files to be used as SnMake makefiles and will be referred to as project files in the remainder of this section. Should Brief be unable to find any project files in the current directory it will display a message on the status line to that effect. SnMake can be invoked from the command line with the /p switch set (see below for details of switches and command line usage) in which case it will attempt to append a '.prj' extension to the makefile name it is given. In this mode it is said to be in 'project mode'. When invoked from Brief SnMake is always in project mode.

Creating project files

Project files must contain a label beginning in the first column of the form:

[SnMake]

SnMake will ignore any text before this label and if it finds the end of file before encountering it will issue an error and exit. Within Brief this will result in an error window appearing . The search for this label is not case sensitive.

Once it has discovered this label SnMake will regard everything following as valid input until it encounters another '[' in the first column or the end of file.

Two other valid labels are currently supported; [Debug] and [Eval]. They only have significance if the project file is selected within Brief. The next non-blank line following the [Debug] label is deemed to hold the Debugger command line the user wishes to have issued upon invoking the 'Debug' menu selection from the Snasm menu (see 'Using SNASM within Brief' for details on using the new Snasm macros within Brief) and similarly for the line following the [Eval] label which invokes the expression evaluator of the users choice. In the case of the [Eval] command line a special token '$$$' can be placed in the string and any marked input will be placed in the string at that point.

Thus a simple project file might look as follows:

```
-------beginning of file here-----------
project file to assemble prog.68k to t7:

[snmake]

t7:;    prog.68k
        snasm68k $! prog.68k,t7:,prog.sym

[Debug]
        snbug68k prog.sym
[Eval]
        snbug68k /v$$$ prog.sym
```

Defining targets.

SnMake regards anything starting in column 0 and terminated with a ';' as a target declaration. Thus:

```
target1;
tgt1;
e:thistgt;
```

are all valid target names. White space in target declarations is stripped out.

Defining dependencies.

Anything following the ';' on the same line as a target declaration is regarded as a dependency declaration. Multiple dependencies are declared separated by white space and line continuation (see below) can be used.

The following line:

```
target1.cpe;  src1.68k src2.68k
```

declares that target1.cpe is dependant on src1.68k and src2.68k and SnMake will attempt to invoke any rules defined for target1.cpe if the date/time stamps of either src1.68k or src2.68k show them to have been updated since target1.cpe was last created.

Special targets.

SnMake supports a number of special targets.

.INIT; any commands following this target declaration will always be carried out first when this project file is executed.

.DONE; any 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 makefile, 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.

```
t7:;            src1.68k
        snasm68k $! src1.68k,t7:
```

declares that src1.68k should be assembled down to SCSI target t7: . This will occur every time the project file is executed. See section 'Special Macros' for the significance of '$!' in the command.

.RESOURCE; this target declaration should be followed by a list of programs that are able to utilise resource files. It must be the first item declared int the project file after the [SnMake] label.

If a command line for one of these programs exceeds 128 characters in length, SnMake will split it down into a temporary file with each command line argument placed on a new line and will call the command with that file preceded by an '@' character. i.e.

```
[SnMake]
.RESOUCE;        somecmd

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

detecting an over-long command line SnMake will create a temporary resource file and call somecmd as follows:

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

Defining explicit rules.

Anything following the end of a target declaration line is considered a rule declaration. To be valid; a rule declaration 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 and indented as described above.

```
target1.obj;  dep1 dep2
       snasm68k  /l dep1 dep2,target1.obj
```

defines a rule telling SnMake that if target1.obj is younger than either dep1 or dep2 it should issue the command
'snasm68k /l dep1 dep2,target1.obj'

Defining implicit rules.

If SnMake cannot find an explicit rule with which to create a target it will attempt to do so by invoking any implicit rules defined in the project file. Implicit rule definitions must begin in the first column and begin with a '.' character. SnMake, on searching the list of implicit rules it has defined, looks for a rule that matches the extension of the target it is trying to make. This is the first portion of the implicit rule declaration. Thus to define an implicit rule that SnMake will use to deal with any targets with a '.obj' extension the following is the required first part of the declaration:

```
.obj
```

having found this SnMake will proceed to attempt to match the second part of the declaration; this is delimited from the first part by a ',' character thus:

```
.obj,.68k
```

this informs SnMake that any targets with a '.obj' extension are to be created from a dependency of the same name but with a '.68k' extension. The second part of the declaration must begin with a '.' character, unless a path name is specified as below:

```
.obj,e:temp\.68k
```

This tells SnMake that any targets with a '.obj' extension are to be created from a dependency of the same name but with a '.68k' extension in directory e:temp.

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. These are $+ and $-. $+ signifies the target and $- its dependency. Thus the following implicit rule

```
.obj,.68k
        snasm68k $+,$-
```

when invoked upon target prog1.obj will result in the following command:

```
        snasm68k  prog1.68k,prog1.obj
```

Points to note on using implicit rules.

a. Explicit rules will always be used in preference to implicit rules if explicit rules have been set for a given target.

b. If more than one set of implicit rules are defined for the same target group, that implicit rule most recently defined (in terms of position within the project file) will be invoked upon any suitable targets thus:

```
.obj,.68k
```

any suitable targets in this area will be created from files of the same name with a '.68k' extension.

```
.obj,e:temp\.68k
```

following this new implicit rule definition any suitable targets in this area will be created from files of the same name with a '.68k' extension in directory e:temp.

Macros.

Macros can be passed into SnMake from the command line using the /e switch as described below in the Command Line Options section. Within a make/projectfile the can be defined using the following syntax:

```
macro1=somename
```

macro definitions must begin in the first column of the line. White space is stripped out of macro definitions.

When defined macros are referenced they must be identified using the following method :

$(macroname)

thus given the above macro definition $(macro1) will expand to 'somename'. '$' 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.

$e(macroname) expands to the extension of macroname.

macroname=test.obj
$e(macroname) expands to '.obj'

$n(macroname) expands to only the filename of the macro definition

macroname=e:test\test.obj
$e(macroname) expands to 'test.obj'

$p(macroname) expands to the pathname of the macro definition

macroname=e:test\temp\test.obj
$p(macroname) expands to 'e:test\temp\'

$d(macroname) expands to the drive name of the macro definition

macroname=e:test\temp\test.obj
$d(macroname) expands to 'e:'

$b(macroname) expands to the filename in the macro definition

macroname=e:test\temp\prog1.obj
$b(macroname) expands to 'prog1'

Line continuation.

To continue a line without introducing a newline character the '\' character immediately followed by a carriage return is used. Thus

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

declares dep1 to dep11 as dependencies to thistgt. Without the continuation mark SnMake would truncate the dependency list at dep9.

Comments.

A line on which the first character is a hash mark will be regarded as a comment line.

Special macros.

For use with Snasm and Snlink a special macro is provided by SnMake which allows the setting of the /i and /d switches on the command lines of those programs.

The /i switch to Snasm and Snlink forces them to create an output window to which they send their output whilst running. This enables the user to watch their progress from within Brief. This option is always set by the Brief macros supplied by Cross Products.

The /d switch to Snasm and Snlink forces them into debug mode, if a file is being assembled/linked to a target machine it will not be run if this switch is set; thus allowing the user to enter the debugger before execution of that code starts. This option can be controlled from the Brief Snasm menu using the 'Set Debug Mode' menu item.

Control of these switches is not possible from within Brief if the $! macro is not present on the Snasm68k and Snlink command lines in the project file as shown below:

```
targ1; dep1 dep2
        snasm68k $!  /l dep1 dep2,targ1
```

assuming that debug mode is set to 'ON' using the 'Set Debug Mode' option from Brief this command will expand to

```
        snasm68k /i /d /l dep1 dep2 ,targ1
```

In addition the /d and /i switches set up two macros, Debugstr and Infostr, which can be tested with the !ifdef command as described below.

Conditionals.

A conditional capability is provided within SnMake by the !ifdef command. This allows the user to determine if a macro is defined and act upon that information. Thus in the following case:

```
t7:;            prog1.68k
        snasm68k $! prog1.68k,t7:,prog1.sym
!ifdef(Debugstr)
        snbug68k prog1.sym
!endif
```

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. If SnMake reaches the end of the project file with an unbalanced number of calls to !ifdef and !endif an error will result.

An !else statement is also supported as shown below:

```
!ifdef(Debugstr)
t7:;    prog1.68k
        snasm68k  $! prog1.68k,t7:,prog1.sym
        snbug68k prog1.sym

!else
t7:;    prog1.68k
        snasm68k prog1.68k,t7:
!endif
```

This example performs the same task as the previous one in a slightly more round-about manner.

Invoking SnMake from the command line

The syntax for calling SnMake from the command line is as follows:

```
        snmake <switches> <makefile> <errorfile>
```

all arguments on the command line are optional. Invoking SnMake with no arguments causes it to look for a makefile called 'makefile' and process that. The optional <makefile> parameter specifies an alternative makefile name. If <errorfile> is specified all error information will be output to that file.

Switches :

SnMake accepts five switches from the command line.

/q Quiet mode. No echoing is done as SnMake proceeds.

/p Project mode. This forces SnMake to treat makefiles as project files i.e. as if invoked from within Brief. If no makefile name is specified SnMake will default to makefile.prj. In project mode all output from SnMake goes to a file called snmk.err, any error output from the programs invoked by SnMake will appear on-sceen unless an error file is specified.

/d Set debug mode. Sets the special macro $!. Only of use if invoking snasm or snlink from the makefile.

/i Set info mode. As above.

/e Pass a macro definition into SnMake. Syntax /ename=Fred. This would set up a macro name which would expand to Fred.

/b Build all. All rules carried out regardless .

Examples:

The following example is a project files that will produce a file called test1.cpe from the source files e:src1.68k e:src2.68k . *Text in this style is comment text added to aid the reader and is not part of the SnMake syntax.*

```
-------------------Project file----------------------
file to create test1.cpe       This text will appear in the project file
                               select menu

[SnMake]                               Snmake in project mode starts reading at
                                       this label
src1.obj;            e:src1.68k
        snasm68k $! /l /z e:src1.68k,src1.obj,src1.sym

src2.obj             e:src2.68k
        snasm68k $! /l /z e:src2.68k,src2.obj,src2.sym

test1.cpe            src1.obj src2.obj
        snlink $!  src1.obj src2.obj, test1.cpe

[Debug]                                         SnMake stops reading here.
        Snbug68k src1.sym src2.sym     Debugger is invoked using this
                                       string
[Eval]
        Snbug68k \v$$$ src1.sym src2.sym     Expression evaluator is
                                                  invoked with this string
------------End of project file---------------
```

Other Utilities

SnCopy

This utility is a variant on the copy command in that it will only copy those source files that have been more recently written than any target files it finds of the same name.

It has the following command line syntax:

```
sncopy <switches> source1+source2+....  target directory
```

where switches are:
/a - refresh all; regardless of relative ages (refresh All mode).
/q - do not echo after refresh (Quiet mode).

Thus:

```
sncopy file1.68k+file2.68k c:\source
```

would cause sncopy to search in directory c:\source for files named file1.68k and file2.68k and compare their time and date stamps (if the files exist) to those of the files of the same name in the current directrory. If the files in the current directory are newer than those in c:\source or files of the same name do not exist in c:\source a copy will take place.

In the following case, due to the use of the /q and /a switches all files are copied and no echoing is done to the screen.

```
sncopy  /q/a file1.68k+file2.68k c:\source
```

SnCopy supports wildcard patterns:

```
sncopy *.68k d:\src
```

will copy all file ending in the '.68k' extension into the directory d:\src if no younger file of the same name is found residing there.

SnCopy will be of most use to those who keep copies of source files on a fast device, such as a ram disk, whilst they work on them and will thus benefit by using a copy program that will not allow the accidental copying of old source files over those that have been more recently updated.

Index