MOTOROLA DIGITAL SIGNAL PROCESSING DEVELOPMENT SOFTWARE

MOTOROLA DSP SIMULATOR REFERENCE MANUAL

Motorola, Incorporated
Semiconductor Products Sector
DSP Division
6501 William Cannon Drive West
Austin, TX, 78735-8598

This document contains information on a new product. Specification and information herein are subject to change without notice. Motorola reserves the right to make changes without further notice to any products described in this document to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein, neither does it convey any license under its patent rights or the rights of others. Motorola is a registered trademark of Motorola, Inc. Motorola, Inc. is an Equal Employment/Affirmative Action Employer.

TABLE OF CONTENTS

Chapter 1
DSP SIMULATOR

1.1 INTRODUCTION

The DSP Simulator program is a software tool for developing programs and algorithms for Motorola Digital Signal Processors (DSPs). This program exactly duplicates the functions of supported Motorola DSP chips, including all on-chip peripheral operations, all memory and register updates associated with program code execution, and all exception processing activity. The device's pipelined bus activity is exactly simulated. This enables the Simulator to provide the user an accurate measurement of code execution time, which is so critical in DSP applications.

The Simulator executes object code which can be generated using either the device Macro Assembler program or the Simulator's internal single-line assembler. The object code is loaded into the simulated device's memory map. The entire internal and external memory space of the DSP is simulated. During program debug the user can display and change any of the device's registers or memory locations. Instruction execution can proceed until a user-defined breakpoint is encountered, or in single-step mode, stopping after a specified number of instructions or cycles have executed.

1.2 FEATURES

Summary of Simulator features:

- Multiple device simulation
- Source level symbolic debug of assembly and C source programs
- Conditional or unconditional breakpoints
- Program patching using a Single-Line Assembler/Disassembler
- Instruction and Cycle timing counters
- Session and/or Command Logging for later reference
- Input/Output ASCII files for device peripherals
- Help file and Help line display of Simulator commands
- Macro command definition and execution
- Display Enable/Disable of Registers and Memory
- Hexadecimal/Decimal/Binary calculator

1.3 OPERATING ENVIRONMENT

The minimum hardware requirements for the DSP Simulator include:

- **IBM AT*** 386 or better) with **2 Mb** of **RAM**
- **PC-DOS / MS-DOS**** v3.0 or later.
- **IBM,AT** and **PC-DOS** are trademarks of International Business Machines.
- **MS-DOS** is a trademark of Microsoft Corp.

The Simulator supports all of the external memory maps of the DSP. It is compiled with a Compiler which supports extended and virtual memory on the PC. The file readme.mem will contain additional information for configuration of the PC to support the memory management.

Floppy diskette drives are adequate for small simulations. However, due to the virtual memory paging scheme and since many of the INPUT and OUTPUT commands reference disk files, a fixed disk drive is highly recommended.

If your simulation involves many assigned disk files, the operating system's limit of the number of open files may be reached. This will cause the simulation to slow down while files are closed and then reopened for accesses. In order to reduce the chance of this situation occurring, it is recommended that your operating system's CONFIG.SYS file be modified with the following MS-DOS configuration commands:

```
BUFFERS = 32
FILES = 20
```

These commands increase the number of disk memory buffers and the maximum number of files that can be open at one time.

1.4 RUNNING THE SIMULATOR

The format for invoking the Simulator is:

SIMDSP [macro command filename]

Although the name **simdsp** is used throughout this manual for example purposes, the actual name of the Simulator is device dependent. For example, the DSP56000 and DSP56001 devices use the Simulator named **sim56000**, while the DSP56116 device uses **sim56100**. See Chapter 8, Simulator Names for the actual name used for your device Simulator.

The macro command filename is an optional parameter. The macro command file should contain a sequence of commands that the user wishes to execute upon Simulator start-up and prior to command entry from the keyboard. If an incorrect command is encountered in the macro command file, the macro command will terminate and command entry will be enabled from the keyboard. Macro command files can be nested (a macro command file can call another macro command file) to any level.

If you do not specify a suffix in the macro command file name, the Simulator will assume the suffix ".cmd".

EXAMPLES

SIMDSP
Invoke the Simulator. Begin keyboard command input immediately (no macro file).

SIMDSP STARTUP
Invoke the Simulator and run the macro file named "STARTUP.CMD".

SIMDSP SETUP.N5
Invoke the Simulator and run the macro file named "SETUP.N5".

SIMDSP SETUP5.
Invoke the Simulator and run the macro file named "SETUP5.".

1.5 USER INTERFACE

The bottom three screen lines function as the command line, an error message line, and a help line.

As each valid command is accepted from the command line, it and its results are scrolled into the display screen. The last 100 lines of display screen entry are available for review at any time by typing **Pg Up** (Ctrl-T), **Pg Dn** (Ctrl-V), **Up-Arrow** (Ctrl-U) or **Down-Arrow** (Ctrl-N).

1.6 COMMAND ENTRY

Upon entry into the Simulator, several of the available commands are displayed on the help line. The remaining commands can be reviewed by pressing the **SPACE** bar when the cursor is at the start of the command line.

The Simulator requires a minimum number of key strokes to recognize a Simulator command. The minimum number of required characters for each command is shown highlighted on the help line. A command can be specified by typing the required characters followed by a space or by typing the entire command word followed by a space.

Entering the command key strokes followed by a space will activate the help line for that particular command. The help line shows the syntax for the remainder of the command. Additional help and examples of the current instructions can be obtained by typing a question mark at any point during the command entry.

Any text following a semicolon on the command line is considered to be a user comment. This provides the user a means of documenting session display.

Command execution begins when the user types the **ENTER** or **CARRIAGE RETURN** key. If the entered command is not one of the predefined Simulator commands, the Simulator interprets the command as a macro file name and executes the macro file. Macro command files can be created by logging command entries. This procedure is explained in the documentation of the Simulator **LOG** command.

Command line editing is supported for command entry corrections. The cursor can be moved on the command line by using the **Left-Arrow** (Ctrl-L) and **Right-Arrow** (Ctrl-R) keys. The grey **Back-Arrow** (Ctrl-H) key on the upper right of the keyboard will backspace and delete the previous character. The **Del** (Ctrl-K) key will delete the following character. The **Ins** (Ctrl-O) key can be used to toggle between insert and overwrite modes of character entry. The **ESC** key will clear the command line.

The **CONTROL-C** or **CONTROL-BREAK** keys can be used to abort the execution of a Simulator command.

Once a valid command is entered it is stored in a holding buffer for repeated execution. To execute the previous valid command the user need only type the ENTER or CARRIAGE RETURN key.

The previous ten commands can also be recalled for editing or execution by typing **Ctrl-B** or **Ctrl-F**. **Ctrl-B** moves backward through the circular list of ten previous commands; **Ctrl-F** moves forward through the list.

1.7 DISPLAY MODES

The Simulator supports three display modes - register, assembly and source. These modes determine the Simulator display at the termination of commands which initiate device cycle execution. The register display mode causes the display of register and memory locations enabled by the **DISPLAY** command. The assembly display mode causes the display of one full screen of disassembled instructions containing the instruction at the current execution address. The source display mode causes the display of one page of the original source file which contains the source line associated with the current execution address. In both the assembly and source display modes the position of the current execution address is marked by **=>** in the left margin.

The source display mode requires symbol and line information in the object file that will normally be the result of assembling with the -g option of the assembler. See the assembler manual for instructions on the use of the -g option.

A display mode can be selected either by the Simulator **VIEW** command, or by toggling among the display modes using the **Ctrl-W** key entry (hold down Ctrl and press w). In addition, Simulator commands which display registers or memory, or otherwise create display to the register display window will select the register display mode; and the Simulator **LOAD** and **LIST** commands will switch from the register display mode to the source display mode.

Chapter 2
SIMULATOR COMMANDS

2.1 COMMAND OVERVIEW

There are a total of twenty-five Simulator commands. These can be grouped into four functional categories: **memory/register modification**, **file I/O**, **simulation execution control**, and **miscellaneous tasks**. An additional group of fourteen commands is available in the **GUI** version of the Simulator for **windows control**.

2.1.1 Memory/Register Modification

There are eight memory/register modification commands. These allow the user to **ASSEMBLE** (ASM) DSP instructions, **CHANGE** register or memory locations, **COPY** a block of memory to a new location, **DISASSEMBLE** code stored in the simulated DSP memory space, **DISPLAY** registers and memory values, **DISPLAY** the Simulator revision number or memory configuration, or **RESET** the device registers or memory space. The **HISTORY** command disassembles and displays the previous thirty-two instructions that were executed by the device. A **WATCH** list may be used to display a variable whenever single stepping or program execution is halted

2.1.2 File I/O

There are five file I/O commands available which allow the user to **INPUT** peripheral or memory location values from a file, **OUTPUT** peripheral or memory location values to a file, **LOAD** macro-assembler object module files or previous simulation state files, **LOG** Simulator commands, session display output or DSP program execution profile, and **SAVE** Simulator memory to an object module file or the Simulator state to a state file.

2.1.3 Simulation Execution Control

There are seven simulation execution control commands. These allow the user to specify **BREAK** conditions, to **GO** until a break condition is met, to **STEP** a specified number of instructions or cycles before displaying register and memory changes, or to **TRACE** a specified number of instructions or cycles displaying register and memory changes at each step. The **NEXT** instruction operates essentially the same as the STEP instruction,

except that if the instruction being executed calls a subroutine or macro, execution continues until return from the subroutine or macro. The **UNTIL** instruction has the effect of setting a temporary breakpoint at a specified address, executing until a breakpoint is encountered, then clearing the temporary breakpoint. The **FINISH** instruction proceeds until an RTS instruction is encountered for the current subroutine.

2.1.4 C Source Code Debug Commands

There are seven C source code debug commands available. The user may use **WHERE** to display the C function call stack. The user can then use **UP**, **DOWN** and **FRAME** to traverse the call stack. The user may **REDIRECT** data from stdin/stdout/stderr to files when **STREAMS** are enabled and the user may also display the data **TYPE** of a variable, function or C expression.

2.1.5 Miscellaneous Tasks

There are eleven miscellaneous task commands available which allow the user to create a new **DEVICE** and specify the device type, **EVALUATE** expressions in five different radices, get **HELP** for command line entry, define a default **PATH** name for storage of temporary files, **QUIT** a simulation session, specify the default numerical **RADIX** used during expression evaluation and data entry or data display, execute a **SYSTEM** command, or **WAIT** a specified number of seconds before proceeding to the next instruction. The **LIST** command displays a specified source file when symbolic debug is in effect. The **VIEW** command allows selection of the Simulator display mode - Source, Assembly or Register. The **UNLOCK** command provides password enabling of unannounced device types.

2.2 COMMAND SYNTAX

The command descriptions in Section 2.3 each begin with a command syntax line showing the general form of the command. The command syntax line contains special punctuation to indicate command keywords, required or optional fields, repeated fields, and implied actions. The following is a description of the special punctuation within the syntax line:

Square brackets **[ ]** enclose optional command parameters. The brackets themselves are not entered as a part of the command. For example, in the "**W**AIT [count(seconds)]" command the **count** parameter is optional.

The slash **/** is used to separate alternate command parameters. The user may only enter one of the parameters in the list. The slash is not entered as a part of the command. For example, when entering the "**LO**G [**c**/**s**/**p**] filename" command, **log c filename** and **log s filename** are valid entries, but not **log c s filename**.

Parentheses **()** surround a description of an implied action. This is only included to help the user understand the action of the command. Neither the parentheses nor the description within are entered as part of the command. For example, when entering the "**CO**PY" command, **copy p:0..10 x:5** is a valid entry. The **from** and **to** words in the command syntax line are only an explanation of the direction of data transfer.

Three consecutive periods **(...)** indicate that the preceding field may optionally be repeated. For example, when entering the "**D**ISPLAY" command, multiple registers can be specified for display on the same command line.

Capitalized WORDS indicate command keywords. Command keywords must be entered exactly as shown. The portion of the command keyword shown in

BOLDFACE represents the minimum portion of the keyword that the user must type. The portion of a keyword not in boldface may be typed if desired, but is not required by the Simulator. The Simulator will type out the remainder of the keyword for you if you type the boldface characters followed by a space.

Other command parameters, shown in the command syntax line in **lower case** (but not within parentheses), are used in place of the expanded definitions shown in Section 2.2.1.

2.2.1 Expanded Syntax for Command Parameters

The following expanded definitions apply to the parameters shown on the command syntax line in lower case (but not within parenthesis):

addr = An address may be specified as a source file line number or as a symbol name if a previously loaded COFF object file contains symbolic debug information - see Chapter 5, Assembler Debug Symbols. Otherwise a memory space designator must be used. Use the Simulator's "help mem" command to obtain a list of the valid memory space prefixes.

addr_block =addr**..**location/addr#count

bn = (break number) decimal integer constant in the range 1 to 99.

break_action =**H**(halt)/**I**n(increment CNTn)/**N**(note)/**S**(show)/**X** [command]

count = positive integer expression in range 1 to $7fffffff.

dev_num = **dv0**..**dv31**

dev_type = see Chapter 8, Device Names

expression =any arithmetic expression valid for the assembler. In addition, the register names can be used in the expression.

c_expression =any expression valid in the current C program. A c_expression must be enclosed in curly braces.

file = any valid pathname for the operating system in use

ioradix = **-RD**(decimal)/**-RF**(float or fractional)/**-RH**(hexadecimal)/**-RU**(unsigned)

location = integer expression. It will be mapped into the device address range. For example, -1 translates to the maximum address.

mode = device operating mode in the form **Mn**. See Chapter 8, Operating Modes for a list of valid operating modes for the device.

pathname = any valid pathname for the operating system in use

periph = Valid peripheral names are displayed by the Simulator **help periph** command.

pin = Valid pin names are displayed by the Simulator **help pin** command. A pin name may optionally be preceded by **pin:** in order to resolve conflicts that may exist between pin and register names or constants.

pin_block = pin**..**pin

port = Valid port names are displayed by the Simulator **help port** command

reg = Valid register names are displayed by the Simulator **display all** command**.** A register name may optionally be preceded by **reg:** in order to resolve conflicts that may exist between register and pin names or constants.

reg_block = reg**..**reg

reg_group = periph/**all**

topic = on-line help topic keywords

2.3 COMMAND SUMMARY

The following is a summary list showing the syntax of the Simulator commands. A detailed description of each command is presented in the remainder of Section 2.3.

Memory/Register Modification

ASM [**B**(byte wide)] [(beginning at) addr] [assembler_mnemonic]
CHANGE [reg[_block]/addr[_block] [expression]]...
COPY (from) addr[_block] (to) addr
DISPLAY [**ON**/**OFF**/**R**/**W**/**RW**] [reg[_block/_group]/addr[_block]]...
DISPLAY **V**(version)
DISASSEMBLE [**B**(byte wide)] [addr[_block]]
HISTORY
RESET **S**(state)/**D**(device) [mode]
WATCH [#n] [radix] reg|addr|expression|c_expression
WATCH [#n] OFF

File I/O

INPUT [#n] [**T**(timed)] addr/port/periph/pin[_group] **OFF**/**TERM**/file [ioradix]
INPUT [#n] pin (from)[dev_num**:**]pin
INPUT [#n] addr (from)[dev_num**:**]addr
LOAD [**S**(state)|**M**(memory-only)|**D**(debug symbols-only)] (from) file
LOG [**OFF**] [**C**(commands)/**S**(session) [file [**-A**/**-O**/**-C**]]]
LOG [**OFF**] **V**(source display status line)
OUTPUT [#n] [**T**] addr/port/periph/pin[_group] **TERM**/file [ioradix/**-RS**] [**-A**/**-O**/**-C**]
OUTPUT [#n] [**T**] addr/port/periph/pin[_group] **OFF**
OUTPUT [#n] [**T**] **history OFF**/**TERM**/file [**-A**/**-O**/**-C**]
OUTPUT [#n] [**T**] **ehistory OFF**/**TERM**/file [**-A**/**-O**/**-C**]
SAVE **S**(state)/addr_block... file [**-A**/**-O**/**-C**]

Simulation Execution Control

BREAK [**#**bn] [expression] [break_action]
BREAK [**#**bn] **R**(read)/**W**(write)/**RW**(access) reg/addr[_block] [break_action]
BREAK [**#**bn[,bn,...]] **OFF**/**E**(enable)/**D**(disable)
BREAK [**#**bn] **DR**(dma read)/**DW**(write)/**DRW**(access) addr[_block] [break_action]
FINISH
GO [(from)location/**R**(reset)] [(to break number)**#**bn] [(occurrence)**:**count]
NEXT [count] [**LI**(lines)/**IN**(instructions)] [**H**(halt at breakpoints)]
STEP [count] [**CY**(cycles)/**LI**(lines)/**IN**(instructions)] [**H**(halt at breakpoints)]

TRACE [count] [**CY**(cycles)/**LI**(lines)/**IN**(instructions)] [**H**(halt at breakpoints)]
UNTIL addr [**H**(halt at breakpoints)

C Source Code Debug

DOWN [n]
FRAME [#n]
REDIRECT **STDIN OFF**/file
REDIRECT **STDOUT**/**STDERR OFF**/file [**-A**/**-O**/**-C**]
REDIRECT **OFF**
STREAMS [**E**(enable)/**D**(disable)]
TYPE **{**c_expression**}**

GUI Windows

WASM [**OFF**]
WBREAKPOINT [**OFF**]
WCALLS [**OFF**]
WCOMMAND [**OFF**]
WHERE [[+/-]n]
WINPUT [**OFF**]
WLIST [win_num] **OFF**/file
WMEMORY [win_num] space [addr]
WMEMORY [win_num] [**OFF**]
WOUTPUT [**OFF**]
WREGISTER [win_num] [**OFF**]
WSESSION [**OFF**]
WSOURCE [**OFF**]
WSTACK [**OFF**]
WWATCH [win_num] [#wn] [radix] reg|addr|expression
WWATCH [win_num] [#wn] [**OFF**]

Miscellaneous

DEVICE [dev_num[dev_type/**ON**/**OFF**/**X**]]
EVALUATE [**B**(binary)/**D**(decimal)/**F**(float)/**H**(hex)/**U**(unsigned)] expression
HELP [command/reg/topic]
LIST [**+/-/.**/addr]
PATH [pathname]
PATH + [pathname]
PATH -

QUIT [**E**(enable)] [**D**(disable)]
RADIX [**B**(binary)/**D**(dec)/**F**(float)/**H**(hex)/**U**(unsigned)] [reg[_block]/addr[_block]]...
SYSTEM [system_command [parameter_list]]
WAIT [count(seconds)]
UNLOCK dev_type password
VIEW [**A**(assembly)/**S**(source)/**R**(register)]

2.3.1 ASM: Single Line Interactive Assembler

ASM [**B**(byte wide)] [(beginning at)addr] [assembler_mnemonic]

The **asm** command invokes a single-line DSP assembler program allowing the user to create or edit DSP object code programs in memory using assembly language mnemonics. The assembler mnemonic is immediately converted into the proper machine language code and stored in memory. The source line entry is not saved.

The addr parameter is optional. The beginning address can be in any of the memory maps of the DSP. Use the Simulator's "help mem" command to obtain a list of the valid memory space prefixes. If no address is specified the Simulator begins assembly in the **p** (program) memory space using the current program counter value as the beginning address. The **pr** memory designation specifies the special bootstrap ROM area of the DSP.

An interactive mode of the **asm** will be initiated if no assembler mnemonic is specified on the command line. Invoking this mode causes the object code at the beginning address to be disassembled and displayed on the screen. The user may optionally enter a new assembler mnemonic on the command line. Subsequent or previous memory locations can be disassembled by typing, respectively, **Up-Arrow** (Ctrl-U) or **Down-Arrow** (Ctrl-N). The assembler is called when the carriage return key is entered. If the new instruction cannot be assembled correctly an error message is displayed on the error line and the cursor is placed at the point of error. Typing the **ESC** key causes the interactive **asm** command to terminate.

The **b** (byte-wide) parameter takes one byte from each memory word starting at the specified address to build up the instruction word to be displayed. Similarly the assembled mnemonic instruction is divided into bytes and stored in successive words.

2.3.1.1 GUI Interactive Assembler

If the interactive assembler is invoked with the GUI version of the Simulator, a dialog box displays the original instruction at the specified location. To change the instruction and display the next, type the new instruction and click [OK]. To exit the interactive assembler, click [CANCEL]. Any new instruction which has been typed before clicking [CANCEL] will not be written to the current location.

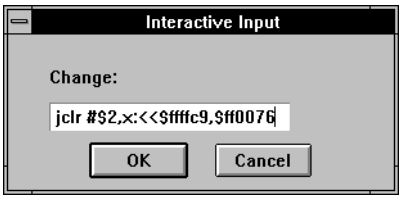


Figure 2-1 Interactive Assembler Dialog Box

The SESSION and COMMAND windows will be written to during interactive assembler operations. Both windows display the original **a**sm command, the SESSION window displays each change as it is applied.

EXAMPLES

asm **p:$50**
Start interactive assembler at program memory address 50 hex.

asm **x:0 move r0,r1**
Assemble single instruction at x memory address 0.

asm
Start assembler at current program counter value.

asm lab_d+5
Start assembler at symbolic address lab_d+5.

asm myfile.asm@7
Start assembler at the address corresponding to myfile.asm line 7

asm **b y:$040100**

Perform byte-wide assembly from address $40100 in y memory. Each byte of the instruction is stored in successive locations, so two or three locations are required to store each 16- or 24-bit instruction. Even if assembled into program memory, this code cannot be executed directly; it is intended for use with code similar to the byte-wide loader in the ROM bootstrap code.

Byte-wide assembly may be used interactively (as in this example) or to assemble a single instruction.

2.3.2 BREAK: Set, Modify, or Clear Breakpoint

BREAK [#bn] [expression] [break_action]
BREAK [#bn] **R**(read)/**W**(write)/**RW**(access) reg/addr[_block] [break_action]
BREAK [#bn[,bn,...]] **OFF**/**E**(enable)/**D**(disable)
BREAK [#bn] **DR**(dma read)/**DW**(write)/**DRW**(access) addr[_block] [break_action]

The **break** command can be used to set, modify, or clear a breakpoint condition and to specify the action that occurs if the breakpoint condition is true. The **break** command has four possible forms as indicated by the four command syntax lines above. The first form causes a break condition if the evaluated expression is non-zero. The second form causes a break condition if a selected register or memory location is accessed by the core. The third form permits a breakpoint, or list of breakpoints, to be selectively enabled, disabled or deleted. The fourth form causes a break condition if a selected memory location is accessed by a DSP dma controller. It is valid only for devices with on-chip dma controllers.

The break_number parameter is optional. The break_number can be specified if the user wishes to assign a specific breakpoint number to a breakpoint definition or wants to modify or delete an existing breakpoint. The break_number should be a positive decimal integer constant in the range 1-99. If the user does not specify a breakpoint number, the Simulator automatically assigns the lowest unused number.

A breakpoint expression can be any logical expression that is valid for the DSP Macro Assembler. The following is a list of operators that may be used in the breakpoint expression:

<	less than
&&	logical "and"
<=	less than or equal to
||	logical "or"
==	equal to
!	logical "negate"
>=	greater than or equal to
&	bitwise "and"
>	greater than
|	bitwise "or"
!=	not equal to
~	bitwise one’s complement
+	addition
^	bitwise "exclusive or"
-	subtraction
<<	shift left
/	division
>>	shift right

See Chapter 5 for more detailed information on expression evaluation.

The breakpoint expression usually involves comparison of register or memory values. Any register name may be used in an expression. There are also two special flag variables that may be referenced in the breakpoint expression:

eof Is TRUE if an end-of-file condition occurs in an input file assigned to a peripheral or memory location.

jump Is TRUE if a "jump" change of flow occurs during code execution.

The Simulator can take various actions when a breakpoint is met during DSP program execution. If no break_action parameter is entered, the default action is to halt program simulation and display all enabled registers and memory blocks. Alternative Simulator actions can be specified by entering one of the following break_action parameters:

H Halt execution. This is the default.
In Increment counter variable **CNT**n (n=1..4).
N Note - display the breakpoint expression and continue.
S Show the enabled register/memory set and continue.
X Execute a Simulator command at breakpoint. Device execution commands, such as trace or go, will not execute.

One other very useful form of breakpoint expression breaks at an address only when the opcode from that memory location is being decoded for next cycle execution. Other forms of the breakpoint expressions which check the value of the pc register or check for a read of a p memory location are less definitive due to the pipelined prefetch of the device. This special form of breakpoint is selected if the breakpoint expression is a single P memory address.

If the ".cld" file contains symbolic debug line number information, breakpoint addresses may be specified using a line_number or filename@line_number designation.

If the ".cld" file contains C symbolic debug information, breakpoint expressions can include any valid C expression for the program.

EXAMPLES

break
Display all currently enabled breakpoints.

break **off**
Remove all breakpoints.

break **#1,3,5..9 off**
Remove breakpoints numbers 1, 3 and 5 through 9.

break **pc>=$500**
Halt DSP program simulation and display enabled registers and memory when the program counter register is greater than or equal to hexadecimal 500.

break **(lc<10)&&(pc>100)**
Halt if the loop counter is less than 10 and the program counter is greater than 100.

break **jump n**
Display breakpoint message if a jump change of flow occurs during execution.

break **eof||pc>$fff**
Halt if an end of file condition occurs in an assigned peripheral input file or if the program counter is greater that hexadecimal FFF.

break **r0==r1**
Halt when the value of register **r0** equals the value of register **r1**.

break **lc>0&&jump i1**
Increment variable **cnt1** if a jump occurs and the loop counter is greater than 0.

break **r r0**
Halt if register **r0** is accessed for a read operation.

break **p:100**
Halt if the execution address is p:100.

break **w lc**
Halt if the loop counter register is written during code execution.

break **10 x evaluate h r0**
Set a breakpoint at the address corresponding to line 10 of the current source file. Execute the Simulator command "evaluate h r0" when the breakpoint occurs.

break **myfile.asm@20**
Set a breakpoint at the address corresponding to line 20 of source file myfile.asm.

break **r xdat..xdat+50**
Halt if a read occurs from one of the 50 addresses beginning at the address associated with the symbol **xdat**.

break **rw p:30..40 s**
Display enabled registers and memory and continue program simulation if any program memory location from decimal 30 to 40 is accessed.

break **#1..10 d**
Disable breakpoints numbers 1 through 10.

break **{j==2}**
Halt if the C expression "j==2" is true.

break **e**
Enable all breakpoints.

2.3.3 CHANGE: Change Register or Memory Value

CHANGE [register[_block]/addr[_block] [expression]]...

The **change** command can be used to change the value of a register, memory location, block of registers, or block of memory. A register block is represented by two register names separated by two periods. For example, **r0..r3** means registers **r0** through **r3**. A memory block can be specified by either start_addr#count or start_addr**..**end_location. For examples: **p:5#20** means 20 locations beginning from program memory location 5; **p:5..20** means program memory locations 5 through 20.

The expression can be a simple constant value or a complex expression with multiple operators and operands. A more extensive discussion of valid expressions is presented in Chapter 5.

Multiple register names, memory locations and expressions can be specified in the same command line. Each specified destination must be followed by the value or expression to be assigned to it.

An interactive mode of register/memory display and change can be initiated by specifying a single register or memory location without an associated expression. In this mode each register or memory location can be examined and optionally modified. Subsequent or previous memory locations or register names can be examined and changed by typing, respectively, **Up-Arrow** (Ctrl-U) or **Down-Arrow** (Ctrl-N). Typing the **ESC** key causes the interactive change command to terminate.

2.3.3.1 GUI Interactive Change Mode

If interactive change mode is entered with the GUI version of the Simulator, a dialog box displays the original value of the specified location, preceded by a semicolon '**;**'. To change the location and display the next, type the new value before the semicolon and click [OK]. The old contents appearing after the semicolon may, but need not, be deleted. To exit interactive change mode, click [CANCEL]. Any new value which has been typed before clicking [CANCEL] will not be written to the current location.

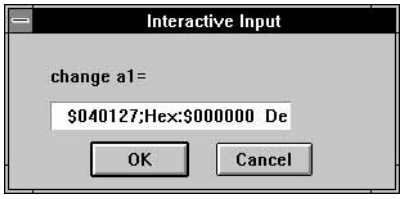


Figure 2-2 Interactive Change Dialog Box

The SESSION and COMMAND windows will be written to during interactive change operations. Both windows display the original **c**hange command, the SESSION window display each change as it is applied.

EXAMPLES

change **pc**
Display register values individually starting with the program counter and prompt the user for new values.

change **xi:$55**
Display internal x memory location hexadecimal 55 and prompt the user for a new value.

change **p:$20 $123456**
Change p memory address hexadecimal 20 to hexadecimal 123456.

change **xdat $234**
Change x memory address corresponding to symbolic label **xdat** to hexadecimal 234.

change **xdat..xdat+5 35**
Change memory block beginning at the address corresponding to symbolic label **xdat** and ending at **xdat**+5 to decimal value 35.

change **r0..r3 0 pi:$30..$300 0 x:$fffe $55 pc 100**
Change registers r0 through r3 to 0, internal p memory addresses hexadecimal 30 through 300 to 0, x memory address hex fffe to hex 55 and the program counter to 100 decimal.

2.3.4 COPY: Copy a Memory Block

COPY (from) addr[_block] (to) addr

The **copy** command copies memory blocks from one location to another. The source and destination memory maps may be different. This allows the user to move data or program code from one memory map to another or to a different address within the same memory map.

EXAMPLES

copy **pi:$100..$500 x:$500**
Copy the internal program memory values located from hexadecimal 100 through hexadecimal 500 to x memory starting at hexadecimal 500.

copy **x:0#100 p:0**
Copy one hundred memory locations beginning at x memory location 0 to p memory beginning at location 0.

copy **lab_1#100 lab_2**
Copy one hundred memory locations beginning at the memory location corresponding to symbolic label **lab_1** to memory beginning at the address corresponding to symbolic label **lab_2**.

copy **xdat..xdat+40 ydat**
Copy 40 memory locations beginning at the address corresponding to symbolic label **xdat** to the block beginning at address corresponding to symbolic label **ydat**.

copy **p:0..20 p:40**
Copy p memory locations 0 through 20 to p memory locations 40 through 60.

2.3.5 DEVICE: Multiple Device Simulation

DEVICE [dev_num [dev_type/**ON**/**OFF**/**X**]]

The **device** command allows the user to create a new device for multiple DSP simulations. It also allows the user to switch to a simulated device for command execution and display, to list the current status of all devices, to disable or enable a device, or to delete a device.

The **dev_num** parameter specifies one of 32 possible devices. The current device number is displayed as the Simulator command line prompt. The number is in the form **DVn**, where the **n** can be a decimal value 0 to 31.

If the **dev_type** parameter is used, it will allocate and initialize a device structure of the specified type to be simulated by device dvn. If the device number is specified, but no device type, subsequent commands and display will reference the new device number. If the device does not exist, it will be created with a default device type and made active. Non-disclosed devices must be unlocked with the Simulator **unlock** command prior to use with the **device** command.

The **ON** parameter makes the specified device active during commands which cause device execution (**go**, **step** or **trace**). During execution cycles, each active device executes a single clock cycle in turn. Device to device pin interconnections specified by the **input** command are updated following each cycle for active devices.

The **OFF** parameter makes the specified device inactive. It does not otherwise change the state of the selected device.

The **X** parameter discards the device structures allocated for a device. If you specify this command for the currently displayed device, the Simulator will switch to another device before discarding the structures. The Simulator needs at least one allocated device in order to have the required structures for the display window, so deletion of the last device is not allowed.

EXAMPLES

device
Display a list of all devices and their current status. Also display the list of possible device types.

device **dv9**
Switch to device **dv9**. If it doesn't exist, create a device **dv9** with the default device type.

device **dv1 on**
Enable device **dv1** cycle execution. If it doesn't exist, create it with the default type.

device **dv0 56116**
Create device **dv0** and initialize it for device type **56116**.

2.3.6 DISASSEMBLE: Object Code Disassembler

DISASSEMBLE [**B**(byte wide)] [addr[_block]]

The **disassemble** command allows the user to review DSP object code in its assembly language mnemonic format. Invalid opcodes are disassembled to a define constant (DC) mnemonic.

The **b** (byte-wide) parameter constructs the instruction words by taking one byte from each word of memory, starting from the specified address.

EXAMPLES

disassemble
Disassemble the next 20 instructions beginning with instruction pointed to by the program counter. Repeatedly entering this command will result in consecutive 20 instruction blocks being disassembled.

disassemble **pr:0..20**
Disassemble program bootstrap rom memory address block 0 to 20.

disassemble **lab_1..lab_2**
Disassemble memory address block beginning at the address corresponding to symbolic label **lab_1** and ending at **lab_2**.

disassemble **xdat#20**
Disassemble 10 instructions beginning at the address corresponding to symbolic label **xdat**.

disassemble **7**
Disassemble instructions beginning at the address corresponding to line 7 in the current source file.

disassemble **test.asm@8**
Disassemble instructions beginning at the address corresponding to line 8 in the source file **test.asm**.

disassemble **x:$50#10**
Disassemble 10 instructions starting at x memory map hex 50.

disassemble **b y:$1000#$40**
Disassemble 40 instructions starting at address y:$1000. The instruction words are constructed by taking one byte from each location; thus depending on the target processor, two or three locations are required to hold each instruction word.

2.3.7 DISPLAY: Display Register or Memory

DISPLAY [**ON**/**OFF**/**R**/**W**/**RW**] [reg[_block/_group]/addr[_block]]...
DISPLAY **V**(version)

The **display** command allows the user to examine the contents of a register group and or memory block. It can also be used to enable, conditionally enable, or disable particular registers or memory locations for automatic display when executing debug commands, or to display the memory configuration or Simulator version number. The display radix for each register and memory location can be individually specified using the **radix** command. The default display radix is hexadecimal.

Entering the **display** command with the single parameter **v** will initiate display of the Simulator version number. The Simulator version number display shows the revision number and date of the Simulator.

EXAMPLES

display **v**
Display Simulator version number and date of release.

Entering the **display** command with no parameters will cause the display of all enabled registers and memory blocks. Registers and memory blocks can be enabled or disabled by entering the command with one of the "enable" keywords - **ON**, **OFF**, **R**, **W**, or **RW** - prior to the register and memory list. The enable keywords have the following meaning:

ON Always display the following registers and memory locations.

OFF Never display the following registers and memory locations.

R Display the following registers and memory locations if they were accessed for a read operation since the last display occurred.

W Display the following register and memory locations if they were accessed for a write operation since the last display occurred.

RW Display the following register and memory locations if they were accessed for read or write operations since the last display occurred.

The R, W, and RW functions cause accumulation of a list of accesses from display to display. All accesses to register locations can be saved. The memory lists only store a maximum of 16 memory accesses (for each memory space). If more than 16 locations were accessed since the previous display, only the last 16 will be stored. Register and memory locations that have been accessed for a write operation are shown highlighted on the display.

EXAMPLES

display **on**
Enable all registers for display

display **on pi:0..20 xi:30..40**
Display enable internal p memory address block 0 to 20 and internal x memory address block 30 to 40.

display **off**
display **on core ssi0**
Disable all display, then enable display of the DSP core registers and the SSI0 peripheral registers.

display **w r0..r3 x:0..100**
Display conditionally (if they are written) registers **r0** through **r3** and x memory locations 0 through 100.

Entering the **display** command with a register or memory list, but without one of the "enable" keywords, will cause immediate display of the listed registers and memory locations without affecting their "enable" status.

The peripheral names can be used in the display list to enable or display all the registers associated with that peripheral. The valid peripheral names for the selected device can be obtained by using the Simulator's "help periph" command. The name **all** can be used to enable or display all registers of the selected device.

EXAMPLES

display
Display all currently enabled registers and memory.

display **p:0..300**
Immediate display of p memory addresses 0 through 300.

display **test.asm@7**
Immediate display of memory location corresponding to line 7 of source file **test.asm**.

display **xdat**
Immediate display of memory location corresponding to symbolic label **xdat**.

display **all**
Immediate display of all registers plus the enabled memory locations.

2.3.8 DOWN: Move Down the C Function Call Stack

DOWN [n]

The **down** command is used to move down the call stack. It can be used in conjunction with the **where**, **frame**, and **up** commands to display and traverse the C function call stack.

After entering a new call stack frame using down, that call stack frame becomes the current scope for evaluation. In other words, for C expressions, the evaluate command acts as though this new frame is the proper place to start looking for variables.

EXAMPLES

down
Move down the call stack by one stack frame.

down 2
Move down the call stack by two stack frames.

2.3.9 EVALUATE: Evaluate an Expression

EVALUATE [**B**(binary)/**D**(dec)/**F**(float)/**H**(hex)/**U**(unsigned)] expression/**{**c_expression**}**

The **evaluate** command is used as a calculator for evaluating arithmetic expressions or for converting values from one radix to another. The result of the expression evaluation is displayed in the specified radix. If a radix is not specified in the **evaluate** command line, the current default radix (specified by the **radix** command) will be used.

An expression consists of an arithmetic combination of operators and operands. An operand can be a register name, a memory location, or a constant value.

The order of evaluation of an expression's operators will be associated from left to right. Parentheses can be used to force the order of evaluation of the expression. A more extensive discussion of the expressions which are valid for the evaluate command is presented in Chapter 5.

When values held in the DSP's registers or memory spaces are used in an expression that involves a multiply operator, the display radix (specified by the **radix** command) will determine whether the operation executed is a floating point or integer multiply.

EXAMPLES

evaluate **r0+p:$50**
Add the value in r0 register to the value in program memory address hexadecimal 50 and display the result using the default radix.

evaluate **b $345**
Convert hexadecimal 345 to binary and display the result.

evaluate **lab_d**
Display the address of the location associated with symbolic label **lab_d**.

evaluate **{count}**
Display the value of the C variable **count**.

evaluate **h %10101010&p:r0**
Calculate the bitwise **AND** of the program memory address specified by the value in r0 register and the binary value 10101010 and display the result in hexadecimal.

2.3.10 FINISH: Step Until End of Current Subroutine

FINISH

The **finish** command executes instructions until a return-from-subroutine (RTS) instruction is executed within the current subroutine. The Simulator simply steps, checking if any instruction is a RTS. If so, that RTS is executed, and instruction execution halts immediately afterward. While stepping, if a branch to subroutine or jump to subroutine instruction is encountered, tests for the RTS instruction are suspended until execution resumes at the address following the subroutine call.

EXAMPLES

finish
Finish the current subroutine, continuing from the current address until an RTS is executed.

2.3.11 FRAME: Select C Function Call Stack Frame

FRAME [#n]

The **frame** command is used to select the current call stack frame. It can be used in conjunction with the **where**, **down**, and **up** commands to display and traverse the C function call stack.

After entering a new call stack frame using **frame**, that call stack frame becomes the current scope for evaluation.

EXAMPLES

frame **#2**
Select call stack frame number two.

frame **#0**
Select call stack frame number zero (innermost frame).

The frame command executes instructions until a return-from-subroutine (RTS) instruction is executed within the current subroutine. The Simulator simply steps, checking if any instruction is a RTS. If so, that RTS is executed, and instruction execution halts immediately afterward. While stepping, if a branch to subroutine or jump to subroutine instruction is encountered, tests for the RTS instruction are suspended until execution resumes at the address following the subroutine call.

2.3.12 GO: Execute DSP Program

GO [(from)location/**R**(reset)] [(to break number)**#**bn] [(occurrence)**:**count]

The **go** command initiates simulated execution of DSP code. The Simulator fetches, decodes, and executes instructions in the exact manner as the processor. The go command will pass control to the Simulator until a breakpoint is reached, a control-c character is entered on the keyboard, or an illegal instruction is encountered.

Invoking the command with no parameters will start simulation from the current program counter value. If an address or reset parameter is included, the instruction pipeline, instruction counter, and cycle counter will be cleared before program simulation. The reset (**R**) parameter causes a simulation of the reset sequence in the processor. The device registers are reset and execution begins at the reset exception address.

The optional #bn parameter may be used to cause the code execution to halt only if that particular breakpoint condition occurs. All other breakpoint conditions are ignored.

The optional :count parameter may be used to cause the code execution to halt only if the breakpoint has occurred a specified number of times. If #bn is not specified, then simulation will stop if **count** number of breakpoint conditions have occurred.

Ctrl-C will always abort the Simulator **go** command, even if specified breakpoint conditions have not occurred.

EXAMPLES

go
Start program simulation from the current instruction. Stop at the first occurrence of any breakpoint.

go $100
Start program simulation at program memory address hex 100 after clearing the instruction pipeline. Stop at the first occurrence of any breakpoint.

go **r**
Clear the Simulator pipeline and start program simulation at the reset vector. The simulated machine state is also reset according to the processor reset sequence.

go #5
Continue execution from the current instruction. Halt on the first occurrence of breakpoint number 5.

go #5 :3
Continue execution from the current instruction. Halt on the third occurrence of breakpoint number 5.

go **lab_d** #5 :3
Start from symbolic address **lab_d** after clearing the Simulator pipeline. Halt on the third occurrence of breakpoint number 5.

2.3.13 HELP: Simulator Help Text

HELP [command/reg/topic]

The **help** command provides syntax and examples of Simulator commands, descriptions of device register bit fields, and help on other topics related to device or Simulator operation.

If no keyword is entered the Simulator displays a summary of the possible help topics. If the keyword is a command name the Simulator displays a summary of that command's parameters along with a brief description and examples. If the keyword is a register name the Simulator displays the specified register's contents along with the help text associated with the register.

The topic keywords below provide on-line help for the described topics:

io : list of on-chip io registers and their addresses

int : list of interrupt vector addresses for the device

periph: list of peripheral names

pin : list of pin names and numbers, and the current pin states

port : list of port names

mode : initial chip operating mode summary

map : memory map descriptions for various omr settings

mem : memory names with block addresses

sym : display program symbol table names and values

reg : display register size, register and peripheral index

stack : display of values on the device stack

EXAMPLES

help
Display a summary of all available commands and their parameters.

help **asm**
Display a summary of the **assemble** command and its parameters.

help **omr**
Display the contents of the DSP's Operating Mode Register.

2.3.14 HISTORY: Disassemble Previously Executed Instruction

HISTORY

The **history** command disassembles and displays the previous 32 instructions executed by the device. The instructions are displayed in the order that they were executed, with the most recent instruction appearing at the bottom of the list. The last instruction in the list has been fetched and decoded by the device and will enter the execute phase in the next device cycle. It is in the same state as instructions that are disassembled and displayed at the end of each trace display.

A typical use for this command would be to determine the sequence of instructions that terminated in a user-defined breakpoint. The user would set the breakpoint condition using the **break** command, then issue the **go** command. When the break condition is met, instruction execution halts and the currently enabled registers are displayed. The user can then issue the **history** command to view the last 32 instructions that executed prior to the breakpoint.

The device execution history can also be logged continuously to an output file using the Simulator **output** command. See the documentation of the **output history** form of the **output** command.

2.3.15 INPUT: Assign Input File

INPUT [#n] [**T**(timed)] addr/port/periph/pin[_group] **OFF/TERM**/file [ioradix]
INPUT [#n] pin (from)[dev_num**:**]pin
INPUT [#n] addr (from)[dev_num**:**]addr

The **input** command retrieves data for a peripheral, memory location, or device pin from the specified source. The valid peripheral and port names for the selected device can be obtained by using the Simulator's "help periph" and "help port" commands.

The input source may be a disk file or the user's terminal. Use of the keyword **TERM** assigns input from the terminal. The source may provide data only or time-data pairs. Use of the keycharacter **T** specifies the time-data pair format. The input data is in ascii. The data value may be expressed in hexadecimal, decimal, or floating point for memory and peripheral name assignments, as one of 5 input values (0,1,n,p or X) for assignment to individual digital pins, or as single precision floating point values for analog input pins.

The data default input radix may be specified by using **-RD**,**-RF**,**-RH** or **-RU** following the filename. Hexadecimal input is the default for addr, port and periph data values. Input analog pin data files must be assigned with the **-RF** radix designator, and the file data must be single precision floating point values. The time value is always expressed in decimal. Chapter 3 contains an extensive description of the input file format.

A special feature, which uses the second form of the command, allows input to a device pin from another device pin without having to store the data in a disk file. The source pin may optionally be preceded by a device number to allow pin to pin connections during multiple device simulations.

Assignment to a memory address causes all subsequent reads of that memory address to reference the input source. This method may be used to simulate the user's unique memory mapped peripherals or to short-circuit the simulation of the on-chip peripherals.

The third form of the command causes the Simulator to read the memory location of the specified source device (specified by dvn:addr) each time the destination memory address is accessed for a read. This enables simulation of interconnection of multiple devices via dual-port memory. The source device must exist (create it with the Simulator **device** command) prior to issuing this form of the input command.

If a filename suffix is not specified, the Simulator will assume ".io" for a non-timed input file and ".tio" for a timed input file.

EXAMPLES

input **xe:$800 xfile -rd**
Get values for external memory location x:800 from input file "xfile.io". The data values are stored in decimal form in the input file.

input **ssi0 hfile**
Get values for the SSI0 peripheral from input file "hfile.io".

input **d15..d0 dfile**
Get values for pins **D15** through **D0** from input file "dfile.io".

input **d15..d0 dfile**
Get values for pins **D15** through **D0** from input file "dfile.io".

input **irqb dv1:pb0**
Input values for the current device’s irqb pin from device dv1’s pb0 pin.

input **t irqa term**
Input time and data pairs from the terminal for the device IRQA pin.

input **x:500 dv5:x:3000**
Input data for memory reads of x:500 of the current device from device number 5 address x:3000.

input **#2 x:$800 xfile -rd**
Get values for external memory location x:800 from input file "xfile.io". The data values are stored in decimal form in the input file. Input assignment number 2 is explicitly replaced due to the #2 in this command form.

input **#2 off**
Input assignment number 2 is explicitly deleted by index number.

input **mic micfile -rf**
Input untimed analog pin data for the mic analog pin from the file "micfile.io".

2.3.16 LIST: List Source File Lines

LIST [**+/-/.**/addr]

The **list** command displays source lines or disassembled instructions from the specified source file, or beginning at the specified address.

The current display mode determines whether a source file or assembly mnemonics will be displayed. If the Simulator is in the register display mode, this command will switch it to the source display mode and display the source file lines associated with the specified address or line number. If the display mode is already source or assembly, the display mode is not altered. The assembly display mode displays disassembled instructions corresponding to the specified address or line number.

The next or previous pages of the currently displayed source file may be selected by specifying **+** or **-** rather than a specific address or line number. In addition, the source or assembly associated with the current execution address may be selected by specifying**.** (period) or by using the list command without a parameter.

EXAMPLES

list **20**
List source or assembly corresponding to line 20 of the current source file.

list **test.asm@20**
List source or assembly corresponding to line 20 of the source file **test.asm**.

list **test.asm**
List source or assembly corresponding to line 1 of the source file **test.asm**.

list **+**
Display the next page of the current source file or assembly.

list **.**
Display source or assembly corresponding to the current execution address.

list **-**
Display the previous page of the current source file or assembly.

list **test.asm**
List source or assembly corresponding to line 1 of the source file **test.asm**.

list **lab_1**
List source or assembly corresponding to symbolic address **lab_1**.

2.3.17 LOAD: Load DSP Files or Configuration

LOAD [**S**(state)|**M**(memory-only)|**D**(debug symbols-only)] (from) file

The **load** command can be used to load DSP object module format (.lod) files or DSP COFF (.cld) files into the Simulator memory or to load a previously saved simulation state file.

If only a **file** parameter is specified, then the Simulator assumes that the file is an object file. The object file may be in either the special ASCII OMF format described in Chapter 3, or in the DSP COFF format generated by the DSP Macro-Assembler. The OMF format file can be created using the Simulator **save** command or with a text editor. A directory path may be specified with the filename. If no filename suffix is specified, the Simulator will search first for a OMF format ".lod" file, then for a COFF format ".cld" file. Loading a COFF format file replaces the Simulator's symbolic debug information unless the **M** option, described in the examples below, is specified.

If the **S** keycharacter is specified, the Simulator will load **filename** as a Simulator state file. The Simulator state file can be created using the Simulator **save s** command. Loading the Simulator state changes the entire setup of the Simulator to the previous definition saved in the state file. If no filename suffix is specified, ".sim" is assumed.

If the **M** keycharacter is specified, the Simulator will load object file **filename**, .cld or .lod, without modifying the Simulator' s symbolic debug information.

If the **D** keycharacter is specified, the Simulator will load only the symbolic debug information from the object file **filename**. The device memory contents are not altered. Only the COFF format files (.cld suffix) are supported by this option.

EXAMPLES

load **\source\testloop.obj**
Load the OMF format "testloop.obj" file from directory "source".

load **\source\testloop.cld**
Load the COFF format "testloop.cld" file from directory "source", including the memory contents and any symbolic debug information contained in the file.

load **lasttest**
Load the OMF format "lasttest.lod" file from current directory.

load **d test.cld**
Load the symbolic debug information from the COFF format "test.cld" file, ignoring the memory contents of the file.

load **m test.cld**
Load the COFF format "test.cld" file, ignoring any symbolic debug information in it.

load **s lunchbrk**
Load "lunchbrk.sim", replacing the entire current Simulator state.

2.3.18 LOG: Log Commands, Session, Profile

LOG [**OFF**] [**C**(commands)/**S**(session)/**P**(profile) [file [**-A**/**-O**/**-C**]]]
LOG [**OFF**] **V**(source display status line)

The **log** command allows the user to record command entries only, to record all session display output, or to record an analysis of DSP program execution. Recording of commands only is useful as a method of generating macro command files. Recording all session display output provides a convenient way for the user to review the results of an extended sequence of commands. Since the output log files are in ascii format, they can easily be printed or reviewed using an editor program. Recording a program profile assists in the analysis of program structure and execution characteristics.

Entering the **log** command with no parameters will cause the Simulator to display the currently opened log filenames. The keyword **OFF** is used to terminate logging. The **C** and **S** keycharacters are used to specify whether the logfile will contain only commands (**C**), or all session output (**S**). A keycharacter **-A**, **-O**, or **-C** may be specified to select append, overwrite, or cancel if the filename already exists. As a default, the user will be prompted during command execution. The **V** keycharacter enables logging of the source display status line to a session log file. It is primarily intended for testing the Simulator display.

The **P** keycharacter specifies that a program execution profile is to be created. The program to be profiled must be loaded before issuing the '**log p**' command. Both memory and symbols must be loaded. Information is gathered as program execution is simulated, and the profile output files are written when the profiling is terminated with the command '**log off p**'.

The suffixes ".cmd" and ".log" are added, respectively, to the commands-only or session filename if no other suffix is specified. For profile logging, two suffixes are used, a ".log" file which contains plain text which may be printed on any 80-column printer, and a ".ps" file which is formatted for a postscript printer.

In multiple device simulations, there is a separate session log file associated with each simulated device, but there is only a single command log associated with the entire multiple device simulation. If a device type is changed using the **device** command, the user interface information associated with the discarded device type, including the session log file name, is cleared; so it is best to specify the log s command following the device command for a particular device.

EXAMPLES

log
Display currently opened log files.

log **s \debugger\session1**
Log all display entries to filename "session1.log" in directory "\debugger"

log **c macro1 -a**
Log all commands to filename "macro1.cmd". Append if it already exists.

log **off c**
Terminate command logging.

log **off**
Terminate all logging.

log **v**
log **s session1**
Log source display status line and all display entries to "session1.log"

load px41v17.cld
log **p** px41v17 **-o**

Load memory and symbols for program px41v17 and log the program profile in files px41v17.log and px41v17.ps. Overwrite these files without warning if they already exist. Note that the program(s) to be profiled must have been loaded before this **lo**g command is issued.

2.3.19 MORE- Enable/Disable session paging control.

MORE [**OFF**]

The **more** command allows the user to enable or disable the paging of data on the session window. This is particularly useful when displaying large amounts of data and you wish to examine the data page by page.

The paging feature is turned off by default and data will scroll vertically across the screen when it is larger than the size of the screen.

EXAMPLES

more

Turn on session display paging control.

more **off**

Disable session display paging control (reset or default state).

2.3.20 NEXT: Step Over Subroutine Calls or Macros

NEXT [count] [**LI**(lines)/**IN**(instructions)] [**H**(halt at breakpoints)]

The **next** command functions the same as the STEP command, except that if the next instruction to be executed calls a subroutine or begins execution of a macro, all the instructions of the subroutine or macro are executed before stopping to display the enabled registers. In order to recognize macros, the symbolic debug information for the program code must be loaded. The debug information is included in the COFF format .cld files generated using the assembler's -g option.

The optional **count** value enables repeating of the **next** command the specified number of times before execution terminates.

As the default, all breakpoints are ignored while the next command is executing. The **h** option enables halting at breakpoints.

As the default, the command executes the next instruction if viewing the assembly or register screens, and the next line if viewing the source screen. The **li** and **in** options permit source line or instruction increments to be specified explicitly.

EXAMPLES

next
Step over subroutine calls or macros; or otherwise just advance one instruction or source line, depending on the display mode, and display the enabled registers and memory blocks.

next **li**
Step over subroutine calls or macros; or otherwise just advance one source line and display the enabled registers and memory blocks.

next **in**
Step over subroutine calls or macros; or otherwise just advance one assembly instruction and display the enabled registers and memory blocks.

next **10**
Execute the equivalent of 10 **next** instructions, halting to display the enabled registers and memory blocks only after the tenth invocation.

next **10 li**
Execute the equivalent of 10 **next li** instructions, halting to display the enabled registers and memory blocks only after the tenth invocation.

next **10 h**
Execute the equivalent of 10 **next** instructions, halting to display the enabled registers and memory blocks after the tenth invocation, or if any breakpoint is encountered.

2.3.21 OUTPUT: Assign Output File

OUTPUT [#n] [**T**] addr/port/periph/pin[_group] **TERM**/file [ioradix/**-RS**] [**-A**/**-O**/**-C**]
OUTPUT [#n] [**T**] addr/port/periph/pin[_group] **OFF**
OUTPUT [#n] [**T**] **history TERM**/file/**OFF** [**-A**/**-O**/**-C**]
OUTPUT [#n] [**T**] **ehistory TERM**/file/**OFF** [**-A**/**-O**/**-C**]

The **output** command stores data from a peripheral, memory location, or device pin to the specified destination. The valid peripheral and port names for the selected device can be obtained by using the Simulator's "help periph" and "help port" commands.

The output destination may be a disk file or the user's terminal. Use of the keyword **TERM** assigns output to the terminal. The Simulator can store data only or time-data pairs. Use of the keycharacter **T** specifies the time-data pair format. The output data is in ascii. The data value may be expressed in hexadecimal, decimal, or floating point for memory and peripheral name assignments, in pin data form (0,1,H,L,n,p,X) for assignment to individual digital pins, or as single precision floating point for assignment to individual analog pins with the **-RF** radix designator.

The output radix for the data value is specified using **-RD**, **-RF**,**-RH**,**-RU** or **-RS** following the filename. The **-RF** radix, when specified for a single output pin, will output the analog single precision floating point value associated with the pins analog function. The **-RS** radix is valid only for output memory locations. It interprets values written to the specified memory location to be the address of a null terminated character string in the same memory space. The character string will be displayed or written to an output file. This string radix is provided primarily for use when debugging programs created with the C Compiler.

The output time value is always expressed in decimal. Chapter 3 contains a thorough description of the output file format. A keycharacter **-A**, **-O**, or **-C** may be specified to select append, overwrite, or cancel if the filename already exists. As a default, the user will be prompted during command execution.

Assignment to a memory address causes all subsequent writes of that memory address to store data in the output file.

If a filename suffix is not specified, the Simulator will attach ".io" to a non-timed output file and ".tio" to a timed output file.

The third form of the output command creates a continuous log of the device execution addresses and disassembled opcodes. The output format is similar to the output generated by the Simulator history command.

The fourth form of the command, which specifies **ehistory**, is an extended version of the **output history** command. Additional execution history information is logged to the output file, including device wait state cycles and bus arbitration cycles and indication of other stall conditions. The extra information is preceded by double asterisks in the log file. Only one of **output history** or **output ehistory** may be active.

EXAMPLES

output **x:$0 xfile -rd**
Store values written to memory location x:0 in output file "xfile.io". The data values will be stored in decimal.

output **ssi0 ssi0file -a**
Store values from the SSI0 peripheral to output file "ssi0file.io".
Append to the file if it already exists.

output **a15..a0 afile**
Store output values for address pins **a15** through **a0** to output file "afile.io".

output **#2 t bg term**
Output time and data pairs from the device **BG** pin to the terminal. Output assignment number 2 is explicitly replaced by this command.

output **#2 off**
Output assignment number 2 is explicitly turned off by index number reference.

output **spkp spfile -rf**
Output untimed single precision floating point values for the analog pin **spkp** to the output file "spfile.io".

output **xdat1 xfile**
Store values written to memory location associated with the symbolic label **xdat1** to the output file "xfile.io". The data values will be stored in the default hexadecimal radix.

output **history hisfile**
Store device execution history to output file "hisfile.io".

output **ehistory hisfile**
Store extended device execution history to output file "hisfile.io".

2.3.22 PATH: Specify Default Pathname

PATH [pathname]
PATH **+** pathname[,pathname,...]
PATH **-**

The **path** command defines the default pathname for storage of Simulator temporary files, log files, macro command files, object files, and peripheral I/O files. If no pathname is specified in the command line, the current default pathname is displayed. The user may still override the default path by explicitly specifying a pathname as a prefix to the filename in any of the commands which reference a file.

Alternate source pathnames may be specified using the "path +" form of the command. Each time the command is issued, the specified pathname, or comma-separated list of pathnames, is added the current list. When searching for files, the Simulator will search first using the default pathname specified for the current device, then in each of the alternate source pathnames, in the order that they were specified.

The third form of the command, "path -", deletes the entire list of alternate source pathnames.

EXAMPLES

path **\sim**
Define the default working directory for Simulator files as "\sim".

path **\sim\day2**
Define the default working directory for Simulator files as "\sim\day2".

path + ..\src
Add pathname "..\src" to the list of alternate source pathnames.

path + ..\src,..\src2
Add pathnames "..\src" and "..\src2" to the list of alternate source pathnames.

path -
Clear the list of alternate source pathnames.

path
Show the default working directory and help file directory for the current device, and the list of alternate source pathnames.

2.3.23 QUIT: Quit Simulator Session

QUIT [**E**(enable)/**D**(disable)]

The **quit** command passes control back to the operating system after closing all log files, input and output files, and macro files.

quit **e**nable and **q**uit **d**isable control the action taken by the Simulator if an error occurs during the execution of a macro command. **q**uit **e**nable specifies that the macro command is aborted and the Simulator quits immediately with a non-zero exit status. **q**uit **d**isable specifies that the Simulator does not exit.

EXAMPLES

quit

Close all currently open files and return to the Operating System.

quit **e**

Specify that errors in a macro command will cause the Simulator to exit with a non-zero status. The Simulator does not exit when this command is issued.

2.3.24 RADIX: Change Input or Display Radix

RADIX [**B**(binary)/**D**(dec)/**F**(float)/**H**(hex)/**U**(unsigned)] [reg[_block]/addr[_block]]...

The **radix** command allows the user to change the default number base for command entry or for display of registers and memory locations. Hexadecimal constants may always be specified by preceding the constant by a dollar sign (**$**). Likewise, a decimal value may be specified by preceding the constant with a grave accent (**‘**), and a binary value may be specified by preceding the constant with a percent sign (**%**). The Simulator, by default, uses decimal input radix and hexadecimal display radix when it is initially invoked. This means that decimal constants may be entered without typing a preceding grave accent. Changing the default input radix allows the user to enter constants in the chosen radix without typing the radix specifiers before each constant.

Specifying a list of register and/or memory locations following the radix specifier will set the display radix of the registers and memory. This does not affect the default input radix.

EXAMPLES

radix
Display the default input radix currently enabled.

radix **h**
Change default input radix to hexadecimal. Hexadecimal constant entries no longer require a preceding dollar sign, but any decimal constants will require a preceding grave accent.

radix **f x:0..10 x0 y0 a b**
Change the display radix for the specified registers and memory blocks to floating point.

2.3.25 REDIRECT: Redirect stdin/stdout/stderr for C Programs

REDIRECT **STDIN OFF**/file
REDIRECT **STDOUT/STDERR OFF**/file [**-A**/-**O**/**-C**]
REDIRECT [**OFF**]

The **redirect** command is used to redirect the stdin/stdout/stderr for C programs. It allows the user to redirect stdin from a file, and redirect stdout/stderr to files.

EXAMPLES

redirect
Display the redirect list, which shows each of the three streams that can be redirected, along with where they are being redirected to.

redirect **stdin input**
Redirect the C stdin (standard input) stream from the file input.cio (.cio is the default extension).

redirect **stdout output.txt**
Redirect the C stdout (standard output) stream to the file output.txt.

redirect **stderr errors**
Redirect the C stderr (standard error) stream to the file errors.cio.

redirect **stdout output -o**
Redirect the C stdout stream to the file output.cio, overwriting the file if it already exists.

redirect **stdout output -a**
Redirect the C stdout stream to the file output.cio, appending to the end of the file if it already exists.

redirect **stdout output -c**
Redirect the C stdout stream to the file output.cio, but don’t redirect if the file already exists.

NOTE: No I/O processing or handling of redirection occurs if the **streams** option has been disabled. See the help page for **streams** for more information.

2.3.26 RESET: Reset Device or State

RESET **S**(state)/**D**(device) [mode]

The **reset** command can be used to reset the device registers (**D**) or the entire Simulator state (**S**). It can also be used to select the operating mode that the device will be set to in response to a simulated hardware reset sequence. The **mode** parameter specifies the DSP operating mode in the form Mn (n=decimal digit). See Chapter 8, Operating Modes for a list of the device dependent valid operating modes.

EXAMPLES

reset **d**
Reset all device registers to the defined reset conditions.

reset **d m0**
Reset the device registers and select operating mode 0 as the default operating mode following subsequent hardware reset sequences.

reset **s**
Reset the entire Simulator state to the start-up condition. All breakpoints are cleared, the memory is initialized, and all logging and I/O files are closed.

2.3.27 SAVE: Save Simulator File

SAVE **S**(state)/addr[_block]... filename [**-A**/**-O**/**-C**]

The **save** command allows creation of a Simulator state file from the current Simulator state, or creation of an object module format file or a COFF format file from specified memory blocks.

If **S** is specified as the second parameter, a Simulator state file is created. It contains the entire simulation state, including memory contents, breakpoint settings, and the current pointer position of any open files. This file is in an internal format that is efficient for the Simulator to store and load (see the **load s** command description). The default suffix for a Simulator state filename is ".sim".

If memory blocks are specified (instead of **S**), the specified memory areas are stored in object format so the file can be reloaded with the Simulator **load** command. The default object format is the OMF format described in Chapter 6. The suffix for an OMF file ".lod", will be appended to the filename if no suffix is explicitly specified. If the COFF file suffix, ".cld", is specified explicitly in the filename, the memory contents will be stored in the DSP COFF object file format. The Simulator does not store symbolic debug information in the output COFF object file.

A keycharacter **-A**, **-O**, or **-C** may be specified to select append, overwrite, or cancel if the filename already exists. As a default, the user will be prompted during command execution. Appending is not a valid option for state files.

EXAMPLES

save **p:0..$ff x:0..$20 session1 -a**
Save all three memory maps to OMF file "session1.lod".
If the file already exists, append to it.

save **s lunchbrk**
Save the Simulator state to filename "lunchbrk.sim".

save **s lunchbrk.b -c**
Save the Simulator state to filename "lunchbrk.b".
If the file already exists, cancel this command.

2.3.28 STEP: Step Through DSP Program

STEP [count] [**CY**(cycles)/**LI**(lines)/**IN**(instructions)] [**H**(halt at breakpoints)]

The **step** command allows the user to execute **count** instructions or clock cycles before displaying the enabled registers and memory blocks. This command gives the user a quick way to specify execution of a number of instructions without having to set a breakpoint. It is similar to the trace command except that display occurs only after the count number of cycles or instructions have occurred.

As the default, all breakpoints are ignored while the step command is executing. The **h** option enables halting at breakpoints.

As the default, the command steps in instruction increments if viewing the assembly or register screens, and in source line increments if viewing the source screen. The **li** and **in** options permit source line or instruction increments to be specified explicitly.

EXAMPLES

step
Step one instruction or source line, depending on the display mode, and display the enabled registers and memory blocks.

step **li**
Step one source line, regardless of the display mode, and display the enabled registers and memory blocks.

step **$50**
Execute hex 50 instructions or source lines, depending on the display mode, then stop and display the enabled registers and memory blocks at the end of the hex 50th instruction.

step **$50 in h**
Execute hex 50 instructions, regardless of the display mode, then stop and display the enabled registers and memory blocks at the end of the hex 50th instruction. Halt if a breakpoint is encountered during the execution.

step **20 cy**
Execute 20 clock cycles and display the enabled registers and memory blocks at the end of the 20th clock cycle.

2.3.29 STREAMS: Enable/Disable Handling of I/O for C Programs

STREAMS [**E**NABLE/**D**ISABLE]

The **streams** command is used to enable and disable the handling of input and output on the host side for C programs. By default, it is enabled. When enabled all input and output that is done in the C program running on the DSP is handled on the host side. So for example, when an fopen() call is made in the C program running on the DSP call, the host software intercepts the call and does the fopen() on the host side.

EXAMPLES

streams **e**
Enable handling of C input/output. All input/output calls done in a C program running on the DSP will be handled by the host software (e.g. fopen(), fwrite(), printf(), etc.).

streams **d**
Disable handling of C input/output.

2.3.30 SYSTEM: Execute System Command

SYSTEM [**-C**(continue immediately)] [system_command [parameter_list]]

The **system** command allows the user to execute an operating system command in two modes. If a system_command and optional parameter list are included in the command line, the specified command is executed. The Simulator is re-entered immediately after execution of the system command. If the command line does not contain a system_command, then a mode is entered in which multiple system commands may be entered. Return to the Simulator occurs when the user enters **EXIT** on the operating system command line.

Operating System commands invoked from within the Simulator will not be logged to the screen buffer for review.

When a system command is specified on the **sy**stem command line, the user is prompted to “Hit return to continue...” before control returns to the Simulator. This allows the user to inspect the command output before it is destroyed.

The command argument “**-C**” (continue immediately) causes control to return to the Simulator without prompting the user. This may be useful in macro commands, allowing system commands to be used without requiring operator intervention.

EXAMPLES

system **dir**
Execute the system "dir" command and immediately return to the Simulator.

system **dir *.io**
Execute the system "dir *.io" command

system
dir *.io
del he.io
exit
Leave the Simulator temporarily. Execute the system "dir *.io" and "del he.io" commands. Return to the Simulator when the system "exit" command is executed.

system **-c** del e:\temp*.lod
Delete the specified temporary files and continue without issuing the continuation prompt.

2.3.31 TRACE: Trace Through DSP Program

TRACE [count] [**CY**(cycles)/**LI**(lines)/**IN**(instructions)] [**H**(halt at breakpoints)]

The **trace** command gives a snap shot of the enabled registers and memory after each instruction or clock cycle during program simulation. Execution terminates after **count** number of cycles or instructions. The h parameter causes tracing to halt at breakpoints; the default operation ignores breakpoints while tracing.

As the default, all breakpoints are ignored while the trace command is executing. The **h** option enables halting at breakpoints.

As the default, the command traces in instruction increments if viewing the assembly or register screens, and in source line increments if viewing the source screen. The **li** and **in** options permit source line or instruction increments to be specified explicitly.

EXAMPLES

trace
Execute one instruction or source line, depending on the display mode, then stop and display the enabled registers and memory blocks.

trace **li**
Execute one source line, regardless of the display mode, then stop and display the enabled registers and memory blocks.

trace **20**
Execute 20 instructions or source lines, depending on the display mode, and display the enabled registers and memory blocks after each trace execution. Ignore breakpoints.

trace **20 in**
Execute 20 instructions, regardless of the display mode, and display the enabled registers and memory blocks after each instruction. Ignore breakpoints.

trace **20 h**
Execute 20 instructions and display the enabled registers and memory blocks after each instruction. Halt if a breakpoint is encountered.

trace **10 cy**
Execute 10 clock cycles and display the enabled registers and memory blocks after each clock cycle.

2.3.32 TYPE: Display the Result Type of C Expression

TYPE **{**c_expression**}**

The **type** command is used to display the result type of a C expression. If result of the expression is a storage location (e.g. just a variable name, or an element of an array), it will display the address of the storage location, in addition to its data type.

EXAMPLES

type **{count}**

Display the type and location of the variable count.

type **{0.5+i}**

Display the type of the given expression.

2.3.33 UNLOCK: Unlock Password Protected Device Type

UNLOCK dev_type password

The **unlock** command provides password enabling for simulation of unannounced device types. Once unlocked, the device type may be selected for simulation using the Simulator **device** command.

EXAMPLE

unlock **56001 x51-234**
Enable device type 56001 for simulation using the password x51-234.

2.3.34 UNTIL: Step Until Address

UNTIL addr [**H**(halt at breakpoints)]

The **until** command sets a temporary breakpoint at the specified line or address, then steps until that breakpoint. It then clears the temporary breakpoint and displays the enabled registers and memory blocks in the same manner as the **step** command.

The addr parameter may be expressed as a line number in the program source file. Specification of a line number is valid only if the symbolic debug information has been loaded from a COFF format .cld file. The debug information is generated using the assembler's -g option. Line numbers may be specified as filename@line_number for a line number in a particular file or simply by line_number for line numbers in the currently displayed file.

As the default, all breakpoints are ignored while the **until** command is executing. The **h** option enables halting at breakpoints.

EXAMPLES

until **20**
Go until the instruction associated with line 20 in the current file is reached.

until **p:$50**
Go until the instruction at hexadecimal address p:50 is reached. Ignore breakpoints.

until **p:$50 h**
Go until the instruction at hexadecimal address p:50 is reached. Do not ignore breakpoints.

until **lab_2**
Go until the instruction at label **lab_2** is reached.

2.3.35 UP: Move Up the C Function Call Stack

up [n]

The up command is used to move up the call stack. It can be used in conjunction with the where, frame, and down commands to display and traverse the C function call stack.

After entering a new call stack frame using **up**, that call stack frame becomes the current scope for evaluation. In other words, for C expressions, the **evaluate** command acts as though this new frame is the proper place to start looking for variables.

EXAMPLES

up
Move up he call stack by one stack frame.

up 3
Move up the call stack by three stack frames.

2.3.36 VIEW: Select Display Mode

VIEW [**A**(assembly)/**S**(source)/**R**(register)]

The **view** command changes the Simulator display mode. There are three display modes: assembly, source and register. See section 1.7 on page 1-5 for a description of the display modes.

If the view command is entered with a parameter, the specified display mode is selected. When no parameter is entered, the display mode cycles to the next display mode in the order source - assembly - register. The same results can be obtained by typing ctrl-w.

EXAMPLES

view
Cycle to next display mode among source, assembly and register modes.

view **s**
Select source display mode.

view **a**
Select assembly display mode.

view **r**
Select register display mode.

2.3.37 WAIT: Wait Specified Time

WAIT [count(seconds)]

The wait command pauses for count seconds or until the user types CTRL-C before continuing to the next command. If the wait command is entered without a count parameter, the command will only terminate if the user types a key.

2.3.38 WASM: GUI Assembly window

WASM [**OFF**]

Wasm is a GUI command that opens an assembly window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wasm

Open an assembly window for the current device.

wasm **off**

Close the assembly window for the current device.

2.3.39 WATCH: Set, Modify, View, or Clear Watch Item

WATCH [#wn] [radix] reg/addr/expression/**{**c_expression**}**

WATCH [#wn] **OFF**

The watch command is used to add, modify, view, and clear watch items. Watch items are on a watch list that gets displayed every time the user does a trace, or a breakpoint is hit. Additionally, any time a user types watch without any parameters, the watch list is displayed.

EXAMPLES

watch **r0**

Add register r0 to the watch list.

watch **x:0**

Add x:0 to the watch list.

watch **{(count+1)%total}**

Add the given C expression to the watch list.

watch **h {count/2}**

Add the given C expression to the watch list, with display radix hex.

watch **b {flag}**

Add the given C variable to the watch list, with display radix binary.

watch **r0+x:0**

Add the expression r0+x:0 to the watch list.

watch

Display the watch list.

watch #**3 off**

Remove item number three from the watch list.

watch **off**

Remove all items from the watch list.

2.3.40 WBREAKPOINT: GUI Breakpoint window

WBREAKPOINT [**OFF**]

Wbreakpoint is a GUI command that opens a breakpoint window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wbreakpoint

Open a breakpoint window for the current device.

wbreakpoint **off**

Close the breakpoint window for the current device.

2.3.41 WCALLS: GUI C Calls Stack window

WCALLS [**OFF**]

Wcalls is a GUI command that opens a C call stack window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wcalls

Open a C call stack window for the current device.

wcalls off

Close the C call stack window for the current device.

2.3.42 WCOMMAND: GUI Command window

WCOMMAND [**OFF**]

Wcommand is a GUI command that opens a command window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wcommand
Open a command window.

wcommand **off**
Close the command window for the current device.

2.3.43 WHERE: GUI C Calls Stack window

WHERE [[+/-]n]

Where is a GUI command that displays the C function Call Stack. Multiple device windows may be opened for debugging target systems with multiple DSPs.

EXAMPLES

where
Display the call stack.

where **3**
Display the three innermost frames in the call stack.

where **-5**
Display the five outermost frames in the call stack.

2.3.44 WINPUT: GUI File Input window

WINPUT [**OFF**]

Winput is a GUI command that opens an input window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

winput
Open an input window for the current device.

winput **off**
Close the input window for the current device.

2.3.45 WLIST: GUI list window

WLIST [**OFF**]

Wlist is a GUI command that opens a list window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wlist **lfile.1st**
Open a list window with the text file lfile.1st displayed.

wlist **win2 lfile.1st**
Open a list window with a window number of 2 with text lfile.txt displayed. If list window 2 already exists, replace the contents with lfile.1st.

wlist **win2 off**
Close list window number 2.

wlist **off**
Close all open list windows.

wlist **win3**
Open a list window with a window number of 3 with no text file displayed.

2.3.46 WMEMORY: GUI Memory window

WMEMORY [**OFF**]

Wmemory is a GUI command that opens a memory window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wmemory **pi**
Open a memory window for the internal program (pi) memory space for the current device.

wmemory **xi 0**
Open a memory window for the xi memory space containing address 0 for the current device.

wmemory **win3 x**
Open a memory window for memory space x with a window number of 3 for the current device.

wmemory **off**
Close all memory windows for the current device.

wmemory **win3 off**
Close memory window 3 for the current device.

2.3.47 WOUTPUT: GUI File Output window

WOUTPUT [**OFF**]

Woutput is a GUI command that opens a file output window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

woutput
Open an output window for the current device.

woutput **off**
Close the output window for the current device.

2.3.48 WREGISTER: GUI Register window

WREGISTER [win_num] [**OFF**]

Wregister is a GUI command that opens a register window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wregister
Open a register window for the current device.

wregister **win3**
Open a register window with a window number of 3 for the current device.

wregister **off**
Close all register windows for the current device.

2.3.49 WSESSION: GUI session window

WSESSION [**OFF**]

Wsession is a GUI command that opens a session window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wsession
Open a session window for the current device.

wsession **off**
Close the session window.

2.3.50 WSOURCE: GUI Source window

WSOURCE [**OFF**]

Wsource is a GUI command that opens a source code window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wsource
Open a source window for the current device.

wsource **off**
Close the source windows for the current device.

2.3.51 WSTACK: GUI Stack window

WSTACK [**OFF**]

Wstack is a GUI command that opens a device stack window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wstack
Open a stack window for the current device.

wstack **off**
Close the stack window for the current device.

2.3.52 WWATCH: GUI watch window

WWATCH [win_num] [#n] [**OFF**]

Wwatch is a GUI command that opens a watch window. Multiple device windows may be opened for debugging simulations with multiple DSPs.

EXAMPLES

wwatch **r0**

Open a watch window for the current device with the register r0 displayed. If the window already exists, add r0 to the list of watched items.

wwatch **x:$100**
Open a watch window for the current device with the memory location x:$100 displayed. If the window already exists, add x:$100 to the list of watched items.

wwatch **r2+3**
Open a watch window for the current device with the expression r2+3 displayed. If the window already exists, add r2+3 to the list of watched items.

wwatch **win2 r0**
Open a watch window for the current device with a window number of 2 with the register r0 displayed. If the window already exists, add r0 to the list of watched items.

wwatch **off**
Close all watch windows for the current device.

wwatch **win3 off**
Close watch window 3 for the current device.

wwatch **#2 off**
Remove watch element #2 from first watch window's list of watched elements.

wwatch **win4 @2 off**
Remove watch element #2 from watch window 4's list of watched elements.

2.4 DEBUGGING C PROGRAMS

The Simulator software is capable of loading programs compiled with the Motorola Optimizing C Compiler, and also has features which aid in the process of debugging such programs. This section provides background information on what features are available, and gives examples of the commands that implement these features. The main thrust of this section is the tutorials at the end, which give practical examples of how the debugging features might be used. No special mode needs to be entered to debug C programs, and all of the familiar Simulator capabilities are available while debugging C programs.

2.4.1 C Debug Features

The features available for debugging C programs include:

- Step line by line through C programs.
- Examine and change the value of C variables.
- Evaluate complex C expressions, including the ability to call C functions from the command line.
- Set breakpoints based on C expressions.
- Add C variables and expressions to a watch list.
- Examine and traverse the C function call stack, examining local variables and parameters at each level of the stack.
- Redirect C input and output.
- Determine the type and location of a C variable.

2.4.2 C Expressions

C expressions may be used as arguments to the **break**, **evaluate**, **type**, and **watch** commands. Expressions must be surrounded by the left and right curly braces (**{** and **}**). This is so that expressions can have spaces in them yet will still be considered a single parameter to a command. Any valid C expression can be used between the braces, with the exception of expressions that contain things mentioned in the following section on restrictions. For information on what makes up a valid C expression, consult a manual on the C programming language.

In addition to supporting basic C expressions, a new operator (#) has been added. This new operator is used to “create” an array from a pointer or another array. The syntax of the operator is:

name#size

where “name” is the name of a pointer or array in the C program, and “size” is a constant integer greater than zero indicating what size array to make. So for instance, if “vals” is a pointer to a group of integers, “vals#10” is an array of the first 10 integers. This can be useful for display purposes. This operator can be used to make single dimensional arrays only. Attempting something like “(name#size1)#size2” will make a one dimensional array with “size2” elements.

One final addition to C expressions is the ability to use DSP registers in expressions by

prefixing them with a dollar sign ($) in the C expression. For registers that are greater than the size of a “long” variable, the upper bits are truncated. So for example, if “$a” were specified in the 56000 software, only the lower 48 bits of register A would be used. PLEASE NOTE: The $ in non-C_expression evaluation is used to designate a hexadecimal value.

2.4.3 Restrictions

To improve usability, an effort has been made to have the fewest possible restrictions, and although some remain, they are very reasonable. The first restriction is that string literals are not supported in expressions. This would have required allocating some portion of the DSP memory for debugging purposes, possibly interfering with the user’s code. The other restriction is on type casts. Only forms of type casting such as the following are allowed:

(type)

(type *)

(enum enumeration_tag)

([struct|union|enum] structure/union/enumeration_tag *)

In these examples, “type” includes both basic C types, and types that were defined with typedef in the C program.

2.4.4 Compiling a Program for Debugging

To use the C debugging features included in the User Interface program, the C program being loaded into the DSP must have been compiled using the “compile with debugging information” flag available in the Compiler. For the Motorola Optimizing C Compilers, this flag is “**-g**”. By default the Motorola Optimizing C Compilers compile programs with optimization turned on. This will **not** be affected by compiling with debugging turned on. Since optimization can change the order in which portions of programs execute, along with eliminating variables, placing variables into registers, etc., you may experience strange behavior when debugging programs that have been optimized. When compiling with the “**-alo**” flag, this strange behavior might be considerably more noticeable. If this is the case, compile with the “**-fno-opt**” flag, which disables optimization.

2.4.5 C Debugging Commands

Certain commands (where, up, down, frame, streams, redirect, and type) exist specifically for debugging C programs, while other commands (break, evaluate, finish, go, next, step, trace, until and watch) are useful in debugging C programs, but are also used in assembly language debugging.

To eliminate duplicated functionality, the evaluate command is used in C debugging as the change, display, and evaluate commands would be used in assembly language debugging. For instance, to display a C variable, evaluate that variable. To change the value of a C variable, evaluate an expression that has an assignment to that variable. Evaluate is used just as it would be for an assembly language expression to evaluate and display

the result of a C expression. In addition to the result, the type of the result is displayed. For example when evaluating an expression that involves long integer variables, the result type displayed would be “long.”

Chapter 3
DEVICE I/O AND PERIPHERAL SIMULATION

3.1 INTRODUCTION

The DSP on-chip peripherals are simulated on a cycle by cycle basis by the SIMDSP program. The Simulator **input** and **output** commands provide a method of assigning file or terminal I/O to each peripheral, as well as to individual memory locations and individual or groups of device pins. This chapter describes the file formats used by the **input** and **output** commands.

3.2 I/O FILE CONTENTS

All file information is represented in **ASCII**, so the I/O files can be conveniently edited or printed. The file may contain repeat punctuation, comments, timing information, peripheral data, pin data, or memory data.

3.2.1 I/O File Repeat Punctuation

The Simulator provides a way to specify repeated input or output data values and sequences. A single data value can be repeated by specifying **#count** following the data item. A group of data items can be indicated by enclosing the group in parentheses. The entire group can then be repeated by placing **#count** immediately following the closing parenthesis. The parentheses can be nested. A closing parenthesis without a following repeat count will cause the data sequence within the parentheses to repeat forever.

Timed values can appear within a repeat group, but in this case, the relative time mode (**+**time) should be used.

EXAMPLES

1FF#20
Repeat the untimed data item **1FF** twenty times.

(+5 CC +10 33)#5
Repeat the sequence of timed data pairs **+5 CC +10 33** five times.

(CC354 CC333 C7000)
Repeat the untimed data sequence **CC354 CC333 C7000** forever.

(1#5 0#5)
Repeat the untimed data sequence **1 1 1 1 1 0 0 0 0 0** forever.

3.2.2 I/O COMMENT

Any information following a semicolon and up to the end-of-line is considered to be a user comment and is not interpreted as input data or timing.

EXAMPLE

FFC 333 972 ;next three p memory data words

The first three data values are applied to the device. The information following the semicolon is a user comment.

3.2.3 I/O File Timing Information

If the **T** keycharacter is specified in the **input** or **output** command, then the assigned file will contain cycle timing information preceding each piece of I/O data. The timing information relates to the Simulator cycle counter value (**cyc** register) at the time when the data transfer occurs. The timing information is always expressed in decimal. If the timing information is preceded by a plus sign (**+**), it indicates a relative number of cycles from the preceding specified timing value; otherwise it indicates the exact value of the Simulator **cyc** register at the time of the transfer.

3.2.4 I/O File Peripheral Data

Each DSP peripheral can have an assigned input or output file. General information that applies to all peripheral files appears below. For more information concerning a specific DSP peripheral see Chapter 8, Peripheral I/O.

The peripheral data value can be represented in hexadecimal, decimal, binary or floating point. The default input radix can be specified in the **input** command; the output radix can be specified by the **output** command.

Floating point input can be expressed in the usual methods. For example, 0.5, 5e-1, and 5.0E-1 are all acceptable data input values. If a data value contains a decimal point, the data will be input as a floating point value, overriding the input radix specification. Likewise, a data value preceded by **$** will always be input as hexadecimal, a value preceded by **‘** will always be input as decimal, and a data value preceded by % will be input in binary.

Untimed peripheral input data values will be applied only during cycles when the peripheral function is enabled. Some peripherals retrieve data from the input file when the peripheral would normally receive new data; other peripherals retrieve the data each cycle. See Chapter 8, Peripheral I/O for specific peripheral information. The final specified data value will remain applied to the peripheral indefinitely. The repeat punctuation and repeat count can be used to specify durations of longer than one cycle.

Timed peripheral input data values will be applied to the peripheral at the time intervals, or at the exact Simulator cycles indicated by the timing information within the file. If the first timing information in the file is a relative value, (timing preceded by **+**) the Simulator will wait until the peripheral function is enabled before applying the data.

If a lower case letter **t** is placed in a data position of the input file, the user will be prompted for the next input data value as described in Section 3.2.8.

Storage of data to the peripheral output file will begin when the peripheral is enabled. In the timed output mode, the Simulator cycle count and a data value are stored each time the peripheral output changes. In the untimed output mode, a data value and a following repeat count are stored each time the data changes.

3.2.5 I/O File Port Data

When assigned to a DSP port, the input file data value represents the value applied to all the pins of the port. The least significant port bit maps to the least significant bit of the data value. General information that applies to all port files appears below. For more information concerning a specific DSP port see Chapter 8, Peripheral I/O.

A port is simply some convenient grouping of device pins. Untimed data applied to a port is retrieved each clock cycle, with one exception: the data bus ports retrieve new data from an assigned input file only once for each memory fetch. See Chapter 8, Peripheral I/O for information on a specific ports.

The port data value can be represented in hexadecimal, decimal, binary or floating point. The default input radix can be specified in the **input** command; the output radix can be specified by the **output** command.

Floating point input can be expressed in the usual methods. For example, 0.5, 5e-1, and 5.0E-1 are all acceptable data input values. If a data value contains a decimal point, the data will be input as a floating point value, overriding the input radix specification. Likewise, a data value preceded by **$** will always be input as hexadecimal, a value preceded by **‘** will always be input as decimal, and a data value preceded by % will be input in binary.

Timed port input data values are applied to the port at the specified relative time intervals (**+**time), or at the exact Simulator cycle indicated by the timing information within the file.

If a lower case letter **t** is placed in a data position of the input file, the user will be prompted for the next input data value as described in Section 3.2.8.

Storage of data to a port output file will occur any time a write operation occurs to the port. In the timed output mode, the Simulator cycle count and a data value are stored each time a word is written. In the untimed output mode, a single data value is stored each time a word is written. No tristate information is stored in the port output data.

3.2.6 I/O File Memory Data

When assigned to a memory location, the input file data value supplies the value that is read when the Simulator references that memory location. The least significant memory bit maps to the least significant bit of the data value.

The input data value can be in decimal, binary, hexadecimal, or floating point form. The Simulator will interpret the data based on the input radix specified in the **input** command. The default input radix is hexadecimal. If a data value contains a decimal point, the data will be input as a floating point value, overriding the input radix specification. Likewise, a data value preceded by **$** will always be input as hexadecimal, a value preceded by **‘** will always be input as decimal, and a data value preceded by % will be input in binary.

Untimed memory input data values will be applied each time the device performs a read operation on the memory location. In other words, the input file acts like a stack of input data; each successive data value is retrieved from the "stack" file when a read operation occurs.

Timed input data values are applied to the memory location at the specified relative time intervals (**+**time), or at the exact Simulator cycle indicated by the timing information within the file. If the first timing information in the file is a relative value (**+**time) the Simulator will wait until the first read of that memory location before getting the first data value. Otherwise the data application occurs at the exact specified cycle.

If a lower case letter **t** is placed in a data position of the input file, the user will be prompted for the next input data value as described in Section 3.2.8.

Storage of data to a memory output file will occur any time a write operation occurs to the memory location. In the timed output mode, the Simulator cycle count and a data value are stored each time a word is written. In the untimed output mode, a single data value is stored each time a word is written.

The output data value can be in decimal, binary, hexadecimal, floating point or string form. The output radix is specified in the **output** command. The default output radix is hexadecimal. The string form of output data uses the value written to the memory location as the starting address in the same memory space of a zero terminated ASCII character string. The character string is written to the output file.

EXAMPLES

The following untimed memory input file will cause the data sequence **7FFF 7F3F 5D3C 7FC3** to appear during consecutive reads of the specified memory location.

```
7FFF 7F3F 5D3C 7FC3
```

The following untimed memory input file will cause the data sequence **1FF 0** to appear repeatedly during consecutive reads of the specified memory location.

```
(1FF 0)
```

The following timed memory input file will cause the data sequence **0.5 0.3** to alternate in the specified memory location 10 cycle intervals.
(+10 0.5 +10 0.3)

The following timed memory input file will cause **1C3** to appear in the specified memory location at cycle 2000, and **1CF** to appear at cycle 2005.
2000 1C3 2005 1CF

3.2.7 I/O File Pin or Pin Group Data

When assigned to a pin or pin group, the input file data value supplies the zero (**0** or **L**), one (**1** or **H**), a negative pulse within a single cycle(**N**), a positive pulse within a single cycle(**P**), or a tristate (**X**) value to be applied to the pin or to each pin in the group, with the first specified pin mapping to the least significant bit of the data value. Each data word must contain as many characters (**0**, **1**, **L**, **H**, **N**, **P**, or **X**) as there are pins in the group.

If an analog input file is assigned to a device analog pin, the input command must specify the floating point radix with the **-rf** radix designator in the input command. Likewise, an analog output pin file must be specified with the **-rf** radix designator in order to generate floating point output for the pin rather than the digital pin values described in the preceding paragraph. Floating point io files may only be assigned to a single analog pin.

Untimed pin input data values will be applied to the specified pin each Simulator clock cycle.

Timed pin input data values will be applied to the specified pin at the specified relative time intervals (**+**time), or at the exact Simulator cycle indicated by the timing information within the file.

If a lower case letter **t** is placed in a data position of the input file, the user will be prompted for the next input data value as described in Section 3.2.8.

Storage of pin data to an output file will occur any time the pins data value changes, including changes to tristate, 1, 0, H or L. In the timed output mode, the Simulator cycle count and a data value are stored each time a word is written. In the untimed output mode, a single data value is stored each time a word is written.

The Simulator also provides a special mode that allows pin data input to be received from the output of another pin without the necessity of an intermediate disk file.

EXAMPLES

The following untimed Reset pin input file will cause the Reset pin to go low for two cycles, then back high.
0 0 1

The following timed IRQA pin input will cause the IRQA pin to go low at cycle 12000, then back high at cycle 12010.
12000 0 12010 1

The following timed IRQB pin input will cause the IRQB pin to go low after 200 cycles and stay low for 20 cycles. The sequence is repeated 9 times.
(+200 0 +20 1)#9

3.2.8 Terminal Input of Data Values

There are two levels of terminal data input capability provided by the Simulator. If the **input** command specifies **term** as the input filename, the Simulator enters an editor which allows creation of an input data file without leaving the Simulator. The data file is given a temporary name, **termxxxx.io** or **termxxxx.tio** (xxxx=0000-9999), and is saved on the disk at the termination of the input command. The entire contents of the input file may be specified in this manner, including any of the valid fields specified in Chapter 3.

A second level of terminal data input allows the user to be prompted any time the next input data value is needed. This method is triggered if the lower case letter **t** is encountered in the data field of the input file. This is only valid for the data field, not for the time field. Each time a **t** is encountered, the user will be prompted for a single data value from the terminal. The Simulator will read the input data using the radix specified in the input command. Hexadecimal is the default input radix. If the user just types the **return** key at the prompt, without entering a data value, the previous data value will be repeated. If the user types the **esc** key at the prompt, an end-of-file status will be simulated and the previous data value will repeat forever.

EXAMPLES

The following untimed IRQA pin input file will prompt the user for a new input value every 45 clock cycles.
(t#45)

The following untimed memory file input data will prompt the user for the third and fifth values that are read from the specified memory location.
ffcc c1000 t ab12 t 6444

The following timed port input file will prompt the user for port input data at cycle 566 and 800 after alternating the input data sequence 5555 3333 three times at ten cycle intervals.
(+10 5555 +10 3333)#3 566 t 800 t

Chapter 4
SIMULATOR MEMORY CONFIGURATION

4.1 INTRODUCTION

Simulation of a specific DSP device configuration can be selected using the **DEVICE** command. The internal memory attributes are determined by the selected device configuration. External memory accesses are also determined by the device configuration, but the effects of writing external RAM, ROM or peripherals can be totally controlled by the user. The Simulator package provides total flexibility for the user to define external memory responses by supplying the C language source code for the external memory functions. The source code used as a default is contained in the file **simvmem.c**. The external memory access function requirements are described fully in Chapter 7.

4.2 SIMULATOR DEFAULT MEMORY CONFIGURATION

The Simulator will contain the predefined memory map of the default device type as the default memory configuration. The DSP can be configured to exit its reset state in a predefined operating mode. Once the Simulator is active the operating mode can be changed under program control by changing the value of the device Operating Mode Register (OMR). When the Simulator is invoked, the mode pins will be configured for the default operating mode (See Chapter 8, Operating Modes). The operating mode that the device simulates following a simulated hardware reset can be selected using the Simulator **RESET** command.

The full external memory map of the device is, by default, RAM memory. The large external memory space is simulated using a virtual memory technique which automatically pages memory blocks to disk if the operating environment fails to allocate the required space in memory.

The on-chip bootstrap and data ROM areas can be modified using the Simulator **CHANGE** or **ASM** commands. The bootstrap ROM is specified by using **PR:** as the memory space designator. For example, **ASM PR:0** will begin assembly in the bootstrap ROM at location 0. The data ROMs can be specified by **XR:** or **YR:**. Loading an assembler output file with the Simulator **load** can also modify the bootstrap ROM or data ROM areas. The ROM areas can be reinitialized using the Simulator **RESET S** command.

Chapter 5
EXPRESSIONS

5.1 INTRODUCTION

The Simulator allows an expression to be used in most places where a constant is valid. For example, an expression can take the place of the start and stop location in the specification of an address range. An expression is a combination of symbols, constants, operators, and parentheses. Expressions follow the conventional rules of algebra and boolean arithmetic.

Expressions may contain any combination of integers, floating point numbers, memory space symbols and register symbols.

5.2 MEMORY SPACE SYMBOLS

The Simulator evaluator interprets a memory space symbol followed by an expression as the contents of a memory location. Use the Simulator's "help mem" command to obtain a list of the valid memory space prefixes and their corresponding address ranges. The following expression is converted to an integer constant in the address range of the DSP. Some memory space symbols, such as **p:** or **x:**, require the evaluator to first read the device Operating Mode Register (OMR). Others, such as **xi:** or **pr:**, refer to an exact memory location regardless of the chip operating mode.

5.3 REGISTER NAME SYMBOLS

The Simulator evaluator interprets a register symbol as the contents of that register. A list of valid register names can be obtained using the Simulator "help reg" command.

NOTE: Some hexadecimal constants, such as **a0** or **d1**, may also be valid register names for the selected DSP. It is necessary to precede such hexadecimal constants by a dollar sign ($) to distinguish them from registers of the same name.

5.4 ASSEMBLER DEBUG SYMBOLS

The Simulator load command processes the symbol and line number information present in a COFF format object file (.cld file) which has been generated with the assembler's -g option. If symbol information has been loaded, the evaluator will accept symbol names or source file line numbers and translate them into an associated memory address.

In general a symbol name may be referenced in the Simulator just as it was defined in the original source file, except that symbol names which conflict with a Simulator register name must be preceded by the @ character. A symbol name may be further delimited by specifying a containing section name in the form section_name@symbol_name, with the @ character being used as the separator. The section name **global** may be used for the global section. If a symbol is specified without a preceding section name, the evaluator assumes the section containing the current pc.

Line numbers may be expressed simply as a decimal integer preceded by the @ character when referring to a line in the current source file. If an address field is being specified in a command, the line number's preceding @ character may be omitted. A line number in a particular source file may be expressed in the form source_filename@line_number.

Below are valid forms of symbol names and line numbers:

symbol_name - translates to the address associated with symbol_name

Example: change pc lab_d

@symbol_name - translates to the address associated with symbol_name

Example: disassemble @start_1

section_name@symbol_name - translates to the address associated with symbol_name in section section_name

Example: display sec3@xdata

@section_name@symbol_name - translates to the address associated with symbol_name in section section_name

Example: display @sec3@xdata

line_number - translates to the address associated with line_number in the current source file.

Example: break 30

@line_number - translates to the address associated with line_number in the current source file.

Example: change pc @30

source_filename@line_number - translates to the address associated with line_number in the named source file.

Example: change pc test.asm@30

@source_filename@line_number - translates to the address associated with line_number in the named source file.

Example: change pc @test.asm@30

source_filename- translates to the address associated with the first line in the named source file.

Example: list test.asm

@source_filename- translates to the address associated with the first line in the named source file.

Example: list @test.asm

5.5 CONSTANTS

Constants represent quantities of data that do not vary in value during the execution of a program.

5.5.1 Numeric Constants

The numeric constants can be in one of three bases:

Binary - Binary constants consist of a percent sign (**%**) followed by a string of binary digits (0,1).

Example: %11010

Hexadecimal - Hexadecimal constants consist of a dollar sign (**$**) followed by a string of hexadecimal digits (0-9,A-F or a-f).

Example: $12FF, $12ff

Decimal - Decimal constants can be either floating point or integer. Integer decimal constants consist of a string of decimal (0-9) digits. Floating point constants are indicated either by a preceding, following, or included decimal point or by the presence of an upper or lower case 'E' followed by the exponent. The special constants **inf** and **nan** can be used in floating point expressions to represent the IEEE floating point values of infinity and not-a-number for DSP devices which operate with IEEE floating point values.

Example: 12345(integer)
6E10(floating point)
.6(floating point)
2.7e2(floating point)

A constant can be written without a leading radix indicator if the input radix is changed using the **RADIX** directive. For example, a hexadecimal constant can be written without the leading dollar sign (**$**) if the input radix is set to **hex**. The default input radix is decimal. See the **RADIX** directive for more information.

5.6 OPERATORS

Some of the Evaluator operators can be used with both floating point and integer values. If one of the operands of the operator has a floating point value and the other has an integer value, the integer will be converted to a floating point value before the operator is applied and the result will be floating point. If both operands of the operator are integers, the result will be an integer value. Similarly, if both the operands are floating point, the result will be a floating point value.

Operators recognized by the Assembler include the following:

5.6.1 Unary operators:

minus	(**-**)	
negate	(_~)	- Integer only
logical negate	(**!**)	- Integer only

The unary negate operator will return the one's complement of the following operand.

The unary logical negation operator will return an integer 1 if the operand following it is 0 and will return a 0 otherwise. The operand must have an integer value.

5.6.2 Arithmetic operators:

addition	(**+**)
subtraction	(**-**)
multiplication	(*****)
division	(**/**)
mod	(**%**)

The divide operator applied to integer numbers produces a truncated integer result.

The mod operator applied to integers will yield the remainder from the division of the first expression by the second. If the mod operator is used with floating point operands, the mod operator will apply the following rules:

Y % Z = Y if Z = 0
= X if Z <> 0

where X has the same sign as Y, is less than Z, and satisfies the relationship:

Y = i * Z + X

where i is an integer.

5.6.3 Bitwise operators (binary):

AND	(**&**)	- Integer only
inclusive OR	(**|**)	- Integer only
exclusive OR	(**^**)	- Integer only

Bitwise operators cannot be applied to floating point operands.

5.6.4 Shift operators (binary):

shift right	(**>>**)	- Integer only
shift left	(**<<**)	- Integer only

The shift right operator causes the left operand to be shifted to the right (and zero-filled) by the number of bits specified by the right operand.

The shift left operator causes the left operand to be shifted to the left by the number of bits specified by the right operand. The sign bit will be replicated.

Shift operators cannot be applied to floating point operands.

5.6.5 Relational operators:

less than	(**<**)
greater than	(**>**)
equal	(**==**) or (**=**)
less than or equal	(**<=**)
greater than or equal	(**>=**)
not equal	(**!=**)

Relational operators all work the same way. If the indicated condition is true, the result of the expression is an integer 1. If it is false, the result of the expression is an integer 0. For example, if D has a value of 3 and E has a value of 5, then the result of the expression D**<**E is 1, and the result of the expression D**>**E is 0. Each operand of the conditional operators can be either floating point or integer. Test for equality involving floating point values should be used with caution, since rounding error could cause unexpected results.

Relational operators are primarily intended for use with the Simulator **BREAK** command.

5.6.6 Logical operators:

Logical AND (**&&**)
Logical OR (**||**)

The logical AND operator returns an integer 1 if both of its operands are non-zero; otherwise, it returns an integer 0.

The logical OR operator returns an integer 1 if either of its operands is non-zero; otherwise it returns an integer 0.

The types of the operands may be either integer or floating point.

Logical operators are primarily intended for use with the Simulator **BREAK** command.

5.7 OPERATOR PRECEDENCE

Expressions are evaluated with the following operator precedence:

1. parenthetical expression (innermost first)
2. unary minus, unary negate, unary logical negation
3. multiplication, division, mod
4. addition, subtraction
5. shift
6. less than, greater than, less or equal, greater or equal
7. equal, not equal
8. bitwise AND
9. bitwise EOR
10. bitwise OR
11. logical AND
12. logical OR

Operators of the same precedence are evaluated left to right. All integer results (including intermediate) of expression evaluation are 32-bit, truncated integers. Valid operands include numeric constants, memory addresses, or register symbols. The logical, bitwise, unary negate, unary logical negation and shift operators cannot be applied to floating point operands. That is, if the evaluation of an expression (after operator precedence has been applied) results in a floating point number on either side of any of these operators, an error will be generated.

Chapter 6
DSP OBJECT MODULE FORMAT

6.1 INTRODUCTION

The DSP COFF object module format, which is produced by the DSP Macro Cross-Assemblers beginning with release 4.0, is described in full in the assembler reference manual. The COFF format object files can also be produced using the Simulator **save** command by specifying the ".cld" suffix for the output file. The ASCII object module format (OMF) produced by the Simulator **save** command and by versions of the DSP Assembler prior to release 4.0 is also referred to as the ".lod" format in reference to the default ".lod" suffix of the filename. The remainder of this chapter describes the ".lod" format object files.

The ".lod" OMF is an ASCII file consisting of variable-length text records. Records may be defined with a fixed number of fields or contain repeating instances of a given field (such as instructions or data). Fields within the records are separated by whitespace characters (blank, tab, form feed, newline). The general format for a DSP OMF record is illustrated below ("ws" is whitespace).

_<**TYPE**><ws><field1><ws><field2><ws>...<fieldn>

Every record starts with a type definition field; this field begins with an underscore (_) character. For records with repeating fields, the underscore character indicates where one record ends and another begins. A scanning program would examine the first character of each field looking for the underscore character. If found, the program would know it had encountered a new record and would use the remainder of the field to determine the record type. The type definition may be upper or lower case, although the assembler guarantees upper case output.

The only exception to this processing is when a comment occurs in the object file as a result of an IDENT or COBJ assembler directive. Comments in the object file are bracketed by newline characters and thus appear on a line by themselves. Since the location of comment fields in an OMF record is well defined, scanning software need only look for an opening and closing newline sequence to determine the bounds of a comment.

The assembler will fill lines in the object file to a maximum of 80 characters, using the minimum white space (one blank or newline) to delimit fields. Records with repeating fields may be of arbitrary length.

6.2 RECORD DEFINITIONS

There are six DSP OMF record types defined. The record types are START, END, DATA, BLOCKDATA, SYMBOL, and COMMENT; currently DATA records are used for both code and data.

Start Record

Format: _**START** <Module id> <Version> <Rev #> <Device #> <Asm Version> <Comment>

The **START** record begins an assembler object module file. The information contained in the record corresponds to the parameters in the first valid IDENT directive encountered in the assembler input. If no IDENT directive is given, the assembler uses the input file name (without extension) as the module name, supplying zero for version and revision numbers and an empty comment field (which appears as a blank line in the object file).

The module id field conforms to the definition of a legal assembler symbol, that being a series of up to eight ASCII characters starting with an alphabetic character and followed by alphanumeric characters or the underscore (_). The version and revision numbers are ASCII numeric values corresponding to the expressions found in the IDENT directive. The device number and assembler version fields indicate the target device number and the version number of the assembler that created the object module.

End Record

Format: _**END** <Entry point address>

The **END** record terminates an object module file. The only field in the record contains an address which is the result of the expression in an END directive. If no END directive was encountered in the assembler source, the address is the result of the expression found in the first valid ORG assembler directive with a reference to runtime program memory space (P). The address is in ASCII hex format; it contains only the hex digits 0-F, with no special radix characters such as a leading ’0X’ or trailing ’H’. Use the Simulator’s "help mem" command to obtain a list of the valid memory space prefixes and the address range for each memory space.

Data Record

Format: _**DATA** <Memory space> <Address> <Code/data> ...

The **DATA** record is used to load values based on the specifier in the memory space field. The space specifier consists of one to three characters representing the memory space to be loaded. Use the Simulator’s "help mem" command to obtain a list of the valid memory space prefixes.

The characters may be upper or lower case, although the assembler guarantees upper case output. The address is an ASCII hex value indicating where to begin loading in the specified memory space. It contains only hex digits 0-F, with no leading or trailing radix characters.

A variable number of ASCII hex values to load follows the starting address. These values are in the same format as the load address, that being hex digits only with no radix indicator. The list ends when a field is read with an underscore in the first character position, signaling the start of a new record.

In the case of DATA records with an L space memory specifier, the data values will be paired high:low such that the first data value in the pair will be loaded into the X memory space and the second data value will be loaded into Y memory space.

BlockData Record

Format: _**BLOCKDATA** <Mem space> <Addr> <Count> <Value>

The **BLOCKDATA** record provides a shorthand method for loading repeated data values, as might appear in a block constant storage (BSC) assembler directive. This makes the object file more compact, but requires more work on the part of the loading software.

The space specifier consists of one to three characters representing the memory space to be loaded. Use the Simulator's "help mem" command to obtain a list of the valid memory space prefixes.

The characters may be upper or lower case, although the assembler guarantees upper case output. The address is an ASCII hex value indicating where to begin loading in the specified memory space. It contains only hex digits 0-F, with no leading or trailing radix characters.

The count field specifies the number of times the following value is to be loaded into consecutive memory locations starting at the load address. The count value has the same format and range as the starting address, and should be interpreted as an unsigned integer. The value field contains the value to be loaded. It has the same format and range as the values in a standard DATA record (hex digits 0-F).

Symbol Record

Format: _**SYMBOL** <Mem space> < <Symbol> <Address> > ...

The **SYMBOL** record contains information about symbols (labels) found in the assembler source file. SYMBOL records are created at the end of assembly as the result of a SYMOBJ directive or the SO assembler option.

The space specifier consists of one to three characters representing the memory space to be loaded. Use the Simulator's "help mem" command to obtain a list of the valid memory space prefixes.

An arbitrary number of symbol/address pairs follows the memory space attribute. The symbol field conforms to the definition of a legal assembler symbol, that being a series of up to eight ASCII characters starting with an alphabetic character and followed by alpha-numeric characters or the underscore (_). The address is an ASCII hex value indicating the address at which the symbol was defined. It contains only hex digits 0-F, with no leading or trailing radix characters.

Comment Record

Format: _**COMMENT**
<Comment>

The **COMMENT** record puts a comment into the object file; it is produced via the COBJ assembler directive. The comment text appears on a line by itself in the object file; it is delimited by newline characters.

Chapter 7
C LIBRARY FUNCTIONS

7.1 INTRODUCTION

The SIMDSP Simulator package includes several object code libraries of Simulator functions that were used to create the Simulator. The libraries allow the user to build his own customized Simulator and integrate it with his unique project. Section 7.2 documents each Simulator entry point that is available to the user.

The Simulator package includes the source code for many of the SIMDSP functions, including the code that defines the DSP external memory accesses, the code for the main entry point, the code for the terminal I/O functions, and example code for a non-display version of the Simulator. The source code can be modified to create a Simulator customized for a particular application. Section 7.3 provides a description of the external memory access functions. Section 7.4 provides a description of the terminal I/O functions.

Object libraries are supplied which support display or non-display versions of the Simulator. The user may choose to eliminate the user interface functions altogether and control the simulation directly through lower level function calls. Topics concerning the non-display version of the Simulator are discussed in section 7.5.

Simulation of multiple DSP devices is fully supported by the DSP library functions. Section 7.6 discusses topics related to simulating and interconnecting multiple DSP devices.

Section 7.7 provides a description of the public function names used by the Simulator.

Section 7.8 gives a description of the global variables used by the Simulator.

Section 7.9 describes modifications that can be made to the Simulator global structures.

7.2 SIMULATOR OBJECT LIBRARY ENTRY POINTS

The following is a quick reference list of the higher level Simulator entry points provided in the Simulator object libraries. The prefix indicates whether or not the function is available in the non-display version of the Simulator. Function names beginning with the prefix **dsp**_ or **dspt**_ are available to both the display and non-display versions of the Simulator, while function names beginning with **sim**_ are only available when using a display version of the Simulator. The **dspt**_ prefix indicates a device dependent function. The _**xxxxx** suffix on these indicate a device family number. Lower level Simulator functions, which have a prefix of **dspl**_, **siml**_ or **dsptl**_, are not intended for direct access by the user's program. They are not described in this document. The higher level functions listed below are described in detail in Sections 7.2.1 through 7.2.30.

dspt_masm_xxxxx(mnemonic,ops,err);	Assemble mnemonic string to ops
dspt_unasm_xxxxx(ops,sr,omr,sdbp);	Disassemble DSP opcodes
dsp_exec(devn);	Execute one clock cycle for DSP device
dsp_findmem(devn,memname,map);	Get map index for memory prefix
dsp_findpin(devn,pinname,pinnum);	Get pin number for pin name
dsp_findport(devn,portname,pnum,pmask);	Get port number and mask for port name
dsp_findreg(devn,regname,pval,rval);	Get peripheral and register index for register
dsp_fmem(devn,map,addr,blocksz,val);	Fill memory block with a value
dsp_free(devn);	Free memory allocated for a DSP device
dsp_init(devn,mode);	Initialize selected device and mode
dsp_ldmem(devn,filename);	Load device memory from filename
dsp_load(filename);	Load all device states from filename
dsp_new(devn,device_type);	Create new DSP device
dsp_path(path,base,suffix,new_name);	Create filename from path, base and suffix
dsp_rapin(devn,pin_number,val);	Read output analog pin state from device
dsp_rmem(devn,map,addr,mem_val);	Read dsp memory map address to mem_val
dsp_rpin(devn,pin_number);	Read output pin state from device
dsp_rport(devn,port,data,force);	Read output port state from device
dsp_rreg(devn,periphn,regn,regval);	Read DSP peripheral register to regval
dsp_save(filename);	Save the state of all devices to filename
dsp_startup();	Initialize Simulator structures
dsp_unlock(device_type,password);	Unlock password protected device type
dsp_wapin(devn,pin,value);	Write device analog input pin with value
dsp_wmem(devn,map,addr,val);	Write DSP memory map address with val
dsp_wpin(devn,pin,value);	Write device input pin with value
dsp_wport(devn,port,mask,data,force);	Write device port with data and force value
dsp_wreg(devn,periphn,regn,regval);	Write DSP peripheral register with regval
sim_docmd(devn,command_string);	Perform Simulator command on DSP device

sim_gmcmd(devn,command_string);	Get command string from macro file
sim_gtcmd(devn,command_string);	Get command string from terminal

7.2.1 dspt_masm_xxxxx: Assemble DSP Mnemonic

```
int dspt_masm_xxxxx(mnemonic,ops,error_ptr)
char *mnemonic;          /* Pointer to assembler mnemonic string */
unsigned long *ops;      /* Where to put the words of assembled opcode */
char **error_ptr;        /* Will point to error message if an error occurs */
```

This function invokes the single line assembler to assemble a DSP mnemonic. It returns one of the following integer codes:

-1 An error occurred. The user supplied error pointer will point to a message that explains the error.

0 The line mnemonic provided was a comment

1 The mnemonic assembled correctly and required 1 word of code. The code will be in the ops[0] location.

2 The mnemonic assembled correctly and required 2 words of code. The first word will be in placed in ops[0], the second in ops[1].

3 The mnemonic assembled correctly and required 3 words of code. The first word will be in placed in ops[0], the second in ops[1], the third in ops[2].

Note that the **xxxxx** in the function name should be replaced by a device family number. It should be 56k for the 56000 family devices, 56100 for the 56100 family devices, and 96k for the 96000 family devices.

EXAMPLE

```
/* Assemble the instruction "move r0,r1" */
unsigned long opcodes[3];
char *error_ptr;
int retval;
retval=dspt_masm_56k("move r0,r1",&opcodes[0],&error_ptr);
```

7.2.2 dspt_unasm_xxxxx: Disassemble DSP Mnemonics

```
int dspt_unasm_xxxxx(ops,return_string,sr,omr,gdbp)
unsigned long *ops;        /* Pointer to opcodes to be disassembled */
unsigned long sr;          /* Value of device status register */
unsigned long omr;         /* Value of device operating mode register */
char *gdbp;                /* Return value reserved for use by debugger*/
char *return_string;       /* Pointer to return character buffer */
```

This function disassembles ops[0] (and possibly ops[1] and ops[2] if ops [0] requires a second or third word) and places the disassembled mnemonic in the return_string buffer supplied by the user. If correct disassembly requires a device status register and/or operating mode register value, the values should be provided in the sr and omr parameters. The gdbp parameter is a pointer reserved for use by the symbolic debugger, and should be NULL for other applications.

The mnemonic may require as many as 120 characters of return buffer. The function returns the number (1 to 3) of words consumed by the disassembly. It returns 0 for illegal opcodes and a return string containing a DC directive.

Note that the **xxxxx** in the function name should be replaced by a device family number. It should be 56k for the 56000 family devices, 56100 for the 56100 family devices, and 96k for the 96000 family devices.

EXAMPLE

```
/* Disassembly of the opcode representing NOP */
unsigned long ops[3];      /*Instruction words to be disassembled.*/
char return_string[120];   /*The return mnemonic goes here.*/
int numwords;              /*Number of operands used by disassembler.*/
ops[0]=0L;
ops[1]=0L;
ops[2]=0L;
numwords=dspt_unasm_56k(ops,return_string,0L,0L,NULL);
/* Now numwords==1, return_string=="nop"*/
```

7.2.3 dsp_exec: Execute Single Device Clock Cycle

```
dsp_exec(device_index)
int device_index;          /* DSP device index to be affected by command */
```

This function executes a single DSP device cycle and updates the selected dsp device structure. All device inputs, outputs and registers are updated as a result of this call. In addition, it tests user-defined breakpoint conditions and clears the device's **executing** status flag if a breakpoint occurs. It also calls the functions which handle cycle by cycle I/O from assigned input or output files.

EXAMPLE

```
/*Execute 1000 cycles on a device*/
int devn;
int cycles;
dsp_startup();
devn=0;
dsp_new(devn,"56116");                          /* allocate new device */
for(cycles=0;cycles<1000;cycles++) dsp_exec(devn);
```

7.2.4 dsp_findmem: Get Map Index for Memory Prefix

```
dsp_findmem(device_index,memory_name,memory_map)
int device_index;                 /* DSP device index to be affected by command */
char *memory_name;                /* memory space name */
enum memory_map *memory_map;      /* return memory map type */
```

This function searches the **dt_var.mem** structure for a match to the **memory_name** string provided in the function call. If a match is found, dsp_findmem returns the memory map maintype structure value through the **memory_map** parameter and 1 as the function return value; otherwise it just returns 0 as the function return value.

For a list of memory names use the Simulator **help mem** command.

EXAMPLE

```
#include "coreaddr.h"
int devn;
enum memory_map map;
int ok;
dsp_startup();
devn=0;
dsp_new(devn,"56116");            /* Allocate structure for device 0, a 56116 */
ok=dsp_findmem(devn,"p",&map)     /* Get memory map index for "p" memory */
```

7.2.5 dsp_findpin: Get Pin Number for Pin Name

```
dsp_findpin(device_index,pin_name,pin_number)
int device_index;          /* DSP device index to be affected by command */
char *pin_name;            /* pin name */
int *pin_number;           /* return pin index */
```

This function searches the **dt_var.xpin** structures for a match to the **pin_name** string provided in the function call. If a match is found, dsp_findpin returns the pin number through the **pin_number** parameter and 1 as the function return value; otherwise it just returns 0 as the function return value.

Use the Simulator's "help pin" command to produce a list of the valid pin names.

EXAMPLE

```
int devn;
int pinnum;
int ok;
dsp_startup();
devn=0;
dsp_new(devn,"56116");              /* Allocate structure for device 0, a 56116 */
ok=dsp_findpin(devn,"reset",&pinnum);          /* Get index for "reset" pin */
```

7.2.6 dsp_findport: Get Port Number and Mask for Port Name

```
dsp_findport(device_index,port_name,port_number,mask_val)
int device_index;          /* DSP device index to be affected by command */
char *port_name;           /* port or peripheral name */
int *port_number;          /* return memory map index */
unsigned long *mask_val;   /* Pin mask for this port or peripheral name */
```

This function searches the **dt_var.xport** structure and the **dt_var.periph** structures for a match to the **port_name** string provided in the function call. If a match is found, dsp_findport returns the port number through the **port_number** parameter, the port mask value through the **mask_val** parameter, and 1 as the function return value; otherwise it just returns 0 as the function return value.

The Simulator "help port" and "help periph" commands may also be used to produce a list of valid port and peripheral information.

EXAMPLE

```
int devn;
int pnum;
unsigned long pmask;
int ok;
dsp_startup();
devn=0;
dsp_new(devn,"56116");                  /* Allocate structure for device 0, a 56116 */
ok=dsp_findport(devn,"portb",&pnum,&pmask);          /* Get info for "portb" */
```

7.2.7 dsp_findreg: Get Peripheral and Register Index for Register Name

```
dsp_findreg(device_index,reg_name,periph_number,reg_number)
int device_index;          /* DSP device index to be affected by command */
char *reg_name;            /* register name */
int *periph_number;        /* return peripheral index */
int *reg_number;           /* return register index */
```

This function searches the **dt_var.periph** structures for a match to the **reg_name** string provided in the function call. If a match is found, dsp_findreg returns the peripheral index through **periph_number**, the register number through the **reg_number** parameter and 1 as the function return value; otherwise it just returns 0 as the function return value.

You may also use the Simulator "help reg" command to obtain a list of the valid periph_num and reg_num values, and reg_val size for each register.

EXAMPLE

```
int devn;
int regnum;
int pnum;
int ok;
dsp_startup();
devn=0;
dsp_new(devn,"56116");                  /* Allocate structure for device 0, a 56116 */
ok=dsp_findreg(devn,"pc",&pnum,&regnum);    /* Get index for "pc" register */
```

7.2.8 dsp_free: Free a Device Structure

```
dsp_free(device_index)
int device_index;          /* DSP device index to be affected by command */
```

This function frees all allocated memory associated with a device structure and closes any open files associated with the device structure.

EXAMPLE

```
/* Create three new device structures, then get rid of device 2. */
dsp_startup();
dsp_new(0,"56116");              /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");              /* Allocate structure for device 1, a 56116 */
dsp_new(2,"56116");              /* Allocate structure for device 2, a 56116 */
dsp_free(1);                     /* Free structure previously allocated for device 1 */
```

7.2.9 dsp_fmem: Fill Memory Block with a Value

```
dsp_fmem(device_index,memory_map,address,block_size,value)
int device_index;                  /* DSP device index to be affected by command */
enum memory_map memory_map;        /* memory designator */
unsigned long address;             /* DSP memory start address to write */
unsigned long block_size;          /* Number of locations to write */
unsigned long *value;              /* Pointer to value to write to memory location */
```

This function writes a memory block of selected DSP memory.

The **memory_map** parameter is a memory type that selects the appropriate dt_memory structure from dt_var.mem for the selected device. These structures are describe in the **simdev.h** file which is included with the Simulator. The memory_map parameter can be obtained with the function **dsp_findmem** by using the memory name as a key. Use the Simulator **help mem** command for a list of valid memory names. The memory_map enum is memory_map_ concatenated with a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device.

If the selected memory map requires two word values, the least significant word should be at the value location and the most significant word at the value+1 location.

If the memory address selects an external memory location, the **dspl_xmwr** function will be called. The dspl_xmwr function is provided in source form in the file **simvmem.c** and can be modified to simulate special external memory characteristics.

EXAMPLE

```
/* Write 300 locations beginning at p:$200 with the value 4 */
int devn;
unsigned long address, memval, blocksize;
address=0x200L;
blocksize=300;
memval=4L;
dsp_startup();
devn=0;
dsp_new(devn,"56116");             /* Allocate structure for device 0, a 56116 */
dsp_fmem(devn,memory_map_p,address,blocksize,&memval);
```

7.2.10 dsp_init: Initialize a Single DSP Device Structure

```
dsp_init(device_index)
int device_index;          /* DSP device index to be affected by command */
```

This function initializes a device to the same state that existed following the dsp_new() call which created it. It is equivalent to performing the Simulator **RESET S** command. All memory spaces are cleared, the registers are reset, breakpoints and input/output file assignments are cleared.

EXAMPLE

```
dsp_startup();
dsp_new(0,"56116");              /* Create new dsp structure */
.
.                                /* Other Simulator commands */
.
dsp_init(0);                     /* Reinitialize device 0 */
```

7.2.11 dsp_ldmem: Load DSP Memory from OMF File

```
int dsp_ldmem(device_index,filename)
int device_index;          /* DSP device index to be affected by command */
char *filename;            /* Full pathname of OMF format file to be loaded */
```

This function loads the memory space of a specified dsp device from an object file. The file may be created as the output from the DSP MACRO ASSEMBLER, or by using the Simulator **save** command, and may be either COFF format or ".lod" format. In order to specify a COFF format file, the filename suffix must be ".cld". A filename with any other suffix is assumed to be in ".lod" format.

This is a lower level function that does not invoke the user interface modules for pathname and automatic **.lod** suffix extension. The entire pathname must be specified. The function returns 1 if the load is successful, 0 if an error occurred loading the file.

EXAMPLE

```
/* Create DSP device structures for a three device simulation. */
int devn;
int err;
dsp_startup();
for (devn=0;devn<3;devn++)
dsp_new(devn,"56116");              /* Create new dsp structures */
/* Load device 1 with a program named filter2.lod.*/
err=dsp_ldmem(1,"filter2.lod");
```

7.2.12 dsp_load: Load All DSP Structures from State File

```
int dsp_load(filename)
char *filename;                /* Full name of State File to be loaded */
```

This function loads the Simulator state of all devices from a specified Simulator state file. It is not necessary to allocate the device structures prior to calling **dsp_load**. This function does not invoke the user interface modules for pathname and automatic **.sim** suffix extension; the entire filename must be specified.

EXAMPLE

```
int err;
dsp_startup();
err=dsp_load("lunchbrk.sim");
```

7.2.13 dsp_new: Create New DSP Device Structure

```
dsp_new(device_index,device_type)
int device_index;          /* DSP device index to be affected by command */
char *device_type;         /* Name corresponding to DSP device type */
```

This function creates a new dsp structure that represents a DSP device and initializes it. It will be necessary to use the dsp_unlock() function call prior to dsp_new() if the selected device type is password protected.

EXAMPLE

```
/* Create DSP device structures for a three device simulation. */
int devn;
dsp_startup();
for (devn=0;devn<3;devn++)
dsp_new(devn,"56116");              /* Create new dsp structures */
```

7.2.14 dsp_path: Construct Filename

```
dsp_path(path_name,base_name,suffix,new_name)
char *path_name;        /* Directory pathname */
char *base_name;        /* Base filename to be appended to path_name */
char *suffix;           /* Suffix string to be appended to base_name */
char *new_name;         /* Pointer to return buffer for constructed pathname */
```

This function concatenates the user-provided pathname, base name and suffix. If the base_name begins with a pathname separator or with a device designator, the path_name will not be prepended to the base_name. If the base_name already ends with '**.**' and some suffix, the suffix will not be appended.

EXAMPLE

```
/* Load a file named filter2.lod from the current working directory for device 0. */
#include "simcom.h"
#include "simdev.h"
extern struct dev_const dv_const;     /* Simulator device structures */
char newfn[80];
dsp_startup();
dsp_new(0,"56116");                   /* Create new dsp structure */
dsp_path(dv_const.sv[0]->pathwork,"filter2","lod",newfn);
dsp_ldmem(0,newfn);                   /* Load file into dsp device 0 */
```

7.2.15 dsp_rapin: Read DSP Analog Pin State

```
int dsp_rapin(device_index,pin,retvf)
int device_index;        /* DSP device index to be affected by command */
int pin;                 /* Pin number to read */
float *retvf;            /* Pointer to floating point (single precision) return value */
```

This function reads a DSP analog device pin value. It is only valid for device pins which are defined as having analog values, such as codec output pins; other pins will return 0.0 as the analog value. The function return value will be DSP_PINVAL_L and a floating point value returned in retvf if found; or DSP_ERR if there is an error condition. Use the Simulator's "help pin" command to produce a list of the valid pin names and each pin's corresponding pin index. The device pin number can also be obtained by using the pin name as a key and calling the function **dsp_findpin**. The DSP_PINVAL return values are defined in the **simdev.h** file.

EXAMPLE

```
int devn;
int pinnum;
int err;
float aval;
dsp_startup();
devn=0;
dsp_new(devn,"56156");                  /* Allocate structure for device 0, a 56156 */
dsp_findpin(devn,"spkp",&pinnum);       /* Get pin number for pin named spkp */
err=dsp_rapin(devn,pinnum,&aval);       /* Read value of device 0 pin spkp*/
```

7.2.16 dsp_rmem: Read DSP Memory Location

```
int dsp_rmem(device_index,memory_map,address,return_value)
int device_index;                  /* DSP device index to be affected by command */
enum memory_map memory_map;        /* memory designator */
unsigned long address;             /* DSP memory address to read */
unsigned long *return_value;       /* Memory value (or values) will be returned here */
```

This function reads the contents of a selected dsp memory location and writes it to **return_value**. If the memory_map implies a two word value, the least significant word will be returned to return_value; the most significant word will be returned to the return_value+1 location. This function also returns a flag that indicates whether or not the memory location exists. It returns 1 if the location exists, 0 otherwise.

The **memory_map** parameter selects the appropriate dt_memory structure from dt_var.mem for the selected device. These structures are describe in the **simdev.h** file which is included with the Simulator. The memory_map parameter can be obtained with the function **dsp_findmem** by using the memory name as a key. Use the Simulator **help mem** command for a list of valid memory names. The memory_map enum is memory_map_ concatenated with a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device.

This function calls the function dspl_xmrd() if the address indicates an external memory location. The dspl_xmrd() function is provided in source form in the file **simvmem.c** and can be modified to simulate special external memory characteristics.

EXAMPLE

```
/* Read X memory location 100 from device 0. */
unsigned long address;
unsigned long memval;
int devn;
int ok;
dsp_startup();
devn=0;
dsp_new(devn,"56116");             /* Allocate structure for device 0, a 56116 */
address=100L;
ok=dsp_rmem(devn,memory_map_x,address,&memval);
```

7.2.17 dsp_rpin: Read DSP Pin State

```
int dsp_rpin(device_index,pin)
int device_index;          /* DSP device index to be affected by command */
int pin;                   /* Pin number to read */
```

This function reads a dsp device pin value. The return value may be DSP_PINVAL_L, DSP_PINVAL_H, DSP_PINVAL_F, DSP_PINVAL_0, or DSP_PINVAL_1 indicating low output, high output, floating, low input or high input pin state. Use the Simulator's "help pin" command to produce a list of the valid pin names and each pin's corresponding pin index. The device pin number can also be obtained by using the pin name as a key and calling the function **dsp_findpin**. The DSP_PINVAL return values are defined in the **sim-dev.h** file.

EXAMPLE

```
int devn;
int pinnum;
int pin_value;
dsp_startup();
devn=0;
dsp_new(devn,"56116");              /* Allocate structure for device 0, a 56116 */
dsp_findpin(devn,"rw",&pinnum);     /* Get pin number for pin named rw */
pin_value=dsp_rpin(devn,pinnum);    /* Read value of device 0 pin rw */
```

7.2.18 dsp_rport: Read DSP Port State

```
dsp_rport(device_index,port,data,force)
int device_index;        /* DSP device index to be affected by command */
int port;                /* Port number to read */
unsigned long *data;     /* Return port data value goes here */
unsigned long *force;    /* Return port forcing state goes here */
```

This function reads a DSP device port state. It returns two values. The value returned in the **data** parameter contains the current pin data state for all pins in the port. In the case of input pins, this is the last value written to the input pin; in the case of output pins the data state is the last data written to the port by the device. The value returned in the **force** parameter indicates which port bits are actually being driven as outputs by the device.

The **port** parameter acts as the index to the dev_var.xportval array. A list of port names and the corresponding port index can be obtained using the Simulator's "help port" and "help periph" commands. The port index can also be determined by using the port name as a key when calling **dsp_findport**.

EXAMPLE

```
int devn;
int portnum;
unsigned long portbdata, portbforce;
unsigned long portmask;
dsp_startup();
devn=0;
dsp_new(devn,"56116");              /* Allocate structure for device 0, a 56116 */
dsp_findport(devn,"portb",&portnum,&portmask);
dsp_rport(devn,portnum,&portbdata,&portbforce);
```

7.2.19 dsp_rreg: Read a DSP Device Register

```
dsp_rreg(device_index,periph_num,reg_num,reg_val)
int device_index;          /* DSP device index to be affected by command */
int periph_num;            /* DSP peripheral number */
int reg_num;               /* DSP register number */
unsigned long *reg_val;    /* Return register value goes here */
```

This function reads a selected register from the regval array in a DSP dev_periph structure. Registers which return more than one word as the register value will return the least significant word in reg_val[0].

Use the Simulator "help reg" command to obtain a list of the valid periph_num and reg_num values, and reg_val size for each register. Also, **dsp_findreg** can be used to obtain the peripheral and register number by using the register name as a key.

EXAMPLE

```
int devn;
int periph_num, reg_num;
unsigned long regval;
dsp_startup();
devn=0;
dsp_new(devn,"56116");              /* Allocate structure for device 0, a 56116 */
if (dsp_findreg(devn,"pc",&periph_num,&reg_num))
   dsp_rreg(devn,periph_num,reg_num,&regval);
```

7.2.20 dsp_save: Save All DSP Structures to State File

```
int dsp_save(filename)
char *filename;              /* Full name of State File to be saved */
```

This function saves a DSP device structure to a simulation state file. This function does not invoke the user interface functions which provide pathname and **.sim** suffix extension, so the entire filename must be specified. The function returns 1 if the save is successful, 0 if an error occurs when saving the file. This function will call the function **dspl_xmsave** as one of the steps of saving the DSP structure.

EXAMPLE

```
int ok;
dsp_startup();
dsp_new(0,"56116");              /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");              /* Allocate structure for device 1, a 56116 */
                                 /* Save device 0 and 1 to state file lunchbrk.sim. */
ok=dsp_save("lunchbrk.sim");
```

7.2.21 dsp_startup: Initialize DSP Structures

int dsp_startup();

This function initializes DSP structures. It should be called once (and only once) at the first of your program prior to any calls to **dsp_new**.

EXAMPLE

```
dsp_startup();
dsp_new(0,"56116");          /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");          /* Allocate structure for device 1, a 56116 */
```

7.2.22 dsp_unlock: Unlock Password Protected Device Type

```
dsp_unlock(device_type, password)
char *password;          /* Pointer to string containing password */
char *device_type;       /* Name corresponding to DSP device type */
```

This function provides the password for protected device types. It must be used prior to the dsp_new function call if the device type is password protected.

EXAMPLE

```
/* Create a device simulation of the password protected 56001 device */
int devn;
dsp_startup();
 dsp_unlock("56001","x51-234");      /* provide password for device */
devn=0;
  dsp_new(devn,"56001");             /* Create new dsp structures */
```

7.2.23 dsp_wapin: Write DSP Analog Pin State

```
int dsp_wapin(device_index,pin,value)
int device_index;        /* DSP device index to be affected by command */
int pin;                 /* Pin number to write*/
float value;             /* Input value for specified pin */
```

This function writes a selected DSP device pin with a single precision floating point input value. Use the Simulator's "help pin" command to produce a list of the valid pin names and each pin's corresponding pin index. The device pin number can also be obtained by using the pin name as a key and calling the function **dsp_findpin**.

EXAMPLE

```
#include "simcom.h"
#include "simdev.h"
/* Write the reset pin of device 0 with a high level. */
int devn;
int pinnum;
float pinval;
dsp_startup();
devn=0;
dsp_new(devn,"56156");              /* Allocate structure for device 0, a 56156 */
dsp_findpin(devn,"mic",&pinnum);    /* Get pin number for pin named mic */
pinval=0.709;
dsp_wapin(devn,pinnum,pinval);
```

7.2.24 dsp_wmem: Write DSP Memory Location

```
dsp_wmem(device_index,memory_map,address,value)
int device_index;                /* DSP device index to be affected by command */
enum memory_map memory_map;      /* memory designator */
unsigned long address;           /* DSP memory address to write */
unsigned long *value;            /* Pointer to value to write to memory location */
```

This function writes a selected dsp memory location.

The **memory_map** parameter is selects the appropriate dt_memory structure from dt_var.mem for the selected device. These structures are describe in the **simdev.h** file which is included with the Simulator. The memory_map parameter can be obtained with the function **dsp_findmem** by using the memory name as a key. Use the Simulator **help mem** command for a list of valid memory names. The memory_map enum is memory_map_ concatenated with a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device.

If the selected memory map requires two word values, the least significant word should be at the value location and the most significant word at the value+1 location.

If the memory address selects an external memory location, the **dspl_xmwr** function will be called. The dspl_xmwr function is provided in source form in the file **simvmem.c** and can be modified to simulate special external memory characteristics.

EXAMPLE

```
int devn;
unsigned long address, memval;
address=200L;
memval=0L;
dsp_startup();
devn=0;
dsp_new(devn,"56116");           /* Allocate structure for device 0, a 56116 */
dsp_wmem(devn,memory_map_p,address,&memval);
```

7.2.25 dsp_wpin: Write DSP Pin State

```
int dsp_wpin(device_index,pin,value)
int device_index;        /* DSP device index to be affected by command */
int pin;                 /* Pin number to write*/
int value;               /* Input value for specified pin */
```

This function writes a selected dsp device pin with a value DSP_PINVAL_L, DSP_PINVAL_H, DSP_PINVAL_F, DSP_PINVAL_N, or DSP_PINVAL_P indicating low, high, floating, negative pulse, or positive pulse. Use the Simulator's "help pin" command to produce a list of the valid pin names and each pin's corresponding pin index. The device pin number can also be obtained by using the pin name as a key and calling the function **dsp_findpin**. The DSP_PINVAL values are defined in the **simdev.h** file

EXAMPLE

```
#include "simcom.h"
#include "simdev.h"
/* Write the reset pin of device 0 with a high level. */
int devn;
int pinnum;
dsp_startup();
devn=0;
dsp_new(devn,"56116");              /* Allocate structure for device 0, a 56116 */
dsp_findpin(devn,"reset",&pinnum);  /* Get pin number for pin named reset */
dsp_wpin(devn,pinnum,DSP_PINVAL_H);
```

7.2.26 dsp_wport: Write DSP Port State

```
dsp_wport(device_index,port,mask,data,force)
int device_index;        /* DSP device index to be affected by command */
int port;                /* Port number to write */
unsigned long mask;      /* Pin mask for this port   */
unsigned long data;      /* Port data value */
unsigned long force;     /* Port forcing state */
```

This function forces data on a DSP device port from outside the device. The value supplied in the **data** parameter contains the new input data to be written to the port. The value supplied in the **force** parameter indicates which port bits are actually being driven as inputs to the device. The value supplied in the **mask** parameter specifies which pins in the port are to be affected by this write; the other pins in the port remain in their previous state.

The **port** parameter acts as the index to the dev_var.xportval array. A list of port names and the corresponding port index can be obtained using the Simulator's "help port" and "help periph" commands. The port index and mask value can also be obtained by using the port name as a key when calling **dsp_findport**.

This function call can be paired with the **dsp_rport** function to simulate a port to port connection between devices.

EXAMPLE

```
/* Write portb of device 1 from portb of device 0 */
 int portnum;
unsigned long portbdata, portbforce;
unsigned long portmask;
dsp_startup();
dsp_new(0,"56116");                /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");                /* Allocate structure for device 1, a 56116 */
dsp_findport(0,"portb",&portnum,&portmask);
dsp_rport(0,portnum,&portbdata,&portbforce)
dsp_wport(1,portnum,portmask,portbdata,portbforce)
```

7.2.27 dsp_wreg: Write a DSP Device Register

```
dsp_wreg(device_index,periph_num,reg_num,reg_val)
int device_index;          /* DSP device index to be affected by command */
int periph_num;            /* DSP peripheral number */
int reg_num;               /* DSP register number */
unsigned long *reg_val;    /* Value to be written to register */
```

This function writes a selected register in the a DSP structure.

Use the Simulator "help reg" command to obtain a list of the valid periph_num and reg_num values, and reg_val size for each register. Also, the function **dsp_findreg** can be used to obtain the peripheral and register number by using the register name as a key.

If a register requires more than one word to represent the data value the least significant word should be at reg_val, with more significant words at reg_val+1, etc.

EXAMPLE

```
int devn;
int periph_num, reg_num;
unsigned long regval;
dsp_startup();
devn=0;
dsp_new(devn,"56116");              /* Allocate structure for device 0, a 56116 */
regval=100L;
if (dsp_findreg(devn,"pc",&periph_num,&reg_num))
   dsp_wreg(devn,periph_num,reg_num,&regval);
```

7.2.28 sim_docmd: Execute Simulator User Interface Command

```
sim_docmd(device_index,command_string)
int device_index;          /* DSP device index to be affected by command */
char *command_string;      /* User interface command to be executed */
```

This function executes any Simulator command that the Simulator normally accepts from the terminal. SIMDSP normally calls sim_gtcmd() to get a valid command string from the terminal, then calls sim_docmd to execute it. The device_index determines which dsp device (in a multiple dsp simulation) is affected by the command execution. The devices are numbered 0,1,2...n-1 in an n-device system, so be very careful, for example, to use 0 for the device_index parameter in a single device system.

If the command_string begins macro execution the selected device structure **in_macro** flag will be set by sim_docmd. SIMDSP retrieves valid commands from the macro file by calling sim_gmcmd() as long as the **in_macro** flag is set. The commands are still executed by sim_docmd, whether they come from the terminal or a macro file.

Commands which initiate device cycle execution (such as **go** or **trace**) will set the device structure **sim_var.stat.executing** flag. SIMDSP executes device cycles until the **executing** flag is cleared by an execution breakpoint.

EXAMPLE

```
int devn;
dsp_startup();
devn=0;
dsp_new(devn,"56116");                /* Allocate structure for device 0, a 56116 */
sim_docmd(devn,"change pc $40");      /* Change device 0 pc register to $40 */
sim_docmd(devn,"break r p:$80");      /* Set a breakpoint for device 0 */
sim_docmd(devn,"go");                 /* Begin execution of device 0 */
```

7.2.29 sim_gmcmd: Get Command String from Macro File

```
sim_gmcmd(device_index,command_string)
int device_index;          /* DSP device index to be affected by command */
char *command_string;      /* Pointer to return buffer for command line */
```

This function reads the next Simulator command string from a macro file. The sim_docmd() function will normally determine that a command is a macro, open the macro file and set the device structure **sim_const.in_macro** flag. The sim_gmcmd() function returns the next line from the open macro file each time it is called. It will clear the **in_macro** flag at the end of macro execution or if an invalid macro command is processed. The command_string buffer should be at least 80 characters.

EXAMPLE

```
/* Execute the macro command file startup.cmd on dsp device structure 0. */
#include "simcom.h"
#include "simusr.h"
extern struct sim_const sv_const;          /* Simulator device structures */
char command_string[80];
int devn;
dsp_startup();
devn=0;
dsp_new(devn,"56116");                     /* Create new dsp structure */
sim_docmd(devn,"startup");                 /* Begin the startup macro */
while (sv_const.in_macro){
  sim_gmcmd(devn,command_string);          /* Get command string from macro file */
  sim_docmd(devn,command_string);          /* Execute command string */
  }
```

7.2.30 sim_gtcmd: Get Command String from Terminal

```
sim_gtcmd(device_index,command_string)
int device_index;          /* DSP device index to be affected by command */
char *command_string;      /* Pointer to return buffer for command line */
```

This function gets the next command string from the terminal in an interactive mode. The command line editing, command expansion and on-line help functions are invoked by this terminal command input function. The command string is fully checked for errors prior to returning. The command_string buffer should be at least 80 characters.

EXAMPLE

```
/* Get and execute Simulator commands for device 0 until a go type command is */
/* entered. */
#include "simcom.h"
#include "simusr.h"
extern struct sim_const sv_const;              /* Simulator device structures */
char command_string[80];
int devn;
dsp_startup();
devn=0;
dsp_new(devn,"56116");                         /* Create new dsp structure */
while (!sv_const.sv[devn]->stat.executing){          /* Check for go */
  sim_gtcmd(devn,command_string);                    /* Get command */
  sim_docmd(devn,command_string);                    /* Execute command */
 }
```

7.3 SIMULATOR EXTERNAL MEMORY FUNCTIONS

The following sections describe functions which are provided in source code form in the Simulator package in the file **simvmem.c**. These functions define all the operations associated with reading, writing or storing dsp external memory locations. The Simulator memory allocation function is also included in this module since the representation of external memory is implemented with a virtual memory technique that is integrated with the memory allocation service. The external memory functions, with the exception of **dsp_alloc**, would not normally be called directly from the user's code. They are referenced from other Simulator functions, such as **dsp_load** or **dsp_rmem**, described in Section 7.2. The following is a reference list of the external memory functions:

dsp_alloc(num_bytes);	Allocate Simulator Program Memory
dspl_xmend(devn,map);	End DSP External Memory access
dspl_xmfree(devn);	Free DSP Device External Memory
dspl_xminit(devn);	Initialize DSP Device External Memory
dspl_xmload(devn,fp);	Load DSP External Memory from State File
dspl_xmnew(devn);	Create New External Memory Structure
dspl_xmrd(devn,map,add,val,fetch);	Read DSP External Memory Location to val
dspl_xmsave(devn,fp);	Save DSP External Memory to State File
dspl_xmstart(devn,map);	Start DSP External Memory access
dspl_xmwr(devn,map,add,val,store);	Write DSP External Memory with val

The external memory access functions are provided in source form so that the external memory map attributes can be customized. This is especially useful for multiple dsp simulations in which complex configurations such as dual-port memory may be required. The functions in **simvmem.c** simulate the entire external memory space of up all dsp devices.

7.3.1 dsp_alloc: Allocate Simulator Program Memory

```
void *dsp_alloc(numbytes)
unsigned int numbytes;      /* Size of memory block needed in bytes */
```

This function must return a character pointer to the requested number of bytes of memory. It is not necessary for the memory to be cleared. A simple version could just call malloc(). The Simulator will not recover if the dsp_alloc() call fails, so an exit() must occur within dsp_alloc() if the requested memory cannot be allocated.

EXAMPLE

```
/* Allocate memory for a new structure. */

void *dsp_alloc();
struct new_struct{
    char buf[1000];
    int bufindex;
  } *newp;
newp=(struct new_struct *) dsp_alloc(sizeof(struct new_struct));
```

7.3.2 dspl_xmend: End DSP External Memory Access

```
dspl_xmend(device_index,memory_map)
int device_index;                  /* DSP device index to be affected by command */
enum memory_map memory_map;    /* memory designator */
```

The core simulation calls this function during the last clock cycle of each external memory access. The memory map parameter will be a memory designator as returned by **dsp_findmem**. Use the Simulator **help mem** command for a list of valid memory names. The memory_map enum is memory_map_ concatenated with a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device.

7.3.3 dspl_xmfree: Free DSP Device External Memory

```
dspl_xmfree(device_index)
int device_index;              /* DSP device index to be affected by command */
```

This function must free any memory that has been allocated to represent the external memory space of a selected device. Note that this function should not be called directly by the user's code. It is called as one of the steps in freeing an entire device structure by dsp_free().

EXAMPLE

```
/* Free external memory of dsp device structure 0. */
int devn;
devn=0;
dspl_xmfree(devn);
```

7.3.4 dspl_xminit: Initialize DSP Device External Memory

```
dspl_xminit(device_index)
int device_index;          /* DSP device index to be affected by command */
```

This function must initialize the values in any structures used to represent the external memory of a dsp device. The Simulator commands **reset s** and **load s** will call dspl_xminit() in the processes of initializing or reloading the Simulator state.

EXAMPLE

```
/* Initializing external memory for device 1 */
int devn;
devn=1;
dspl_xminit(devn);
```

7.3.5 dspl_xmload: Load DSP External Memory from State File

```
int dspl_xmload(device_index,fp)
int device_index;          /* DSP device index to be affected by command */
FILE *fp;                  /* Pointer to file opened in text read mode ("r") */
```

This function must restore external memory from a Simulator state file. Note that this function should not be called directly by the user's code. The dspl_xmload() call is the last step of the dsp_load() function which loads a Simulator state file. The file pointer provided to dspl_xmload will have been opened with fp=fopen(filename,"r") and the remainder of the Simulator state will have already been restored from the state file. The steps used to restore the external memory should complement the steps used to save external memory in the dspl_xmsave() function. The return value of dspl_xmload() should be 1 if successful, 0 if an error occurred. The dsp_load() function will close the file following the dspl_xmload() call.

EXAMPLE

```
/* Call of dspl_xmload() from dsp_load() */
int status;
FILE *fp;
fp=fopen(filename,"r");
./* Loading of other Simulator state structures */
.
status=dspl_xmload(devn,fp);
```

7.3.6 dspl_xmnew: Create New External Memory Structure

```
dspl_xmnew(device_index)
int device_index;              /* DSP device index to be affected by command */
```

This function must create and initialize the external memory for a device. Note that this function should not be called directly by the user's code. The dsp_new() function calls dspl_xmnew() as part of the process of creating a new dsp device structure.

EXAMPLE

```
/* Call to dspl_xmnew() from dsp_new() */
dspl_xmnew(devn);
```

7.3.7 dspl_xmrd: Read DSP External Memory Location

```
int dspl_xmrd(device_index,mem_map,address,return_value,fetch)
int device_index;                /* DSP device index to be affected by command */
enum memory_map memory_map;      /* memory designator */
unsigned long address;           /* DSP memory address to read */
unsigned long *return_value;     /* Memory value will be returned here */
int fetch;                       /* Flag indicating that a dsp fetch is in progress */
```

This function must return the value of a dsp device's external memory location. The Simulator calls dspl_xmrd() when a dsp device reads an external memory location, or when the Simulator user interface reads the location for display purposes. This function also returns a flag value of 1 if the memory location exists, 0 if it doesn't exist. The **fetch** parameter indicates to **dspl_xmrd()** whether or not the read is being executed by the dsp device. If fetch=1, the dsp device is fetching the memory location during execution of a device cycle. If fetch=0, **dspl_xmrd()** is being called from some other source not associated with device cycle execution (for example, from the memory display routines). Although the fetch parameter is not used in the version of **dspl_xmrd()** provided in the file **simvmem.c**, it is provided to enable special processing that should only occur when the device cycle simulation is taking place. The memory map parameter will be a value representing the memory space being accessed. Use the Simulator **help mem** command for a list of valid memory names. The memory_map enum is memory_map_ concatenated with a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device.

EXAMPLE

```
/* Read "pe" memory location $5000 of device 2 */

unsigned long address;
int devindex;
unsigned long retval;
int ok;
int fetch;
address=0x5000L;
devindex=2;
fetch=0;
ok= dspl_xmrd(devindex,memory_map_pe,address,&retval,fetch);
```

7.3.8 dspl_xmsave: Save DSP External Memory to State File

```
int dspl_xmsave(device_index,fp)
int device_index;          /* DSP device index to be affected by command */
FILE *fp;                  /* Pointer to file opened in write mode */
```

This function must save the external memory state to a Simulator state file. The dspl_xmsave() call is the last step of the dsp_save() function which saves a Simulator state file. The file pointer provided to dspl_xmsave will have been opened with fp=fopen(filename,"w+") and the remainder of the Simulator state will have already been saved to the state file. The return value of dspl_xmsave() should be 1 if successful, 0 if an error occurred. The dsp_save() function will close the file following the dspl_xmsave() call.

EXAMPLE

```
/* Call of dspl_xmsave() from dsp_save() */
int status;
FILE *fp;
fp=fopen(filename,"w+");
./* Saving of other Simulator state structures */
.
status=dspl_xmsave(devindex,fp);
```

7.3.9 dspl_xmstart: Start DSP External Memory Access

```
dspl_xmstart(device_index,memory_map)
int device_index;                   /* DSP device index to be affected by command */
enum memory_map memory_map;   /* memory designator */
```

The core simulation calls this function during the first clock cycle of each external memory access. The memory map parameter will be a value as returned by **dsp_findmem**. Use the Simulator **help mem** command for a list of valid memory names. The memory_map enum is memory_map_ concatenated with a valid memory name. As an example, memory_map_pa refers to off chip pa memory on the 96002 device.

7.3.10 dspl_xmwr: Write DSP External Memory Location

```
int dspl_xmwr(device_index,mem_map,address,value,store)
int device_index;                 /* DSP device index to be affected by command */
enum memory_map memory_map;       /* memory designator */
unsigned long address;            /* DSP memory address to write */
unsigned long value;              /* Value to be written to memory location */
int store;                        /* Flag indicating that a device store is in effect */
```

This function must store a value to a dsp device's external memory location. The Simulator calls dspl_xmwr() when a dsp device writes an external memory location, or when the Simulator user interface alters the location. The **store** parameter will indicate if the reference is from the dsp device (store=1) during simulation of device cycle execution, or some other source (store=0) not related to device cycle execution. For example, the CHANGE memory Simulator command will set the parameter store to 0. The store parameter is not used in the **dspl_xmwr** function provided in the file **simvmem.c**, but is available to the user if modifications are made to the **simvmem.c** file for special external memory processing. The memory map parameter will be a value representing the memory space being accessed. Use the Simulator's "help mem" command to obtain a list of the valid memory space prefixes.

EXAMPLE

```
/* Write value of 3 to "xe" memory location 5 of device 2 */

unsigned long address;
int devindex;
int ok;
unsigned long newval;
int store;
address=5L;
devindex=2;
newval=3L;
store=0;
ok=     dspl_xmwr(devindex,memory_map_xe,address,newval,store);
```

7.4 SIMULATOR SCREEN MANAGEMENT FUNCTIONS

The following sections describe functions which are provided in source code form in the Simulator package in the file **scrmgr.c**. These functions define all the operations associated with Simulator terminal I/O. The code includes conditionally compiled sections for MSDOS, UNIX, and VMS. The code is provided to allow customization of the Simulator terminal I/O for a particular environment. The user may, for example, wish to redefine the control characters used by the Simulator so that they map to some particular terminal.

The following is a quick reference list of the Simulator screen management functions:

simw_ceol();	Clear to end of line
simw_ctrlbr();	Check for CTRL-C signal
simw_cursor(line,column);	Move cursor to specified line, column
simw_endwin();	End the Simulator display
simw_getch();	Non-translated keyboard input
simw_gkey();	Translated keyboard input
simw_putc(c);	Output character to terminal
simw_puts(line,column,text,flag);	Output string to terminal at line and column
simw_redo(device);	Repaint screen with output from device
simw_redraw(count);	Redraw screen after scrolling count
simw_refresh();	Screen update after buffering output
simw_scrnest();	Nest output buffering another level
simw_unnest();	Pop output buffering one level
simw_winit();	Initialize window parameters
simw_wscr(string,commandflag);	Write string and perform logging functions

7.4.1 simw_ceol: Clear to End of Line

simw_ceol()

This function must clear the display from the current column to the end of line, then return the cursor to the previous position.

7.4.2 simw_ctrlbr: Check for CTRL-C Signal

simw_ctrlbr()

This function must check for the occurrence of a CTRL-C signal from the terminal. If the CTRL-C signal occurs, it sets a flag for the active breakpoint dsp (defined by **sv_const.breakdev**). It returns the sim_var.stat.CTRLBR flag for the current device. This allows the program to select the device that will halt in response to the CTRL-C signal from the keyboard in a multiple device simulation.

7.4.3 simw_cursor: Move Cursor to Specified Line and Column

simw_cursor(line,column)

This function must move the cursor to the specified line and column and update the sim_const.curline and sim_const.curclm variables.

7.4.4 simw_endwin: End Simulator Window

simw_endwin()

This function is normally called when returning to the operating system level from the Simulator. It must terminate any special processing associated with terminal I/O for the Simulator and clear the display.

7.4.5 simw_getch: Non-translated Keyboard Input

simw_getch()

This function gets a single character in a non-translated mode from the terminal. It is not used much by the Simulator - only when returning from the execution of the **system** command prior to the time when the Simulator's special terminal I/O processing is reinitialized.

7.4.6 simw_gkey: Translated Keyboard Input

simw_gkey()

This function gets a keystroke from the terminal and maps it to one of the accepted internal codes used by the Simulator. The internal codes are defined in **simusr.h**. This function should not output the character to the terminal. This function is a good candidate for modification if you want to change the set of input control characters used by the Simulator.

7.4.7 simw_putc: Output Character to Terminal

```
simw_putc(c)
char c;
```

This function outputs the character in the variable **c** at the current cursor and column position. It advances and updates the sim_const.curclm variable. This function is not used often by the Simulator, and it is not very time critical when it is used, so the Simulator implementation is just to call simw_puts() after creating a temporary string from the character **c**.

7.4.8 simw_puts: Output String to Terminal

```
simw_puts(line,column,text,flag)
int line;          /* Move cursor to this line for output */
int column;        /* Move cursor to this column for output */
char *text;        /* Text string to be output */
int flag;          /* 0=non-bold, 1=bold on/off by {}, 2=all bold */
```

This function outputs a string to the terminal at the specified line and column. Highlighting of output can be enabled either by setting the **flag** parameter to 2 or by enclosing text in curly braces and setting the **flag** parameter to 1.

7.4.9 simw_redo: Repaint Screen With Output From Device

```
simw_redo(device)
int device;        /* Use screen buffer from this device to repaint screen */
```

This function repaints the screen from a device screen buffer. It is normally only called when re-entering the Simulator following a **system** command, after loading the device state with the **load s filename** command, or after switching devices in a multiple device simulation with the **device** command.

7.4.10 simw_redraw: Redraw Screen After Scroll Count

```
simw_redraw(count)
int count;              /* Number of lines to scroll before repainting the screen */
```

This function scrolls up or down **count** lines in the display buffer, then redisplays the text in the buffer at that position. This function only displays the text that is in the scrolling portion of the display.

7.4.11 simw_refresh: Screen Update After Buffering Output

```
simw_refresh()
```

The Simulator buffers screen output in implementations other than MSDOS in order to decrease the time spent repainting the screen. This provides a fixed display effect for consecutive trace commands. The simw_refresh() function will take care of refreshing the screen following buffering of screen output. It also resets the sim_const.scrnest variable to 0 to coincide with the non-buffered status of the screen following the refresh.

7.4.12 simw_scrnest: Increase Screen Buffering One Level

```
simw_scrnest()
```

This function increments a counter to signify the screen output buffering level. The companion **simw_unnest()** and **simw_refresh()** functions provide the output buffering operations for the Simulator. The sim_const.scrnest variable is incremented each time this function is called.

7.4.13 simw_unnest: Decrease Screen Buffering One Level

```
simw_unnest()
```

This function decrements the sim_const.scrnest variable each time it is called. If the screen buffering level drops below one, simw_unnest() will call simw_refresh() to update the screen.

7.4.14 simw_winit: Initialize Window Parameters

```
simw_winit()
```

This function initializes any screen or keyboard parameters that are required for the Simulator terminal I/O environment. It is called whenever the Simulator is entered from the operating system level, which includes the initial Simulator entry and re- entry following the **system** command.

7.4.15 simw_wscr: Write String and Perform Logging

```
simw_wscr(text,command_flag)
char *text;              /* Text string to write to screen */
int command_flag;        /* Flag 1=string is a command, 0= not a command */
```

This function outputs the string to the terminal above the command line after scrolling the display up one line. It also takes care of writing the text string to the proper log files specified by the Simulator **log s** or **log c** commands.

7.5 NON-DISPLAY SIMULATOR

The Simulator package contains object libraries which support both display and non- display versions of the Simulator. The library **nwsim** contains functions available to the non-display version of the Simulator. The library **wwsim** contains functions that may only be used in a display version of the Simulator. For each device type there are also display and non-display device-specific libraries named **wwxxxxx** and **nwxxxxx** where the **xxxxx** is the device number (see Chapter 8, C Object Libraries).

The source code contained in **snwdsp.c** can be linked with the nwxxxxx and nwsim libraries to create a non-display version of the Simulator. Elimination of the user interface functions cuts the code size of the Simulator almost in half. However, all of the functions listed in Section 7.4 and **sim_docmd()**, **sim_gmcmd()** and **sim_gtcmd()** described in Section 7.2, are sacrificed.

The remainder of the functions in Section 7.2 and all of the functions in Section 7.3 are available in the non-display Simulator libraries.

Some major features of the Simulator are eliminated by the loss of the sim_docmd() function. In particular, there are no low-level entry points provided to set a breakpoint or to assign input or output files to DSP peripheral functions. However, the basic functions required to create a device, load a program, execute the code, and test or modify device registers are all still available. In addition, the dsp_save() function provides the capability to save the state of the non-display version. The state file can later be reloaded by a display version of the Simulator for visual examination of the registers and memory contents.

The following sections cover several topics that concern the non-display version of the Simulator. Section 7.5.1 deals with creating a new device. Section 7.5.2 describes how to load a program or state file. Section 7.5.3 describes how to execute device cycles. Section 7.5.4 describes how to test breakpoint conditions.

7.5.1 Creating a New Device

The **simcom.h** file defines the maximum number of DSP devices in the constant DSP_MAXDEVICES. A new device can be created and numbered from 0 to DSP_MAXDEVICES-1. The structures are allocated by calls to the **dsp_new()** function described in Section 7.2.13.

EXAMPLE

The following C source code illustrates the steps necessary to create 3 DSP devices.

```
dsp_startup();
dsp_new(0,"56116");          /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");          /* Allocate structure for device 1, a 56116 */
dsp_new(2,"56116");          /* Allocate structure for device 2, a 56116 */
```

7.5.2 Loading Program Code or Device State

The display version of the Simulator provides the high level sim_docmd() function interface. It allows the user to simply execute the high level **load** or **load s** Simulator commands to load program code or a Simulator state file. The non-display version of the Simulator makes use of the lower level function calls, **dsp_ldmem()** and **dsp_load()**, to accomplish the same results. They are described in Section 7.2. The major difference from their high-level counterparts is that no file-name expansion is provided in the lower level calls.

The program code loaded by the **dsp_ldmem()** function may be any COFF format or OMF format file. The OMF format is created as the output of versions of the macro assembler prior to release 4.0 and of the Simulator **save** command. It is described in Chapter 6. The COFF format files are the output of the macro assembler beginning with release 4.0, or those saved by the Simulator **save** command with the suffix ".cld".

The Simulator state loaded by the **dsp_load()** function may have previously been saved by a display or non-display version of the Simulator. The formats are the same.

The **dsp_save()** function is provided as a low-level entry point that saves the Simulator state for a non-display version of the Simulator. It is the same function that is called during execution of the high level **save s** command, which is only available in the display version. The only limitation is that the full save filename must be specified. No automatic expansion is done for the working path or filename suffix as in the higher level Simulator calls. The **dsp_save()** function is described in Section 7.2.

7.5.3 Executing Device Cycles

After creating a new device - as described in Section 7.5.1 - and loading a program or state file - as described in Section 7.5.2 - the Simulator is ready to execute the program code.

Execution will begin at the start address specified in the load file, or continue from the previous location in a Simulator state file. The user's code may select a new execution address by writing register "pc" using the **dsp_wreg** function.

The Simulator will advance the device state by one clock cycle each time the **dsp_exec** function is called. The device pin states are updated each clock cycle, and can be examined or changed using the **dsp_rpin**, **dsp_wpin**, **dsp_rport**, or **dsp_wport** functions.

7.5.4 Testing Breakpoint Conditions

The display version of the Simulator provides a way to specify breakpoint conditions that are evaluated each time **dsp_exec** is called. If the breakpoint condition is met, the Simulator displays the enabled registers and clears the device structure **sim_var.stat.executing** flag (assuming the breakpoint action is **halt**).

The non-display Simulator does not provide a way to specify breakpoint conditions. It is up to the user's code to examine device registers or memory conditions and decide whether or not to continue cycle execution. The device registers and memory can be examined using the **dsp_rreg** and **dsp_rmem** functions. The example program **snwdsp.c** simply checks the Simulator cycle counter for device 0 and terminates execution after some number of cycles.

Another variable that may be particularly useful in breakpoint testing is dev_var.flg_stat. It maintains bit flags which signal end-of-instruction (**DSP_GEOI**), end of repeat cycle (**DSP_GEOR**), and illegal opcode (**DSP_GILLEG**). The bit flag definitions are defined in **simdev.h**.

7.6 MULTIPLE DEVICE SIMULATION

The SIMDSP Simulator initially simulates a single dsp device; but additional devices can be created using the **device** command. Device to device pin connections or device to device memory map connections can be specified with the Simulator **input** command. The following sections describe some details about the way the Simulator handles multiple device simulation. Section 7.6.1 describes the required steps which allocate and initialize multiple dsp structures. Section 7.6.2 describes the method of interleaving device execution in order to maintain multiple device synchronization. Section 7.6.3 describes simulation of the external memory space of the dsp devices. Section 7.6.4 describes multiple device pin connections. Section 7.6.5 describes display of device output in the multiple device environment.

7.6.1 Allocation and Initialization of Multiple Devices

Most of the higher level Simulator functions require a device index as one of the parameters. The Simulator uses the device index to select a previously allocated DSP structure. The DSP structures are allocated dynamically by calling the **dsp_new** function for each device. The device type is also selected in the dsp_new function call. In the display version of the Simulator, the **device** command handles the details of calling **dsp_new**. The proper sequence of instructions necessary to allocate three DSP devices is shown below.

```
dsp_startup();
dsp_new(0,"56116");          /* Allocate structure for device 0, a 56116 */
dsp_new(1,"56116");          /* Allocate structure for device 1, a 56116 */
dsp_new(2,"56116");          /* Allocate structure for device 2, a 56116 */
```

7.6.2 Interleaving Multiple DSP Simulations

The **dsp_exec** function executes a single DSP clock cycle and updates the selected DSP device structure. In order to simulate simultaneous multiple DSP execution, dsp_exec should be called for every device before proceeding to the next clock cycle. The **simdsp** Simulator executes a single clock cycle for each active dsp device, then halts if any active device has cleared its sim_var.stat.executing flag. It allows Simulator commands, such as register or memory modifications, on the viewed device until a command sets the executing flag again. The device which causes a breakpoint becomes the viewed device by default, but the viewed device can be changed with the **device** command without changing the status of any device. A particular device can be halted by setting the **CTRLBR** flag in its sim_var.stat structure. This has the same effect as typing CTRL-C at the keyboard while a device is running. It breaks device execution at the end of the current instruction. Note that it is not mandatory to wait for the sim_var.stat.executing flag to be set to begin device execution, or to halt if the executing flag is clear. These are just convenience features for the Simulator user interface. Device cycle execution can be advanced in single cycle increments at any time by calling **dsp_exec**.

7.6.3 External Memory Definition

The Simulator package contains the C source file **simvmem.c** which contains all the external memory access functions used by the SIMDSP Simulator. These functions are described in detail in Section 7.3. The functions, as written, will automatically simulate the entire external memory space of all DSP devices, assuming that the operating system can allocate enough memory to store the device structures for the devices.

The user may wish to modify the simvmem.c functions in order to define special external memory configurations. The functions can be modified, for example, to simulate the response of dual-port RAM or special memory-mapped peripherals. Another good reason to modify the external memory functions is to increase the speed of the simulation. If the user's simulation only requires some minimum amount of external memory, then the virtual memory management functions provided with the Simulator are probably overkill.

7.6.4 Multiple DSP Pin Interconnections

The **dsp_exec** function will automatically update the DSP device pin states by one clock cycle change each time it is called. The display version of the Simulator will also retrieve or send data to assigned I/O files as defined by the **input** and **output** commands. The **input** command supplies a method of connecting device pins back to other device pins on the same device as well as to device pins on another device.

The device pin states for any device can be examined or written using the **dsp_rpin** and **dsp_wpin** functions described in Section 7.2. Simulation of pin to pin connection simply requires reading the state of the output pin each cycle with **dsp_rpin** and writing it to the input pin with **dsp_wpin**. A bidirectional pin connection requires reading and writing both pins. The Simulator maintains separate buffers for input and output data for each pin, so there is no problem writing a pin, even if it is defined as an output. The input value will be stored, but will only be used if the pin is subsequently reconfigured as an input.

An entire port state can be read or written using the **dsp_rport** and **dsp_wport** functions described in Section 7.2. The port and pin states are derived from the same storage variables in the device structure. The dsp_rport, dsp_wport, dsp_rpin, and dsp_wpin functions just provide a convenient method of retrieving the data from this structure.

7.6.5 Multiple DSP Simulator Display

The Simulator display functions are contained in the source file **scrmgr.c** in the Simulator package. This code supports a virtual screen for each simulated dsp.

The supplied display code uses a single window. The lines above the command line form a scrollable region in which session output is displayed. The command line, error line and help line are the three bottom lines of the display. Each allocated device contains screen buffer memory which saves the previous 100 lines of output which is written to a device's scroll region. The terminal screen update is inhibited unless the sim_const.viewdev value matches the device index, but the output is always placed in the device's screen buffer. The screen can be completely refreshed from a selected device screen buffer by executing the **simw_redo** function.

The **device** command allows the user to switch the displayed device. When it switches to a new device, it refreshes the entire screen from the device's display buffer.

7.7 RESERVED FUNCTION NAMES

The public function names used in the Simulator all begin with the prefixes **dsp** or **sim**. Functions which begin with **sim** are only available when a display version of the Simulator is created. Functions which begin with **dsp** are available to both display and non display versions. The screen management functions all begin with **simw**_. In general, functions which begin with **dsp**_ or **sim**_ are higher level functions available for direct reference from the user's code; those beginning with **dspl**_ or **siml**_ are meant only for internal use by the Simulator. The higher level functions and the screen management functions are documented in Sections 7.2, 7.3, and 7.4. The public function names are listed in the file named global.sym which is included with the distribution.

7.8 SIMULATOR GLOBAL VARIABLES

In order to reduce conflicts with user variable names, the Simulator global variables have been grouped together into several large structures. In general, the structure names beginning with **s** are used defined in **simusr.h** and are only used in the display version of the Simulator; while those beginning with **d** are defined in **simdev.h** and are used by both the display and non-display versions of the Simulator. The prefixes **st**_ and **dt**_ are used for structure names of device-type structures, that is structures which must be defined for each device type. The prefixes **sim**_ and **dev**_ are used for structure names of general device or simulation structures.

Global variable names may have a prefix **dx**_, **dv**_, **sx**_, or **sv**_. The prefix **dx**_ is used for variables of **dt**_ structures. The prefix **dv**_ is used for variables of **dev**_ structures. The prefix **sx**_ is used for variables of **st**_ structures. The prefix **sv**_ is used for variables of **sev**_ structures. A list of Simulator global variables is included in the distribution file named global.sym.

7.9 MODIFICATION OF SIMULATOR GLOBAL STRUCTURES

The source file **simglob.c**, which is included in the Simulator package, contains the global structures **sv_const** and **dv_const**. There are some useful modifications, described below, that can be made to the constant definitions at the beginning of **simglob.c**. The **simglob.c** module must then be recompiled and relinked using the make file provided with the Simulator package. In addition to these, there may be device-specific modifications that can be made to the Simulator (see Chapter 8, Modification of Device Global Structures).

DSP_MAXDEVICES This define constant determines the maximum number of devices that can be allocated using the Simulator's **device** command. As a default it is set to 32.

DSP_CMDSZ This define constant determines the size of the previous command stack. The Simulator commands are stored in the stack and can be reviewed using the **ctrl-f** and **ctrl-b** key entries. As a default the previous command stack size is set to 10.

DSP_HISTSZ This define constant determines the size of the execution history buffer, which stores device instructions as the Simulator executes. The buffer is used by the Simulator **history** command, and has a default size that can save 32 instruction words.

DSP_WINSZ This define constant determines the size of the screen buffer that is maintained and displayed by the **scrmgr.c** functions. It specifies the number of display lines that will be allocated for each device as they are created with the Simulator **device** command. The user can use the **ctrl-u**, **ctrl-t**, **ctrl-v**, and **ctrl-d** key sequences to review display lines that have scrolled off the screen. This constant should not be set to a value smaller than the number of lines in the display window.

Chapter 8
DEVICE-DEPENDENT INFORMATION

8.1 INTRODUCTION

The preceding chapters describe Simulator information that common to all of the dsp Simulator programs. This chapter describes information relating to the Simulator that is unique for each dsp family Simulator. Each family Simulator contains several device types that can be selected for simulation. Additional documentation for each device type is available using the Simulator help command.

8.2 SIMULATOR NAMES

There are several Simulators covered by this document. The Simulator names are, in general, the prefix "sim" followed by the family device number. As examples, Simulator names are sim56000 for devices in the DSP56000 family, sim96000 for devices in the DSP96000 family, and sim56100 for devices in the DSP56100 family.

8.3 DEVICE NAMES

The Simulator **device** command, with no additional parameters, will provide a list of device types available in the Simulator. These device types can be used as a parameter of the **device** command, or as a parameter of the **dsp_ new** function call, to create the device structures necessary for simulation of the selected device. There may be additional non-disclosed devices available to the Simulator. It is necessary to specify an associated password using the Simulator unlock command before a non-disclosed device will appear in the list of device types and is available for selection. The **dsp_unlock** function call provides similar functionality to the unlock command.

8.4 C OBJECT LIBRARIES

The Simulator software includes object libraries that enable you to rebuild the Simulator. A separate set of display and non-display libraries are provided, so you have the option of generating a non-display version of the Simulator. The libraries with the prefix "ww", followed by the family device number, contain the display version of the object modules. The libraries with the prefix "nw", followed by the family device number, contain the non-dis-

play versions of the same object modules. In addition, the libraries with the prefix "cm", followed by the family device number, contain object modules required by both the display and non-display versions of the Simulator. As an example, relinking the display version of the sim56100 Simulator requires libraries **ww56100** and **cm56100**; a non-display version of the Simulator requires the libraries **nw56100** and **cm56100**.

8.5 OPERATING MODES

The **reset** command allows specification of the device operating mode. The Simulator on-line help command "help mode" can be used to list the valid operating modes for the selected device.

8.6 PERIPHERAL I/O

The device peripherals may have special I/O capabilities enabled by the Simulator **input** and **output** commands. As an example, the SCI peripheral of the 56000 device will accept the strings "idle" and "break" from the attached input file. The special I/O capabilities are documented in the on-line help for each peripheral. Use the command **help** followed by the peripheral name to obtain help for the specified peripheral. Note that these special modes just supplement the pin-data i/o capability that exists for all device pins. For a description of pin-data i/o see I/O File Pin or Pin Group Data on page 3-7

8.7 MODIFICATION OF DEVICE GLOBAL STRUCTURES

The device types available to the simulation are defined in the source module named with the prefix "dsp" followed by the device family name. As an example, the file **dsp56100.c** contains the global structures **dx_56116** and **sx_56116**, which define device-specific information about the DSP56116 device in the 56100 family. You may wish to modify this module to define a new device type that can then be created by the Simulator's **device** command. The basic idea is to create a new device type and definition by modifying a previous definition in the "dsp" file. As an example, using the 56116 device in the file dsp56100.c, the procedure would be:

Make a copy of dsp56100.c and name it something else. Modify the **makefile** file to include this new module name for compilation and linking in the same manner that makefile handles the **dsp56116**.

In the new file, rename the dx_56116 structure to some name other than dx_56116, and put a pointer to this new structure in the **dx_all** array in the file **simglob.c**.

In the new **dt_var** structure, change the device type name to some name other than "56116" - this is in the first member of the dt_var structure

In the new file, change the **sx_56116** structure to some name other than sx_56116, and put a pointer to this new structure in the **sx_all** array in the file **simglob.c**.

After the above steps are completed, lower level structures and define constants in the new module can be modified to change such parameters as on-chip memory size, number of peripherals, or number and names of pins. Continuing with the 56116 example, below is a list of the lower level structures and define constants associated with the 56116 that may be changed in the new file:

DSP_PI_SIZE_116 This define constant determines the size of the on-chip program memory.

DSP_PR_SIZE_116 This define constant determines the size of the on-chip bootstrap rom memory.

DSP_XI_SIZE_116 This define constant determines the size of the on-chip X data memory.

DSP_XP_SIZE_116 This define constant determines the size of the on-chip X memory-mapped peripheral register space.

dx_periph_56116 This structure can be modified to add peripherals to or remove peripherals from the newly defined device. If you modify this structure, you must also modify the **sx_periph_56116** structure in a similar manner. The file **portb100.c** is provided in source form with the Simulator package as a model to be used when creating new peripheral structures.

bootrom This is the default initialization data for the on-chip bootstrap rom. It is loaded at start-up and in response to the **reset s** command. You can also change the contents of the bootstrap rom with the Simulator **asm**, **load**, and **change** commands.

xpin This structure determines the names assigned to device pins, as well as the order in which the pins are displayed in output pin lists. It also provides a cross-reference from the pin name to the physical bit and storage port in which the Simulator maintains the pin data. The portindex cross-reference is an offset to the proper **dev_xpval** structure from the **xportval** pointer in the **dev_var** structure. Each pin has a primary name and a possible alternate peripheral function pin name. You may modify the names or delete or add **dt_xpin** structures from this array, but do not change the portindex and pinmask members of the dt_xpin structures.

mem_56116 This array of **dt_memory** structures can be modified to change parameters associated with the DSP56116 memory attributes. The name member is the memory name used by the Simulator commands to reference the memory space. The memsize member determines the size of arrays allocated for on-chip memory. The memattr member determines whether the memory is located on or off chip and whether it is ram or rom. The romval pointer can point to initialization data for the memory space. If it is NULL, the memory space will be initialized to zero at start-up and in response to the **reset s** command. Although you may change the memory names, memory size, memory attributes, and initialization data, do not add or delete dt_memory structures from the mem_56116 array.

pval This array of **dt_xpidata** structures is used by the Simulator to initialize the input pin data in the dev_var.xportval structures at Simulator start-up and in response to the **reset s** command.

xports This array of **dt_xpid** structures determines port names that can be used in the Simulator input and output commands. These names are in addition to the peripheral names that are specified in the individual peripheral modules. The port names are just a convenient way to specify a subgroup of pins within a single physical port for input and output operations.

dx_56116 This structure is mostly a conglomeration of the other substructures defined within the module for this device type. The only member of this structure that should be modified is **devname**, which will be used to specify the new device type in the Simulator **device** command

mem_dispfw56116 This structure provides display field width information for the different memory spaces defined in the **mem_56116** defined previously.

hlp_56116 This structure provides the help pointers that will be used by the Simulator when help is requested for this device type.

sx_periph_56116 This structure points to display information for the registers of each peripheral; the actual display information is defined locally in each peripheral module. The file **portb100.c** is provided in source form with the Simulator package as a model to be used when creating new peripheral structures.

Chapter 9
GRAPHICAL USER INTERFACE

9.1 INTRODUCTION

9.1.1 Target Audience

This chapter is intended to be read by those who will use the Graphical User Interface (GUI) version of the development system. It describes the use of the GUI for the DSP Simulator. Each operation is described, with illustrations of the windows, dialog boxes, and expected outputs which result from the operation. Important features are indicated on each illustration.

9.1.2 Host System Requirements

The graphic interface version of the DSP Simulator requires the following minimum system configuration:

9.1.2.1 SUN workstation

Any SPARCstation 2, with at least xxMb of free disk space.

9.1.2.2 Hewlett Packard workstation

Any HP workstation, with at least xxMb of free disk space.

9.1.2.3 IBM PC

A 80386 system or better with a math coprocessor, minimum 8Mb ram, color SVGA display at 800 by 600 resolution or better, at least xxMb free disk space. Preferred configuration is 80486 DX system or better, 8Mb ram. A high resolution SVGA (1280 by 1024) 17” display will give a more productive working environment.

9.1.3 Platform Specifics

The operation of the Simulator under the graphic interface varies slightly from one platform to another. This is in the area of certain windows and dialog boxes supplied by the platform itself. In all aspects of the Simulator itself, operation, although not the details of appearance, is consistent across the platforms.

Although this section addresses some of the relevant differences between the platforms, it is assumed that the reader is familiar with his own environment. Although there are frequent references to window operations, no attempt is made to teach the windows or any other system.

After this introductory section, all screen illustrations are taken from the WINDOWS system.

9.1.3.1 General Window Behavior

Under Windows, all the Simulator windows are constrained within the area of the main window. To use the whole screen, the main window must be maximized. When one of the open windows is minimized, it appears as an icon within the main window.

Dialog boxes, however, appear in the center of the screen and are not bound by the main window. They may be moved as desired, some can be re-sized, none can be minimized, and all must be dismissed before any other operation may be performed.

Under motif, the windows are not bound by the main window. They may use the whole screen without restriction. When a window is opened, an icon appears in the main window. When a window is minimized, by clicking in the ‘down triangle’ in the top left corner, it becomes an icon at the right of the screen. These icons are not labelled, so use the icons in the main window (which are labelled) to choose which window to reopen.

Figure 9-3. Main window for Sun SPARCstation 2

9.1.3.2 File Chooser

The dialog box supplied by the platform for the purpose of selecting a file or directory varies significantly in appearance, although not in overall function. All have the same basic features:

- Drive selection (built into the ***X file structure)
- Parent and sub-directory selection
- List of files in current directory
- Accept selection or cancel operation
- A space to type a file name directly

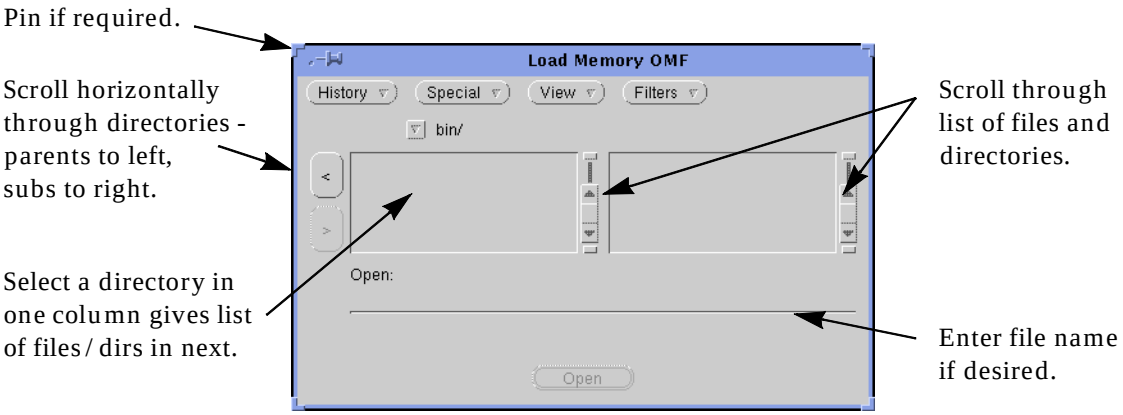


Figure 9-4. SUN File Chooser Dialog Box

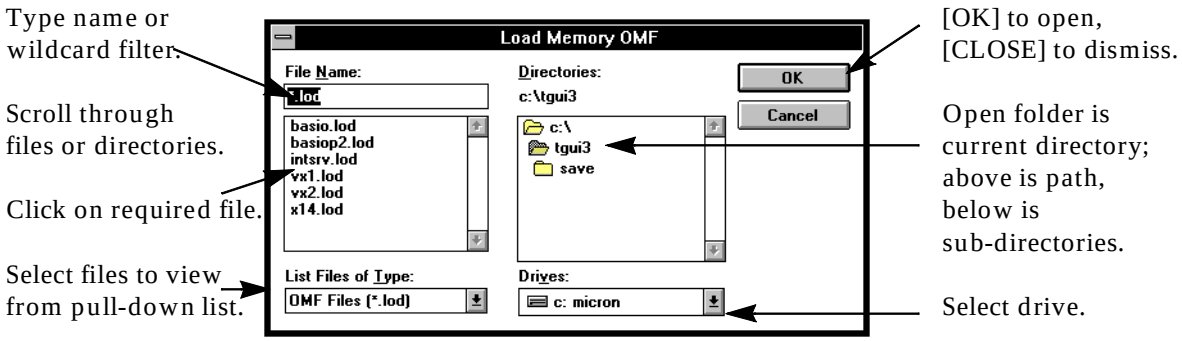
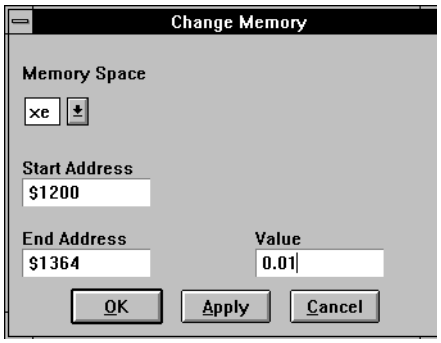


Figure 9-5. WINDOWS File Chooser Dialog Box

9.1.3.3 Multiple Operations



Many operations may need to be performed several times in succession. These include setting breakpoints, specifying display radix for various memory areas, etc. To avoid navigating the menus each time, the dialog box may be retained for (say) setting the next breakpoint.

Under Windows, such dialog boxes have three buttons - usually labelled [OK], [APPLY], [CANCEL]. Clicking on [OK] performs the operation and dismisses the dialog box, [APPLY] retains it for further operations. When the dialog box is no longer needed, it must be dismissed by using [OK] on the last operation, or [CANCEL]. No other GUI operations can be performed until the dialog box has been dismissed.

Under Motif, the same effect is achieved in a different way. There is only one button, [APPLY], which does what the dialog box requires, and then usually dismisses it. The dialog box can be made (semi) permanent by clicking on the pin in the top left corner, so it will not be dismissed after clicking the [APPLY] button. The dialog box may then be used as many times as required; click on the pin again to unpin it, and the window closes. To dismiss the dialog box, double-click on the pin, i.e. ‘pin and release’, and the dialog box closes.

9.1.3.4 Multiple Selections

Many dialog boxes permit the selection of several items from the list. This is handled differently on different platforms.

On Windows or the HP, a click with the left mouse button selects one item and clears any previous selection. Click and drag selects a range of consecutive items; the list scrolls when the drag reaches the end of the window. To add to an existing selection, hold the control key while clicking or click/dragging the extra items.

On the SUN, click or click and drag with the left button to make a selection and clear any previous selection; use the middle button to add to an existing selection.

9.1.4 Graphical Interface Functions Overview

The GUI provides a graphical interface to the debugger for the Motorola families of DSP devices. Versions support both the software DSP Simulator and the ADS emulation systems.

The GUI consists of a set of tools - menus, dialog boxes, windows and buttons. Using these tools, the user selects the desired operation, and the interface generates the appropriate commands for the development system. These commands are passed to the debugger via the COMMAND window, and the output and other information displayed in the SESSION and other windows. The user may also enter commands directly into the COMMAND window, so retaining direct control over the debugging process if desired.

These features provide full control over the development process. The menus provide the control functions, the dialog boxes gather additional information as necessary, and the windows display information, and also provide facilities to modify certain items such as register and memory values.

This section describes in general terms the range of features offered by the GUI. It is intended to provide a brief overview without going into great detail on any subject. References are included to the appropriate sections for further study.

9.1.4.1 Structure

The GUI provides an interface to the command line Simulator, generating commands from the user actions, and interpreting the responses.

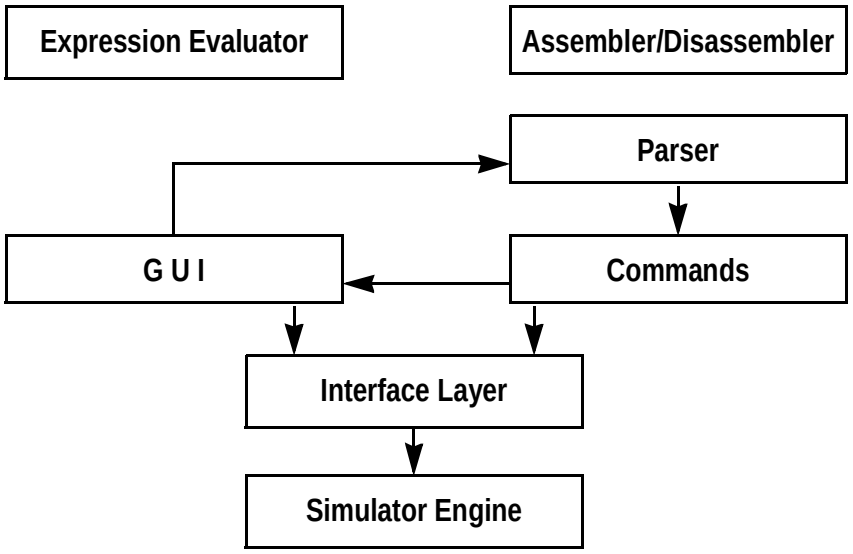


Figure 9-6. GUI Interface to Simulator

9.1.4.2 Starting the Simulator

At system start-up, the main window opens. This provides the menu for the system, and the button bar for convenient access to frequently-used operations.

Figure 9-7. Simulator Main Window

Other windows may also open. This is controlled by the Preferences item in the File menu. If checked, the positions of the windows are saved on exit, so the GUI starts with the required windows already open.

Running under Microsoft Windows, the main window is the whole work area. All windows are held within its bounds. To use the whole screen, it is necessary to maximize the main window, and similarly, when the main window is minimized, all the other windows go with it. On other platforms, the daughter windows are free to use any area of the screen. An icon representing each open window appears on the main window, which can be used to find a window hidden behind others, or reopen a minimized window.

9.1.4.3 File Access Paths

The debugger makes use of two types of path for creating and accessing files. The main path is used for created files (assuming no path is explicitly specified with the file name), and is the first place searched for an input file. This is known as the WORKING DIRECTORY.

ALTERNATE SOURCE PATHS are also searched, in turn, if an input file is not found in the working directory. Thus object files may be stored in one directory, and sources in another, and each may be accessed easily.

These paths are set up with Path... in the File menu.

9.1.4.4 Loading Object Files

The development system can load object files in COFF and OMF formats into simulated memory. These files may be produced by the DSP assembler and C compiler, with file types ‘.CLD’ and ‘.LOD’. COFF files may contain symbolic debugging information in addition to the object code, permitting the use of variable names and labels during the debug session. Use the FILE menu, LOAD option, to load the program into memory. If the source files are present (that is, in the object directory or one of the directories set up with FILE//PATH), the SOURCE window displays the source code around the current instruction.

9.1.4.5 Examining and Changing Memory

After loading the program, you can look at the program in memory. The ASSEMBLY window (Windows menu, Assembly option) lists the memory contents, disassembled. Symbolic references are included if symbolic data was loaded from a COFF file. The ASSEMBLY window also permits editing the program with assembler instructions, and one way of setting and clearing breakpoints. As the program executes, the ASSEMBLY window automatically refreshes to display the area around the PC.

In addition, the MEMORY window displays a block of memory as numeric values (Windows menu, Memory option). You can control the radix used to display each memory location (Modify menu, Radix option) individually. So if one location is a counter, it can display in decimal, if another is a bit mask, binary or hexadecimal might be more suitable. The MEMORY window can be re-sized to display more or less memory (the number of columns adjusts to use the width given), scrolled to cover the whole memory address range. Click on a location to modify an individual memory location. Several MEMORY windows may be opened, to display different memory areas concurrently.

To initialize a block of memory to the same value in each location, as in clearing a buffer, use the Modify menu, Memory option.

9.1.4.6 Examining and Changing Registers.

The registers can also be monitored with the REGISTER window (Window menu, Register option). All registers can be displayed, scroll to locate those you want. Registers can also be modified, as with the MEMORY window; see also Modify menu, Register option.

9.1.4.7 Program Execution - The Tool Bar

The tool bar provides convenient control of program execution. The green light allows program execution to proceed until interrupted, the red light interrupts it. STEP executes either an instruction, or a line of code, depending on whether the source information is available. NEXT is the same as STEP, except on meeting a call to a subroutine (or function, if you speak C). STEP treats the function like the rest of the code, and stops after each instruction in the function. NEXT treats the function as one instruction, and stops after it is finished.

9.1.4.8 Device Selection

The debugger can support multiple DSP devices, up to 32 depending on the configuration. Each device may be configured as part of this session, or excluded.

The DEVICE button selects which DSP processor is affected by user commands at any given time - which device's memory bank is displayed in this MEMORY window, which device's register is being changed, which device is affected by this breakpoint. This is called the DEFAULT DEVICE. The DEVICE entry in the MODIFY menu can configure and turn devices ON or OFF; when instructions are executed, all devices which are on will execute in turn.

9.1.4.9 Breakpoints

Program execution is controlled by the breakpoints. There are several ways of specifying breakpoints. The SOURCE window displays the source code for the executing program; double-click on a line of code to set (or clear) a breakpoint. There is no indication given in the SOURCE window, but the COMMAND window shows the command to set the breakpoint (or clear it), and the corresponding address in the ASSEMBLY window will be highlighted blue to show the position of the breakpoint. Similarly, a double-click in the ASSEMBLY window will set or clear a breakpoint on any instruction, not just the start of a line of code.

These are HALT breakpoints - the program is halted and control returns to the user. With the Execute menu, breakpoints can have several other actions associated with them. For example, incrementing a counter (four are available) lets you know how many times a piece of code was executed, a note can be written to the SESSION window record the event that the breakpoint was executed, or a selection of registers, memory locations, and expressions (values which may never have been calculated by the program during its normal execution, but which may be useful for you to know) can be displayed to the SESSION window. All this is set up by Breakpoint in the Execute menu.

So far all the breakpoints have been associated with program locations. It is also possible to place breakpoints in the data, so that when a specific register or memory location (or memory block) is accessed - wherever the PC is at the time - the breakpoint occurs and the specified action is performed. It is even possible to specify an expression as the breakpoint condition, so that, for example, if a pointer ever gets past the end of a buffer, the breakpoint occurs.

It is possible to set multiple breakpoints on a single location or event, to specify multiple actions - say increment a counter, display some values, and halt - at the same time.

So now we can load a program, look at and change the memory and registers, patch the program, execute all or part of the program, set breakpoints to interrupt execution when certain events occur, and we can monitor program activity with the MEMORY, REGISTER, ASSEMBLY and SOURCE windows.

9.1.4.10 Simulated Input and Output

DSP programs do not usually exist in isolation. We need to be able to simulate interaction with the electrical world outside the device. This is handled by Input and Output in the File menu.

Input associates a data file with some part of the device. Every time that entity is read, the value returned to the program is provided by the data file. Input from a file can be associated with a memory location, a group of device pins, a port, or a peripheral. So, for example, every time the program reads location X:$FFE0, the value returned is taken from the file.

This simulated input may be intended to represent a data stream, so that each access gets the next item. In this situation, time is not a consideration, and each access just gets the next data item. Each entry in the file can be read once, and only once, and cannot be skipped.

However, DSP devices frequently operate in the world of real time. It may be necessary to provide input, not on the basis of ‘next in line’, but ‘what should be input **at this time**’. To allow this, simulated real time is maintained in the form of a cycle count. This cycle count may be used as the basis for simulated input with TIMED INPUT. Here the data file contains not single items of data, but time/data pairs. These are interpreted as “at or after this cycle count, return this data value”. That value will remain in effect, and may be read many times or never, until the cycle count in the next time/data pair. At that time, the new data value will be available for input until it, too, is superseded by the next value.

Simulated output is similar. When a value is written to the specified location, pin, port, etc., a record is written to the output file. This may be pure data, or time/data pairs. Although intended to simulate output, this technique can also be used to provide a record of values written to a particular location.

Simulated input also provides for communication between multiple devices in a simulation. Device pins may be tied together, so the value returned when reading a pin depends on the state of another pin on the same or another DSP device. Similarly, memory locations may be connected, so the value read from one location may be provided by the value stored in another.

9.1.4.11 Stream File Support

Support is also provided for the basic C stream files, STDIN, STDOUT and STDERR. A C program running on the DSP device may use these files, and the IO will be handled by the host. See File menu, Stream to enable and disable stream IO, and File menu, Redirect to redirect the streams to files on the host system. If stream support is disabled, or the file accessed has not been redirected, the request is ignored. Output is discarded, no input is returned.

9.1.4.12 COMMAND and SESSION Windows

There are two windows which are involved in most GUI operations. These are the COMMAND and SESSION windows.

Most GUI operations generate commands for the debugger. These commands are passed to the debugger, and stored in a history buffer which is displayed in the COMMAND window. Stored commands may be retrieved, edited and re-executed. A command entry line permits commands to be entered manually, and a help line gives the syntax of the command being entered. There is only one COMMAND window, shared among all the devices in use.

The SESSION window is, in effect, the main screen for the current device. Whenever the debugger generates output, it is written to the SESSION window. When a command is executed, it is echoed in the SESSION window. When an error is detected, it is reported in the SESSION window. The Display menu basically causes information to be output to the SESSION window.

9.1.4.13 Command and Session Log Files

All activity in the COMMAND and SESSION windows may be recorded to a log file. See LOG in the FILE menu.

All commands entered through the COMMAND window (manually or from the GUI) may be written to a log file. This can serve as a record of the command input to a session, but it can also be used as command input itself. A MACRO file is an ASCII text file containing ADS/Simulator commands, which can be read and executed. A command log is one way of creating such files. See Macro in the File menu.

All activity in the SESSION window can be logged as a permanent record of a debugging session. Thus all the breakpoint data, memory and register values output to the SESSION window, may be examined and analyzed later.

Although there is only one SESSION window, each device has its own output buffer. The SESSION window displays the buffer for the current device; activity on any other device will be recorded in its own buffer (and possibly also written to its own SESSION log file), and displayed when that device becomes the current device.

9.1.4.14 Save Files

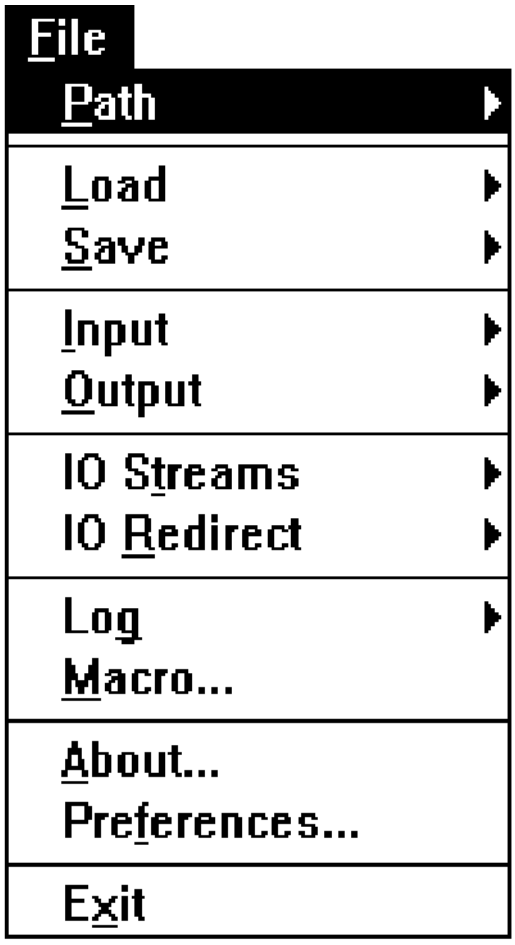
At the end of a development session (or indeed any other convenient time), all or part of the system status may be saved.

The entire debugger configuration - all memory and register contents, counters, display settings, breakpoints, etc., may be saved to a Simulator status file. This may be reloaded later, and development may proceed from where it was interrupted. This is handled by Save State and Load State in the File menu.

Memory contents may be saved as COFF or OMF object modules. These files will contain any patches applied during the session. See Save... in the File menu.

Finally the window positions may be saved on exit. See Preferences in the File menu. The next time the debugger is used, the windows will open where they were left.

9.2 FILE menu



The File menu handles all operations associated with file handling. The operations covered are:

•Set File Access Paths. Specifies a primary directory as the default for all file operations, and alternate paths for file read operations. Separate paths are maintained for each DSP device.

•Load and Save operations for loading object modules into memory, writing selected memory areas out into object modules, and saving and reloading the entire status of the development system.

•Simulate Input and Output for the development system. Provides simulated data for a program, and saves output produced by a program.

•IO Streams and IO Redirect provide a basic stream I/O environment for C programs running on the development system. Stream IO may be enabled or disabled, and the basic stream files STDIN, STDOUT, STDERR redirected to files on the development host.

•Command and Session windows may be logged to files.

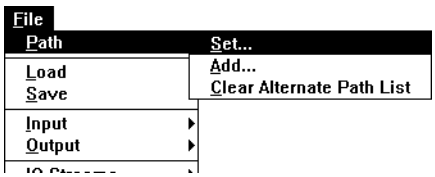
•Commands in a Macro file may be executed.

•About displays the version of the program, and claims and acknowledges copyright.

•Preferences controls the saving of window positions.

•Exit the debugger.

9.2.1 FILE//PATH//...



A separate file search path is maintained for each device. FILE//PATH//SET sets the default directory, referred to as the WORKING DIRECTORY, for all file accesses for the current device (see MODIFY//DEVICE). FILE//PATH//ADD sets the ALTERNATE SOURCE PATHS for the current device.

On all file operations, the working directory specified in FILE//PATH//SET is used as the initial directory in the file open dialog box.

The alternate source path is used if a command is typed directly into the command window, or a file name is typed into a dialog box, specifying a file name without a path. In this case, an output file will be created in the working directory, and an input file will be searched for, initially in the working directory, and if not found there, in each alternate source directory in turn until found.

FILE//PATH//CLEAR... removes all alternate source directories specified by FILE//PATH//ADD. All future file accesses for this device will only use the working directory.

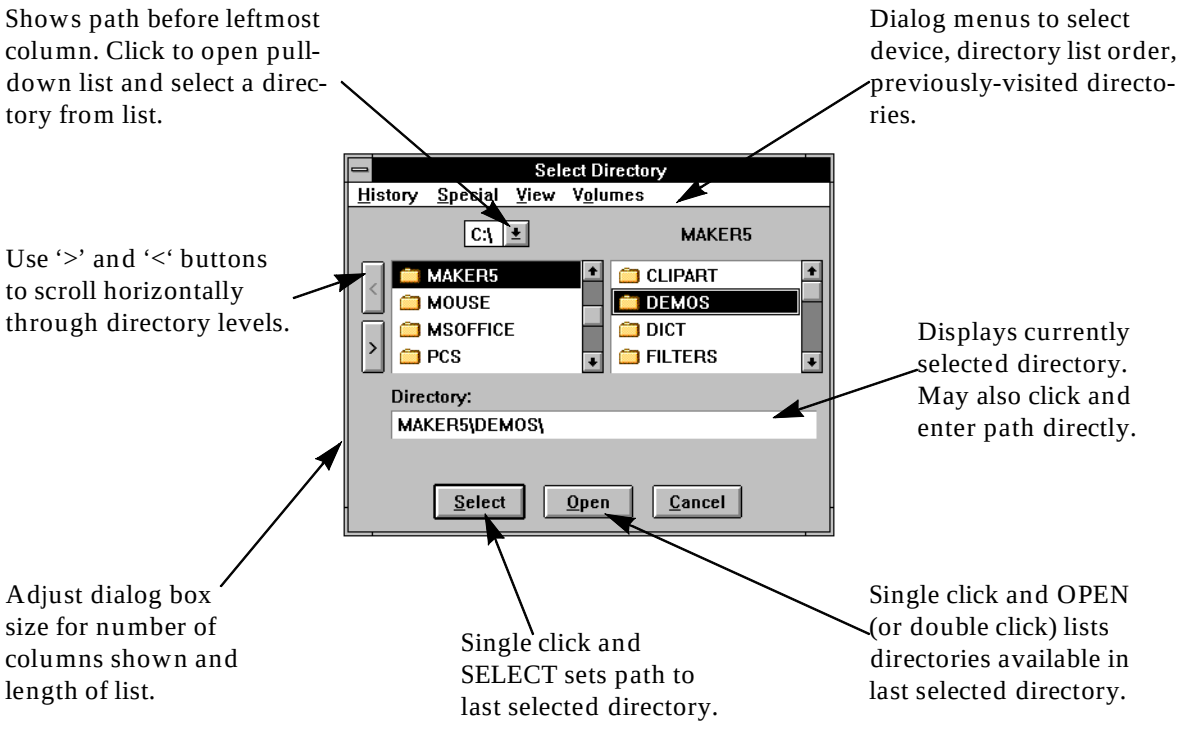
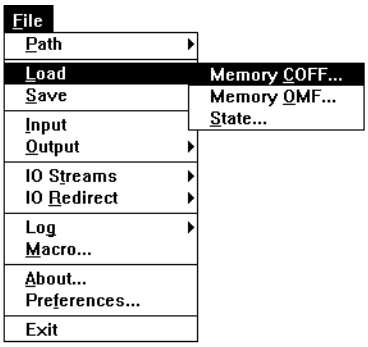


Figure 9-8. FILE//PATH/SET, ADD dialog Box

9.2.2 FILE//LOAD//MEMORY



The FILE//LOAD//MEMORY... menu items read object modules in OMF or COFF format into the DSP memory for the current device (see MODIFY//DEVICE//SET DEFAULT). Complementary functions FILE//SAVE//MEMORY... are available to preserve memory contents in OMF or COFF files which may themselves be loaded.

If MEMORY COFF load is selected, a dialog box gives the choice of loading memory, debug symbols or both. Otherwise, the operation is identical for both OMF and COFF files.

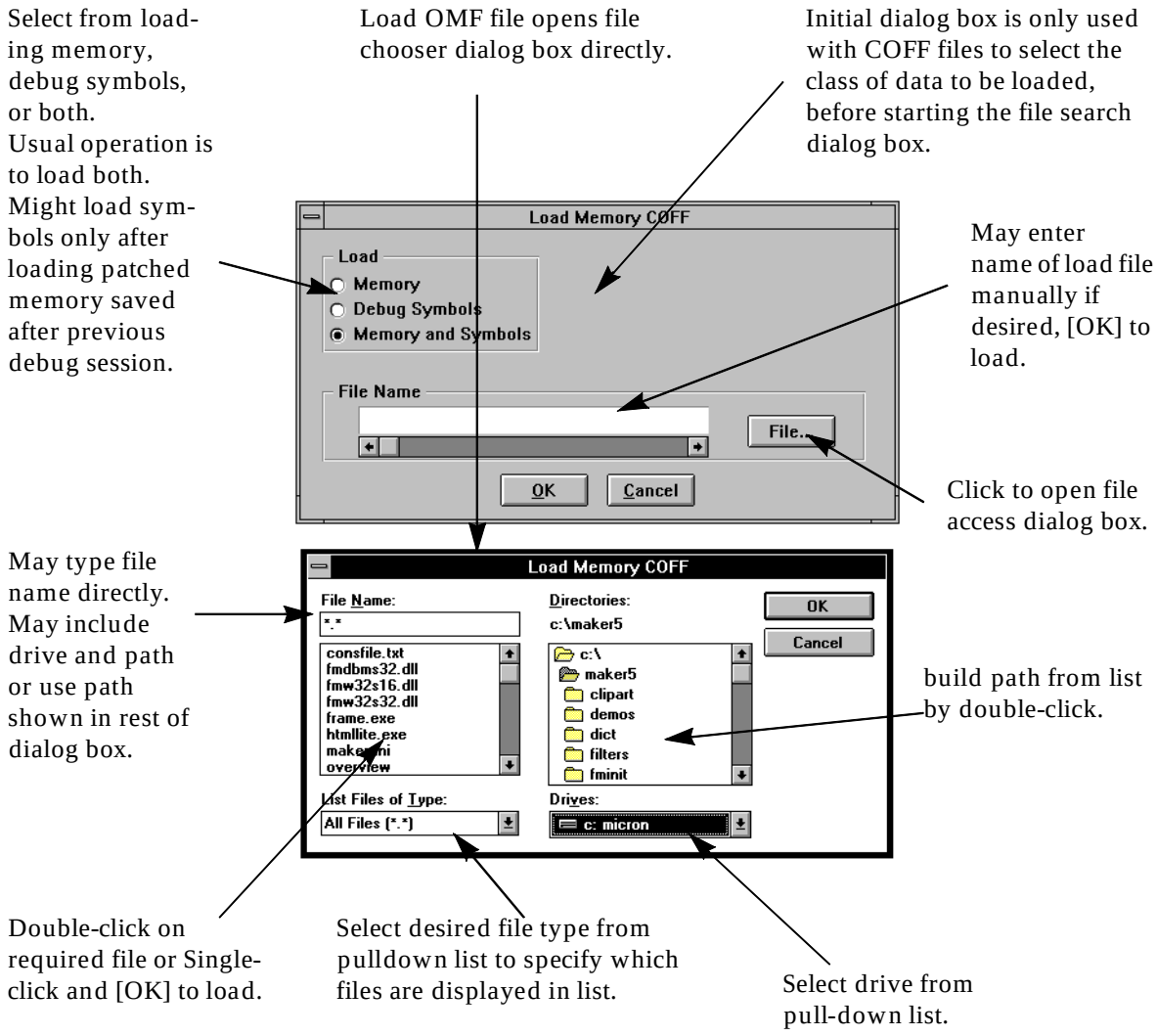
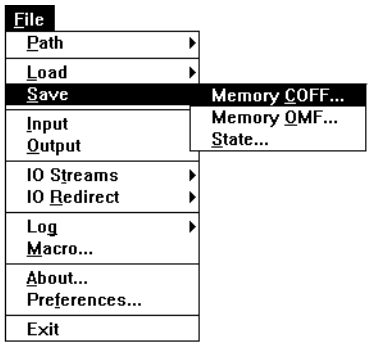


Figure 9-9. FILE//LOAD//COFF, OMF dialog Box

9.2.3 FILE//SAVE//MEMORY...



FILE//SAVE//MEMORY... menu items save contents of memory into DSP COFF or ASCII OMF files which may later be reloaded with the Simulator or in any other environment where such files may be used.

A dialog box is used to specify which area of memory is to be written, by specifying the memory space (p, x, y, etc.) and the address range. A separate operation is required for each memory space to be saved.

FILE//SAVE complements FILE//LOAD//MEMORY.

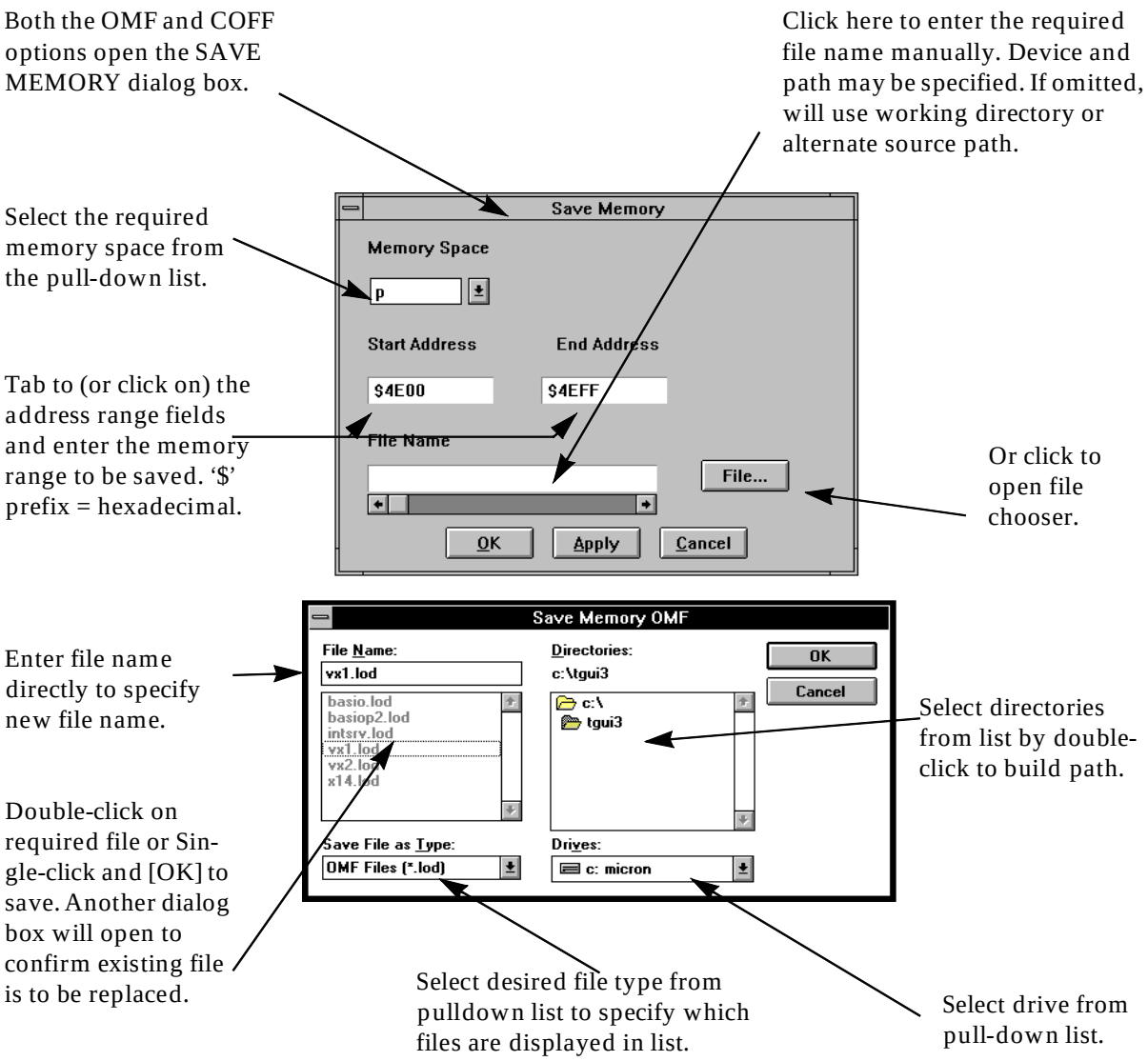
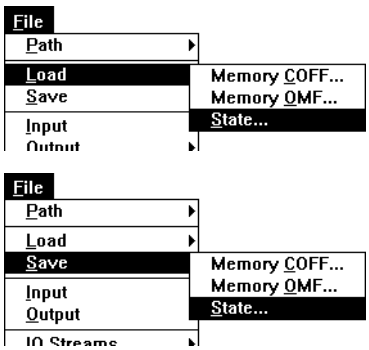


Figure 9-10. FILE//SAVE//COFF, OMF dialog Box

9.2.4 FILE//SAVE//STATE

- see FILE//LOAD//STATE

9.2.5 FILE//LOAD//STATE



the LOAD and SAVE STATE menu items allow the state of the entire Simulator to be saved and later reloaded. This includes the state of all DSP devices in the system, their device type, and for those devices which are enabled, the entire contents of memory, registers, counters, status registers, peripheral registers, etc. Additionally, the state of the GUI is saved, including the command history buffer, and the session output buffer for each device.

This may be used in several ways. A protracted development session may be saved before a break, and reloaded after the interruption to be continued where it was left off. Alternatively, if a particular part of a program is proving troublesome, the state may be saved just before the problem area, simplifying the setup for repeated attempts to isolate the problem. Or a set of standard routines and data areas may be pre-loaded in a Simulator state file, making it easy to set up the environment for testing some new code

The dialog boxes for LOAD//STATE and SAVE//STATE are identical in layout and operation. Only the titles differ.

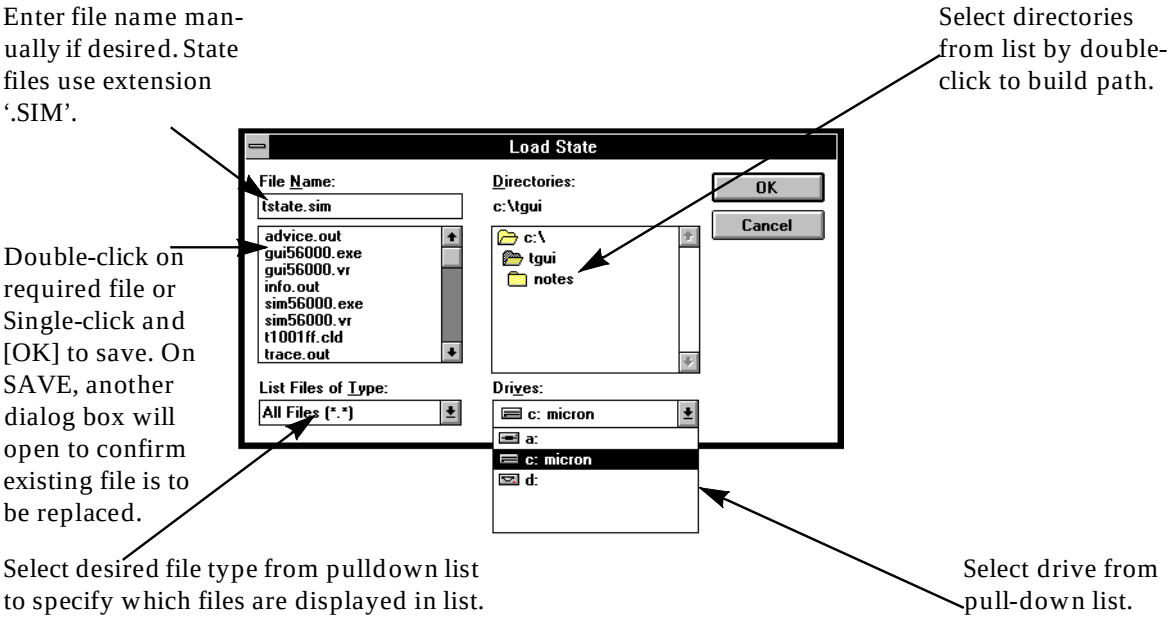
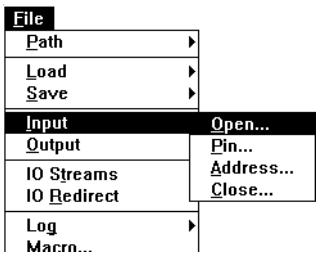


Figure 9-11. FILE//LOAD STATE, SAVE STATE dialog Box

9.2.6 FILE//INPUT//OPEN



FILE//INPUT//OPEN reads data from the terminal or a file to provide simulated input for a peripheral, port, pin or memory location in the default device. Whenever the program reads the specified object, the value returned is determined by the data file.

The data file may contain data only, in which case each access to the object will return the next value in the data file. Alternatively, the file may contain time/data pairs. In this case, each pair specifies the value to input at or after the specified cycle count. Repeated accesses will return the same value until the simulated cycle count reaches the time specified in the next time/data pair.

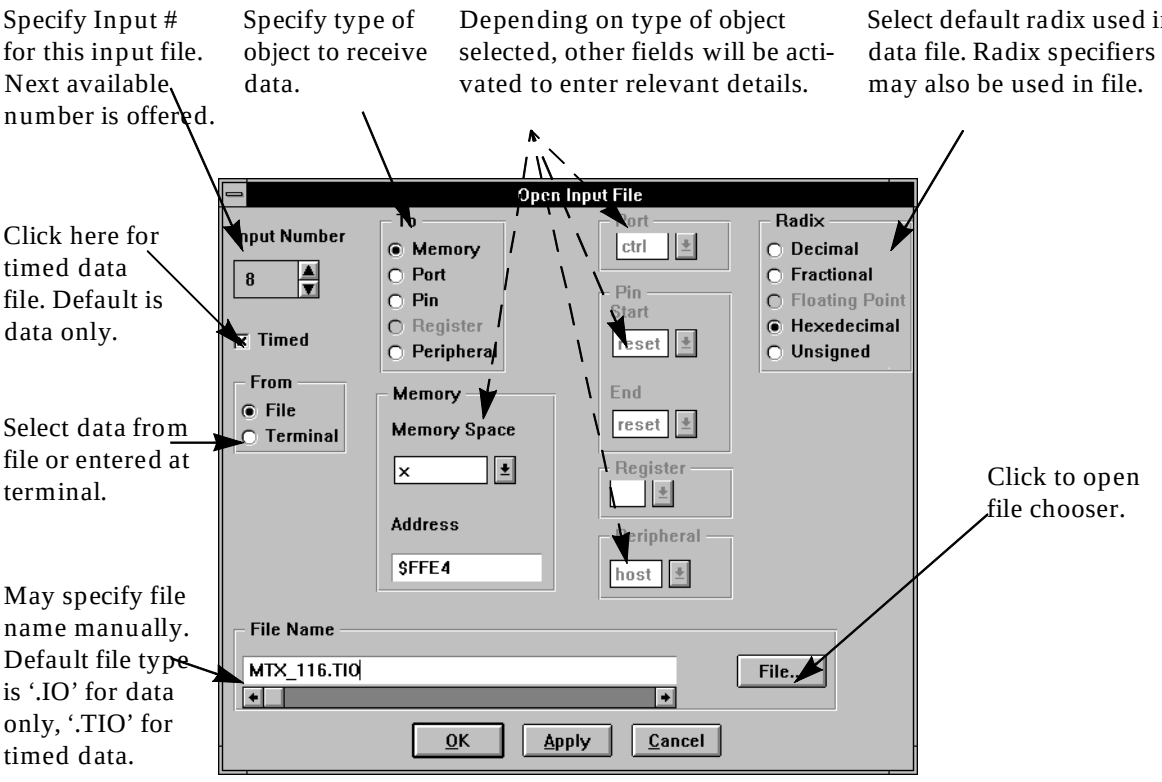
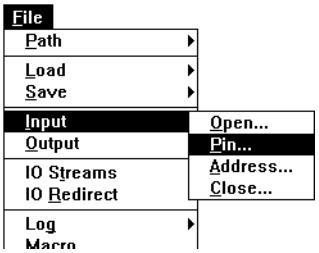


Figure 9-12. FILE//INPUT//OPEN dialog Box

9.2.7 FILE//INPUT//PIN



INPUT PIN provides a logical connection to a pin on the default DSP device from another DSP device pin - either on the same or a different device. Thus when a DSP device reads a pin which has been connected, the value returned depends on the state of the specified source pin.

Note that all FILE//INPUT reference numbers must be unique. Using INPUT # 1 for INPUT//PIN will close any INPUT//..... set up previously with INPUT # 1.

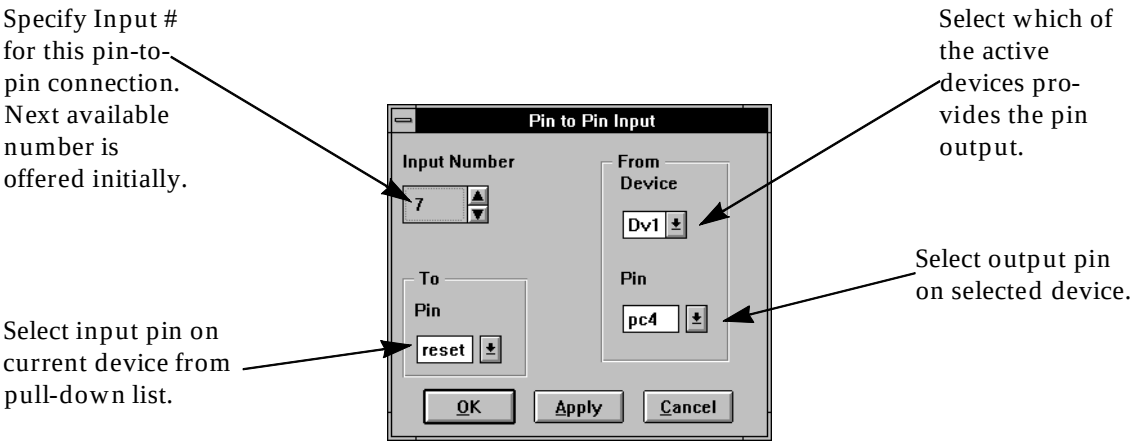
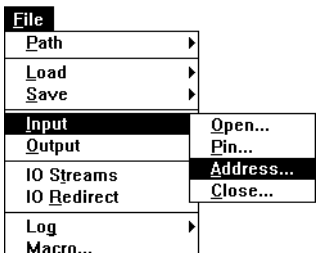


Figure 9-13. FILE//INPUT//PIN dialog Box

9.2.8 FILE//INPUT//ADDRESS



INPUT ADDRESS provides a logical connection from a memory location in one DSP device to another memory location - either in the same or a different device. Thus when a DSP device reads a memory location which has been linked with another location, the value returned depends on the contents of the source location.

Note that all FILE//INPUT reference numbers must be unique. Using INPUT # 1 for INPUT//ADDRESS will close any INPUT//..... set up previously with INPUT # 1.

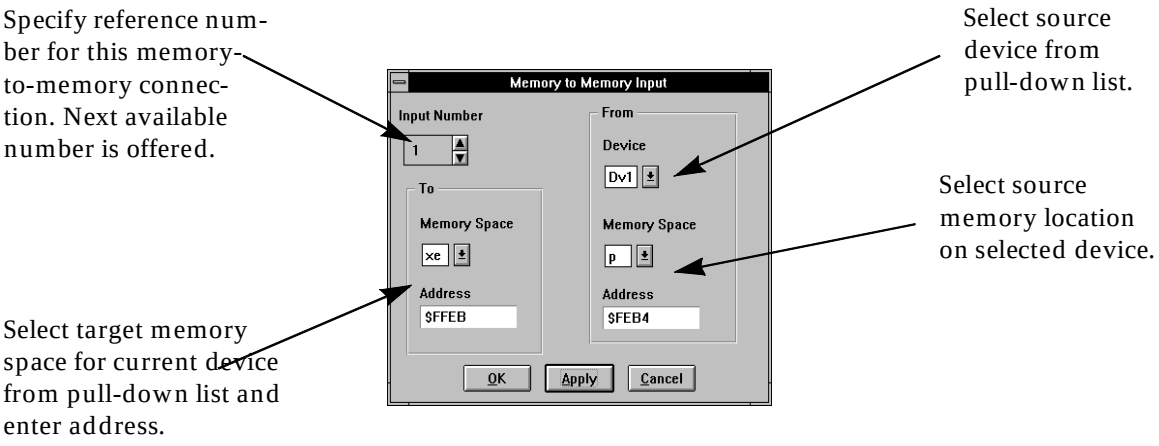
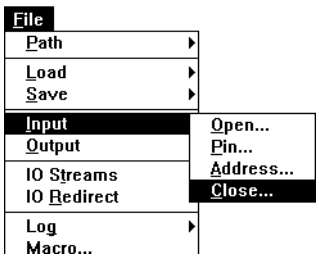


Figure 9-14. FILE//INPUT//ADDRESS dialog Box

9.2.9 FILE//INPUT//CLOSE



INPUT//CLOSE closes all or selected simulated inputs to the default device.

A dialog box opens, offering all of the currently open input numbers for the default device.

Select the inputs to be closed, using the appropriate combination of mouse clicks, <CTRL>-CLICK, and CLICK-AND-DRAG.

Then close all selected inputs by clicking [OK].

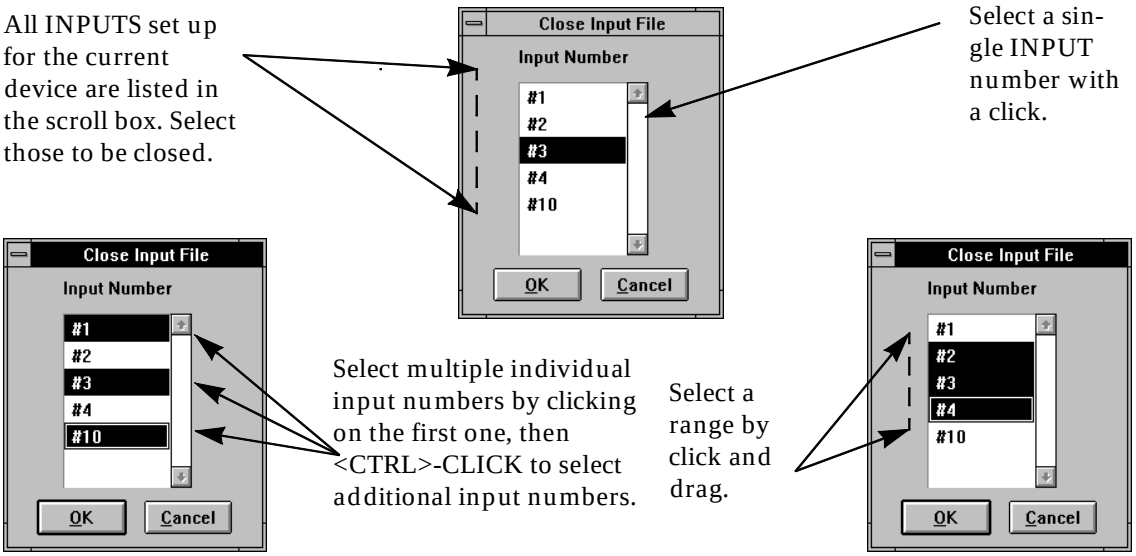
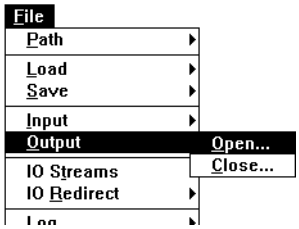


Figure 9-15. FILE//INPUT//CLOSE dialog Box

9.2.10 FILE//OUTPUT/OPEN



FILE//OUTPUT writes a single data item from the default device to a file or to the session window (terminal). The data item may be a single memory location, a port, a range of pins, a peripheral, or execution history. A separate output file must be established for each data item to be written.

A record is output each time a data item is written or when a pin changes state.

HISTORY writes a record to the file for each instruction execution. EXTENDED HISTORY writes a record for each execution cycle. The last record in the file is always the next instruction to be executed.

The output record contains a record number, the optional timing field containing the cycle number, and the data value. For the history file, the data comprises the PC, the instruction word(s) in hexadecimal, and the disassembled instruction.

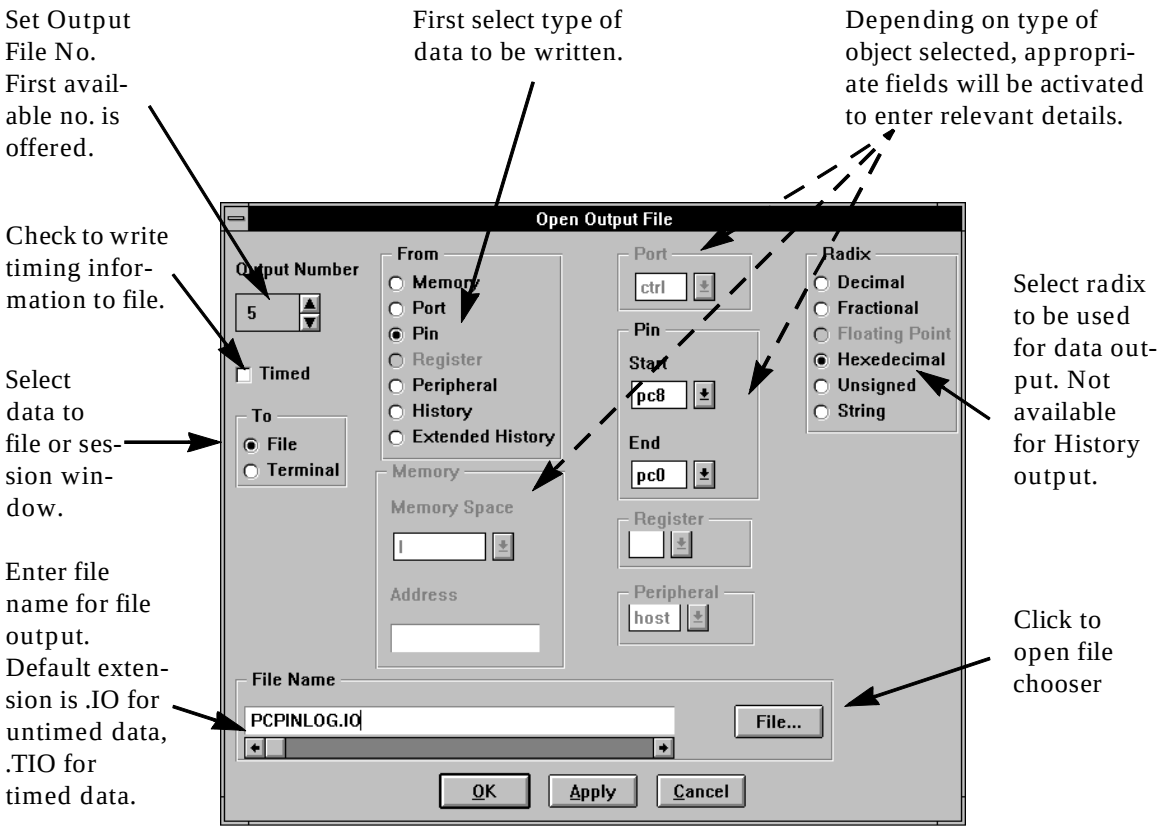
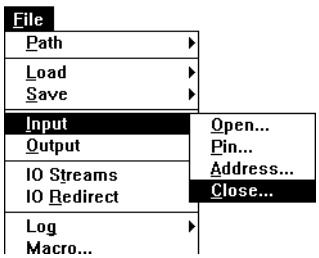


Figure 9-16. FILE//OUTPUT//OPEN dialog Box

9.2.11 FILE//OUTPUT//CLOSE



OUTPUT//CLOSE closes all or selected outputs from the default device.

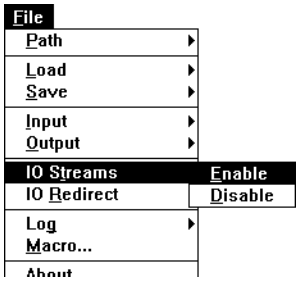
A dialog box opens, offering all of the currently open output numbers for the default device.

Select the outputs to be closed, using the appropriate combination of mouse clicks, <CTRL>-CLICK, and CLICK-AND-DRAG.

Then close all selected outputs by clicking [OK].

See FILE//INPUT//CLOSE for close dialog box usage illustration.

9.2.12 FILE//IO STREAMS//...

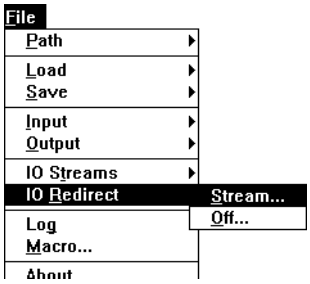


FILE//IO STREAMS enables or disables stream I/O for C programs running on the current device. The standard stream files are supported - STDIN, STDOUT, and STDERR. Any references by C programs to these files may be redirected to files on the host. See FILE//IO REDIRECT.

Stream file handling may be configured independently for each device. By default streams handling is enabled.

If a C program attempts to access a stream file while it is not enabled and redirected, the access is ignored. Output is discarded, and a standard value is supplied as input.

9.2.13 FILE//IO REDIRECT//...



FILE//IO REDIRECT//STREAM redirects the selected stream on the current device to a file on the host. Each stream file may be assigned individually; unwanted streams do not have to be redirected.

Streams may be redirected whether stream support is enabled or disabled; however, for the redirection to be effective, stream operations must be enabled. Disabling stream support while a stream is redirected does not terminate the redirection. It merely makes it ineffective until streams are enabled again.

FILE//IO REDIRECT//OFF ends redirection of one or more streams for the current device. Only streams which have previously been redirected may be selected.

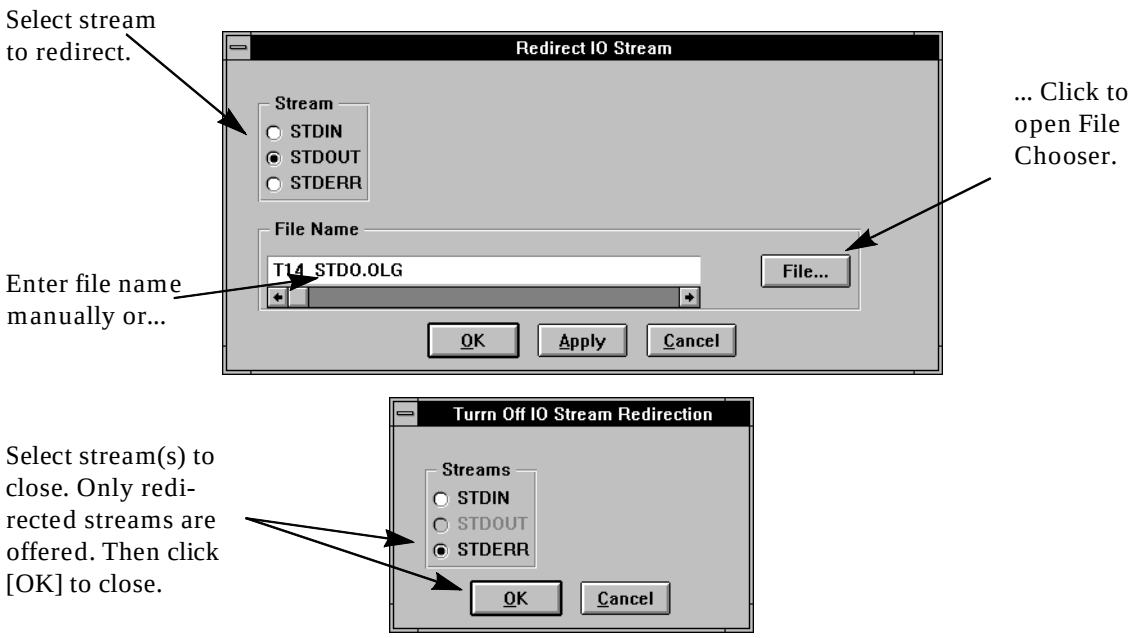
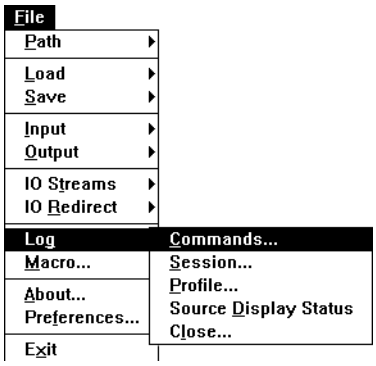


Figure 9-17. FILE//IO REDIRECT//... Dialog Boxes

9.2.14 FILE//LOG//COMMANDS



FILE//LOG//... menu items control the creation of files containing a record of a debugging session. Recording may be started and terminated at any time during the session.

Selecting FILE//LOG//COMMANDS opens the Open Log File dialog box. If an existing file is selected for logging, an action confirmation box opens, with options to append to the existing file, overwrite it, or cancel the operation.

The command log file has two main purposes. Its obvious purpose is to record a development session. In addition, the log file may also be used in the GUI as a macro file (see FILE//MACRO), when all the commands recorded in the log file will be executed. This file is a standard ASCII text file, and may be modified with any text editor as desired.

Note that nearly all GUI operations, including menu operations and window interaction, result in commands executed in the COMMAND window, and will thus be stored in the log file

.

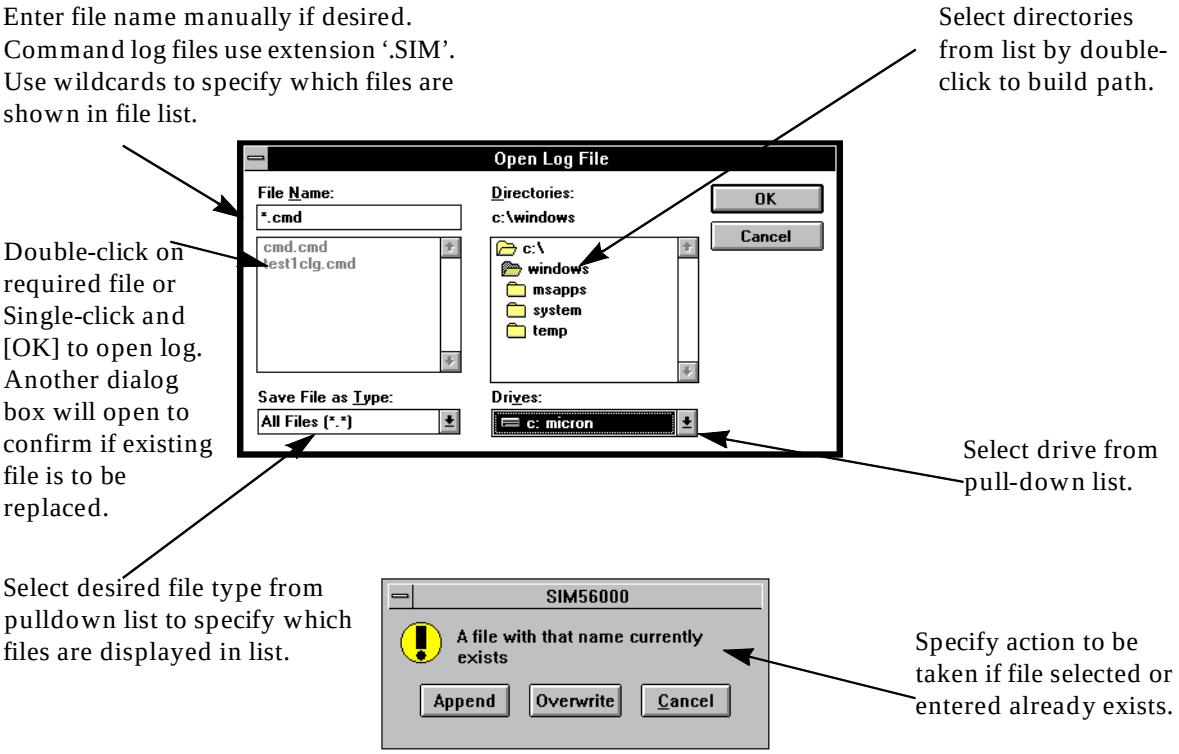
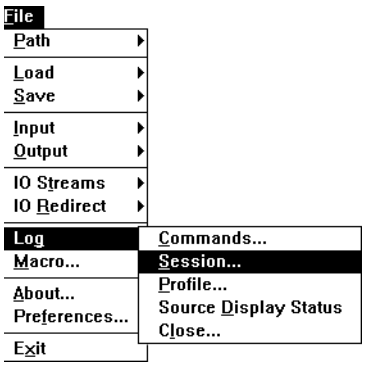


Figure 9-18. FILE//LOG//COMMANDS dialog Box

9.2.15 FILE//LOG//SESSION



FILE//LOG//SESSION logs the SESSION window for the active device (see MODIFY//DEVICE//SET DEFAULT) to a file. Logging may be started and stopped at any time. A separate log file may be established for each device. The SESSION window need not to be open for the session log to be written.

Selecting FILE//LOG//SESSION opens the Open Log File dialog box. If an existing file is selected for logging, an action confirmation box opens, with options to append to the existing file, overwrite it, or cancel the operation.

Everything output to the SESSION window while in REGISTER mode (see DISPLAY// VIEW//REGISTER) is written to the session log file. Changed values and error messages displayed in red in the SESSION window are enclosed in braces ({}).

Using the LIST FILE window, the session log can be viewed without closing the log first, bypassing the limit on the session buffer size. However, anything written to the SESSION window after opening the LIST FILE window will not be accessible in that window

Selecting LOG//SOURCE DISPLAY STATUS writes an additional line to the SESSION log. This requires that the SOURCE window must be tracking the source, or the SESSION window must be set to VIEW SOURCE in the DISPLAY menu...

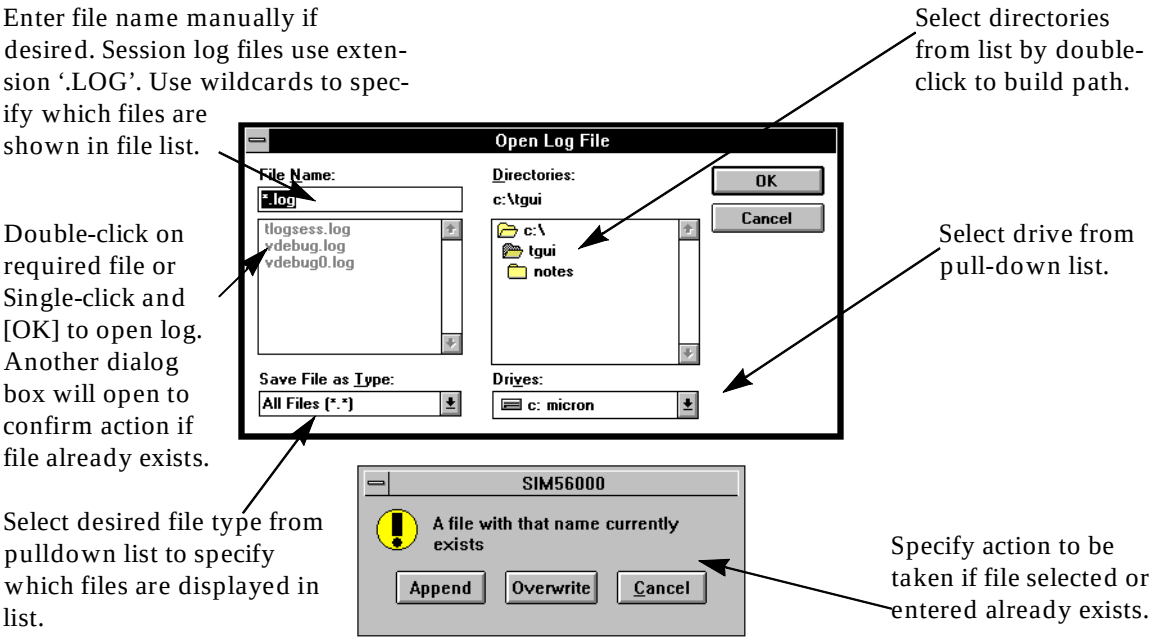
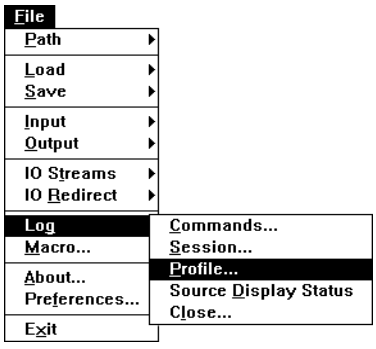


Figure 9-19. FILE//LOG//SESSION dialog Box

9.2.16 FILE//LOG//PROFILE



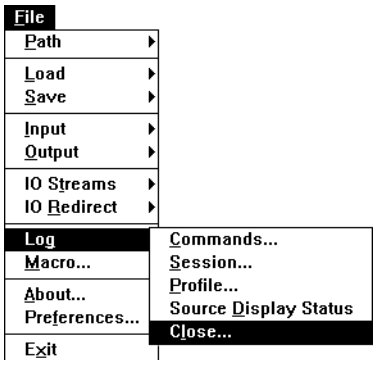
Use FILE//LOG//PROFILE to create a profile or analysis of a program executing on the Simulator. Before opening the profile log, it is necessary to load the program to be profiled from a COFF (.cld) file, loading both memory and symbols.

The dialog box is similar to those used for other log files.

The profiler provides a detailed analysis of all aspects of instruction execution from the time it is turned on until it is turned off again. Two output files are produced, a '.log' file which is a text file suitable for any printer, and a '.ps' file, a postscript file containing the same information as the '.log' file, which produces a better formatted printout when printed on a postscript printer with the appropriate font support.

The program profile includes an analysis of the program composition - number and percentage of each type of instruction in the program, and a similar analysis of the instructions executed. Other features include an analysis of subroutine interaction during program execution, a full breakdown of the use of addressing modes with each type of instruction, even the run time of the program in clock cycles. A description of the profile log file appears in appendix A.

9.2.17 FILE//LOG//CLOSE



Use FILE//LOG//CLOSE to close all or any of the currently open log files for the current device. The Close Log File dialog box offers a list of log files which may be closed; click the check boxes as required and clock [CLOSE] to close the log(s). The check box for any log which is not currently active is shown shaded.

Note: only log files for the current default device will be closed.

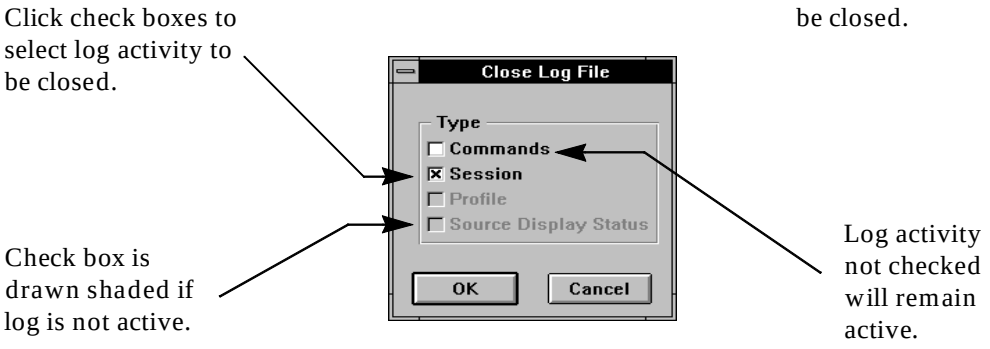
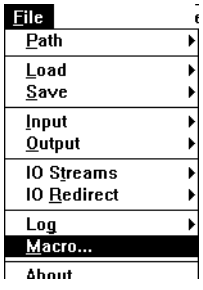


Figure 9-20. FILE//LOG//CLOSE dialog Box

9.2.18 FILE//MACRO



FILE//MACRO reads and executes a file containing commands for the Simulator. These commands are documented in the Simulator Commands chapter.

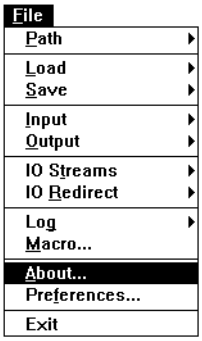
The MACRO file is a standard ASCII text file, and may be created or edited with any text editor. The default file extension is '.CMD'.

Command log files created with FILE//LOG//COMMAND may be submitted as MACRO command files.

As the commands are read from the MACRO file, they are displayed in the COMMAND window, executed, and echoed in the SESSION window, along with any output generated. Commands which affect an individual device will execute on the current device, unless the command specifies a particular device. Thus a single command file may be executed repeatedly, if required, for a number of devices by selecting a different device before each execution.

Macro file execution may be aborted by EXECUTE//STOP or the STOP light button on the tool bar.

9.2.19 FILE//ABOUT



FILE//ABOUT displays an information panel which identifies the product name and version, that Motorola has copyright on the product, and acknowledges copyright of software incorporated into the product. This notice is displayed during start-up, and closes automatically if not dismissed within three seconds.

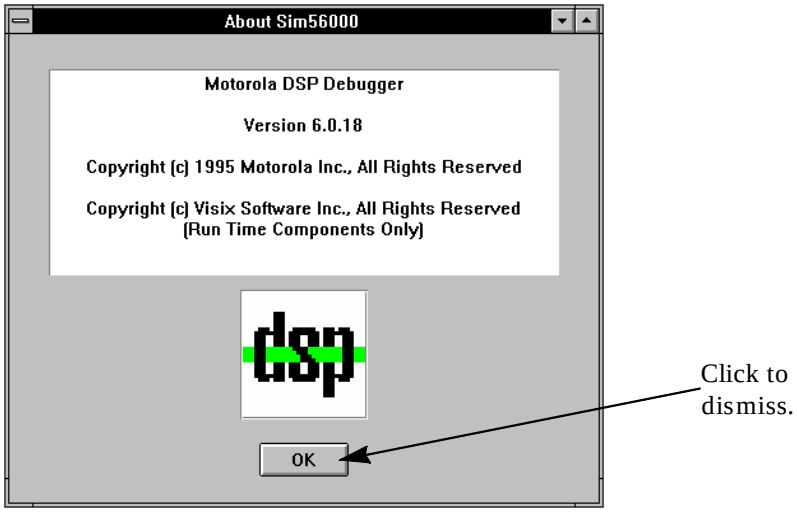
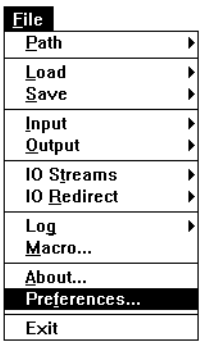


Figure 9-21. FILE//ABOUT dialog Box

9.2.20 FILE//PREFERENCES



The preferences dialog box provides the option to save the window positions on exit. Thus when restarting the Simulator, all windows will be restored to their positions on exit.

This may be used in two main ways.

If left checked permanently, each session will start with the windows positioned as they were left at the end of the last session.

Alternatively, if you prefer the windows to start arranged the same way each time, arrange the windows, check the save box, and exit. Restart and clear the check box. Each time the debugger is started the windows will be arranged the way they were saved.

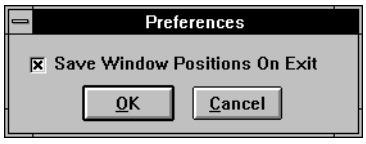
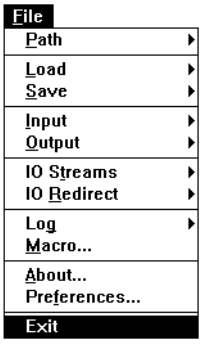


Figure 9-22. FILE//PREFERENCES dialog Box

9.2.21 FILE//EXIT



This option will exit the debugger. The exit dialog box pops up to make sure you intended to exit. This dialog box is also activated by other exit procedures.

Figure 9-23. FILE//EXIT dialog Box

9.3 DISPLAY menu

The Display menu controls the SESSION window. Most of the options cause output to the SESSION window, a few control the way it operates.

Note that each device has its own session buffer. Make the intended device the current device before performing any Display menu operations intended to relate to that device.

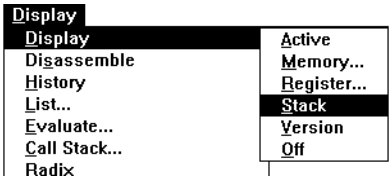
Most of the facilities offered by the Display menu may be obtained in other ways with the dedicated windows. However, the SESSION window does have one advantage - the option to write all SESSION output to a log file.

As all output from the Display menu is sent to the SESSION window for the current device, if the description of any Display menu item does not specify where the output goes, it is assumed to be the appropriate SESSION window.

Features provided:

- •Display selected registers & variables
- •View memory as instructions
- •View last 32 executed instructions
- •List source file in SESSION window
- •Calculate assembler and C expressions
- •Display C call stack frames
- •Set default input and display radix
- •Display device configuration and supported types
- •Display working directory and alternate source paths
- •Display simulated input assignments
- •Display simulated output assignments
- •List stream IO redirection
- •IO stream support enabled/disabled
- •List log file assignments
- •List breakpoints
- •Control expression display at breakpoints
- •Display the type of a C expression
- •Suspend SESSION window output when full
- •Select operating mode of SESSION window

9.3.1 DISPLAY//DISPLAY//ACTIVE



Write the enabled registers and memory locations to the SESSION window. See DISPLAY//DISPLAY//MEMORY D/ /D//REGISTERS, and D//D//WATCH. This is the same display as presented in the SESSION window whenever program execution stops.

The initial setting is all registers and no memory displayed.

Note that DISPLAY//DISPLAY//... is hardware oriented and intended to monitor the DSP processor memory and registers. DISPLAY//WATCH//... provides a similar facility which is also able to monitor program variables and expressions.

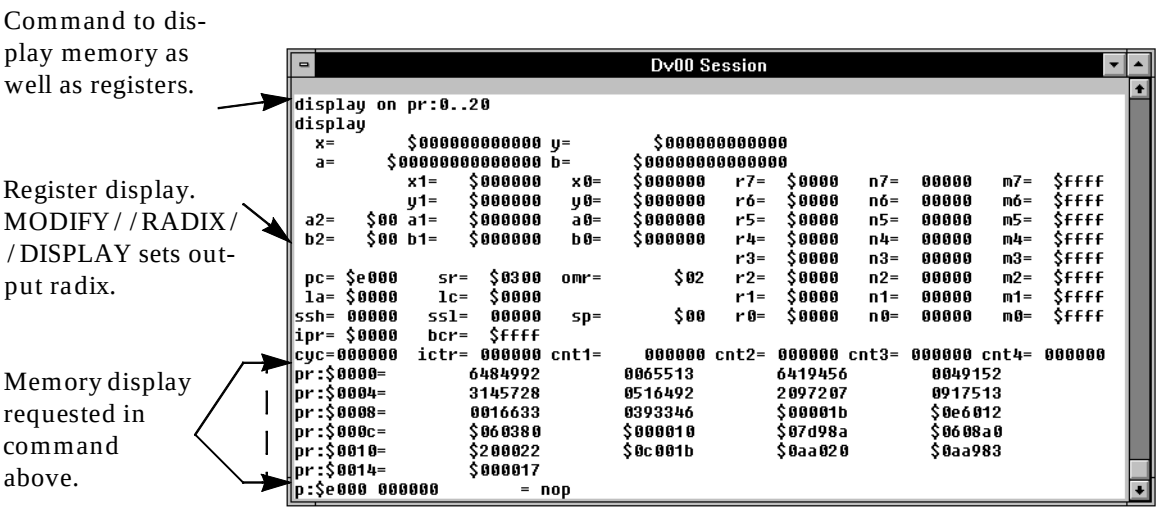
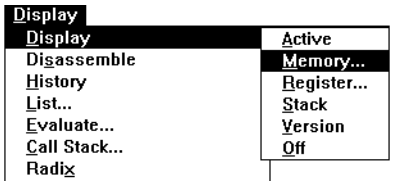


Figure 9-24. DISPLAY//DISPLAY//ACTIVE Output

9.3.2 DISPLAY//DISPLAY//MEMORY



Controls the display of memory areas either immediately or as part of the post-execution display.

Post-execution display may be unconditional or conditional on the way in which memory has been accessed during execution.

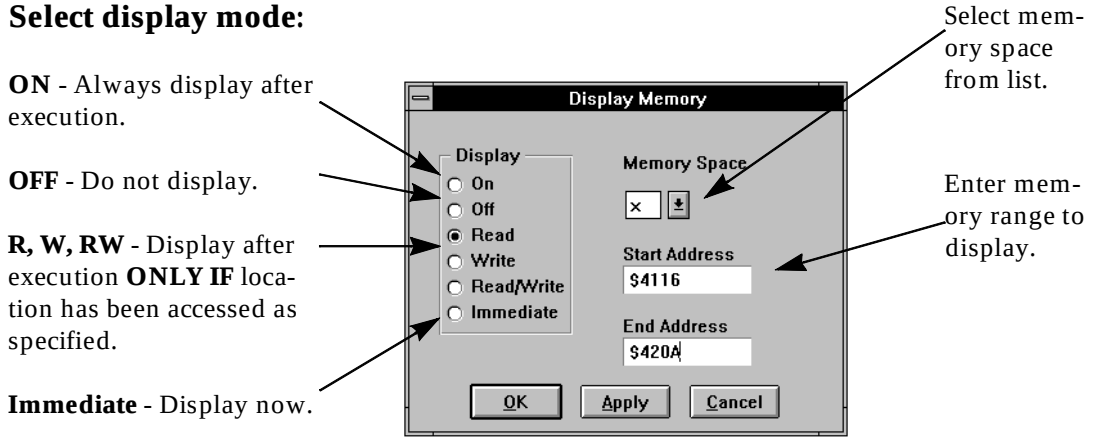
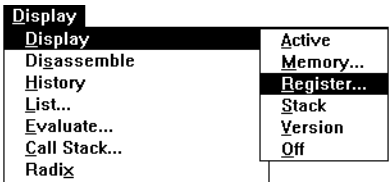


Figure 9-25. DISPLAY//DISPLAY//MEMORY Dialog Box

9.3.3 DISPLAY//DISPLAY//REGISTERS



Controls the display of registers either immediately or as part of the post-execution display.

Post-execution display may be unconditional or conditional on the way in which registers have been accessed during execution.

OFF cancels conditional and unconditional display.

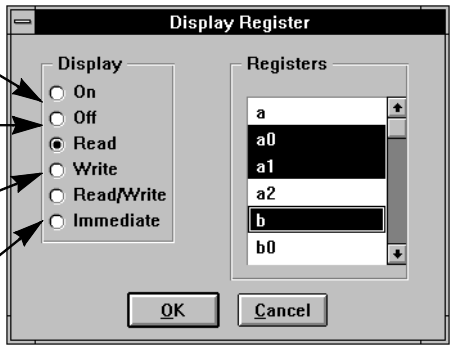
Select display mode:

ON - Always display after execution.

OFF - Do not display.

R, W, RW - Display after execution **ONLY IF** location has been accessed for...

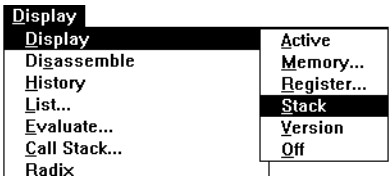
Immediate - Display now.



Select registers with CLICK, CLICK / DRAG, CTRL-CLICK.

Figure 9-26. DISPLAY//DISPLAY//REGISTERS Dialog Box

9.3.4 DISPLAY//DISPLAY//STACK



Output the stack to the SESSION window. The entire stack is output, with the current top-of-stack marked and the active stack area highlighted in red.

All 16 stack levels displayed.

Top of Stack pointer.

Active stack highlighted red.

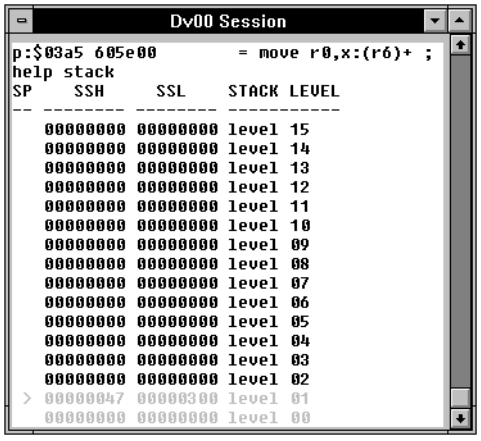
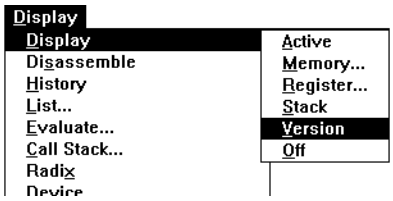


Figure 9-27. DISPLAY//DISPLAY//STACK Output

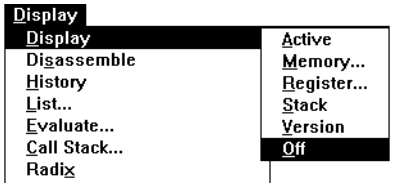
9.3.5 DISPLAY//DISPLAY//VERSION



Displays debugger version number and production date.

Figure 9-28. DISPLAY//DISPLAY//VERSION Output

9.3.6 DISPLAY//DISPLAY//OFF



Cancels all memory and register display at end of execution.

Note - STEP executed, but no register or memory display.

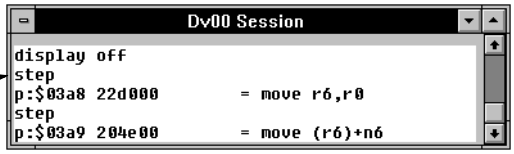
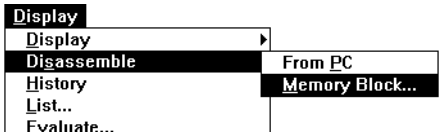


Figure 9-29. DISPLAY//DISPLAY//OFF Output

9.3.7 DISPLAY//DISASSEMBLE//FROM PC

9.3.8 DISPLAY//DISASSEMBLE//MEMORY BLOCK



DISPLAY//DISASSEMBLE// ... reads the specified memory area, disassembles it and writes it to the SESSION window.

DISASSEMBLE//FROM PC reads memory starting with the PC address, fills SESSION window and stops. Each subsequent use continues from last location decoded.

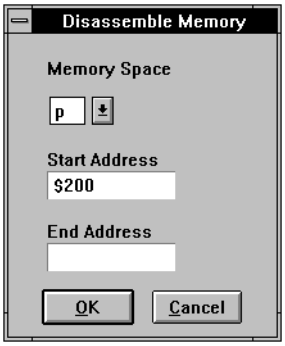


Figure 9-30. DISPLAY//DISASSEMBLE//MEMORY Dialog Box

DISASSEMBLE//MEMORY writes the entire area specified. This could easily be larger than the SESSION window, or even the device buffer. Scroll to view if it is too large for the SESSION window; use DISPLAY//MORE//ON to pause if it is too large for the device buffer. If no end address is specified, the window is filled, but there is no automatic continuation the next time Disassemble Memory is used.

.

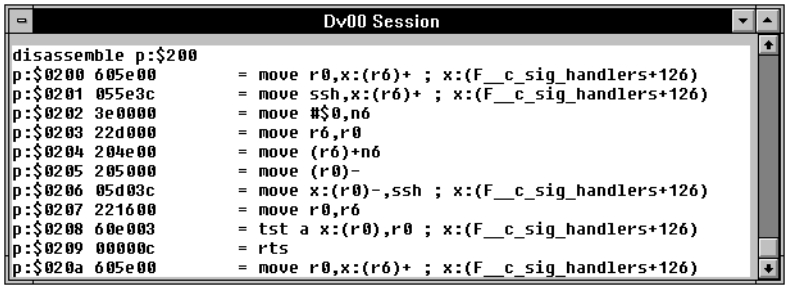
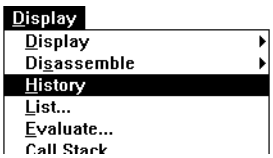


Figure 9-31. DISPLAY//DISASSEMBLE//... Output

9.3.9 DISPLAY//HISTORY



DISPLAY//HISTORY disassembles and displays the last 32 instructions executed. The last instruction displayed is the instruction about to be executed.

This can be useful to determine exactly how the program reached a breakpoint.

If a longer trace is required, see FILE//OUTPUT//OPEN and select HISTORY. This will write a continuous execution trace until closed.

Up to 32 instructions output.

Displayed in order of execution.

Resize or scroll to view.

Last instruction is next to execute.

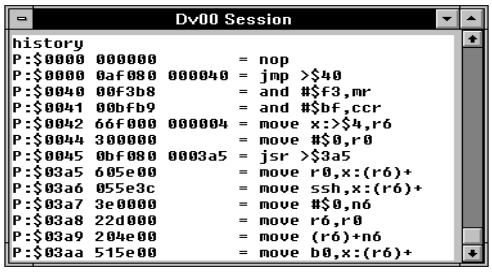
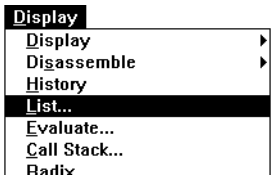


Figure 9-32. DISPLAY//HISTORY Output

9.3.10 DISPLAY//LIST



Displays the source file for the executing program in the SESSION window. As execution proceeds, source display tracks PC.

Step to Next / Previous Page with [APPLY] (1 page = SESSION window size). Revert to PC with Current Page.

If Address is a number, it is interpreted as line number in source file. To specify a memory address, include memory space, as p:$001F.

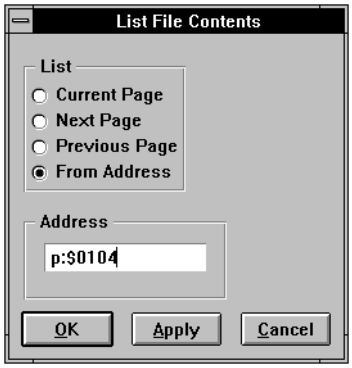


Figure 9-33. DISPLAY//LIST FILE Dialog Box

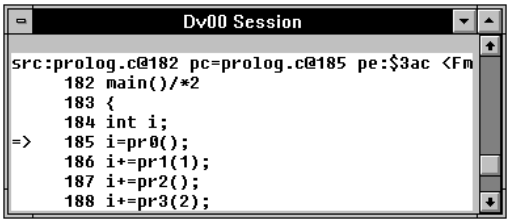


Figure 9-34. DISPLAY//LIST FILE Output

9.3.11 DISPLAY//EVALUATE

Evaluate DSP assembler expressions and C expressions and write the result to the session window.

C expressions display the type of the expression and the value, in the specified format or the normal format for the expression type if ‘All’ is selected.

DSP assembler expressions may be displayed in any type. Selecting ‘All’ gives a selection of interpretations depending on the expression itself.

C expressions are evaluated in the context of the current stack frame by default - that is, the value displayed is that which would have been returned if the expression had been included in the program at the current execution point. C expressions can be evaluated in the context of any of the functions on the call path to the current function. See MODIFY// UP, MODIFY//DOWN, and the CALL STACK window to select an alternative evaluation context.

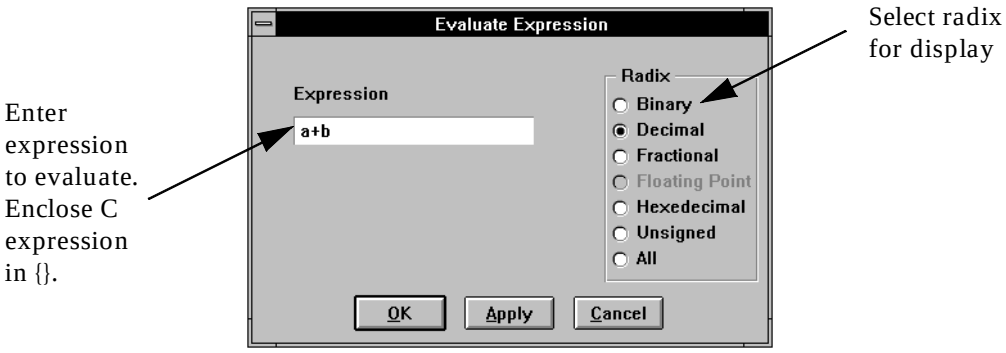


Figure 9-35. DISPLAY//EVALUATE Dialog Box

Expression is echoed, evaluated, and the result displayed.

C expressions in brackets {}.

C expressions display type of expression, but can print in any format.

DSP assembler expressions print in selected format, or 'All' gives a selection depending on the expression.

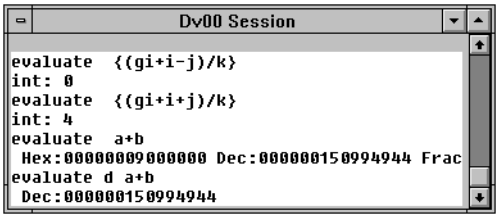


Figure 9-36. DISPLAY//EVALUATE Output

9.3.12 DISPLAY//CALL STACK

Displays summary information about call stack frames. The dialog box initially offers to display the entire call stack; a selection can be made to display only the specified number of innermost or outermost frames.

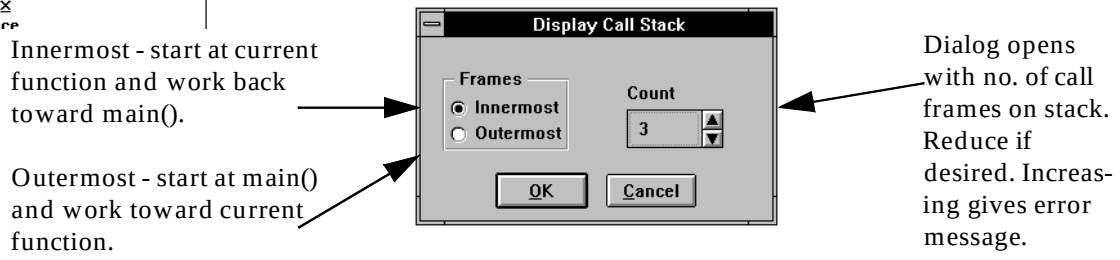


Figure 9-37. DISPLAY//CALL STACK Dialog Box

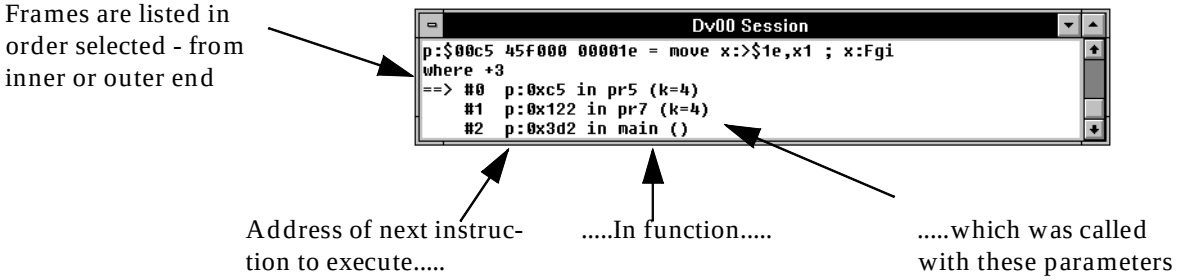
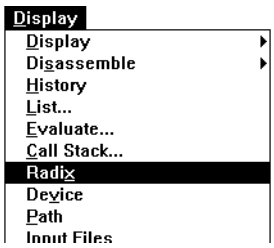


Figure 9-38. DISPLAY//CALL STACK Output

9.3.13 DISPLAY//RADIX



Displays the default radix, used for all numbers input without an explicit radix specifier. This applies whether the number being input is a register or memory contents value, or a memory address. It is not affected by any Display Radix.

The initial default radix is Decimal.

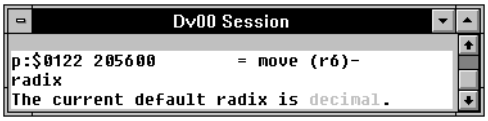
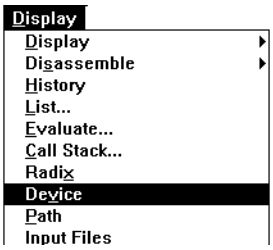


Figure 9-39. DISPLAY//RADIX Output

9.3.14 DISPLAY//DEVICE



Displays the status of each possible DSP device and lists the device types supported.

Configure each device with MODIFY//DEVICE//CONFIGURE.

Status of each possible device is listed.

Supported DSP family members are listed.

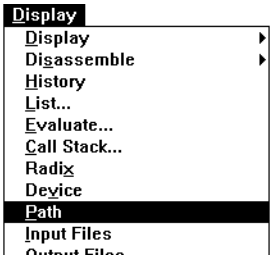
Dv00 Session

```
device
DEVICE STATUS TYPE      DEVICE STATUS TYPE
------ ------ --------  ------ ------ --------
dv00   on     56001     dv16   unused
dv01   unused            dv17   unused
dv02   unused            dv18   unused
dv03   unused            dv19   unused
dv04   on     56004rom  dv20   unused
dv05   on     56000     dv21   unused
dv06   unused            dv22   unused
dv07   unused            dv23   unused
dv08   unused            dv24   unused
dv09   unused            dv25   unused
dv10   unused            dv26   unused
dv11   unused            dv27   unused
dv12   unused            dv28   unused
dv13   unused            dv29   unused
dv14   unused            dv30   unused
dv15   unused            dv31   unused

--Possible device types--
56000
56001
56002
56004
56004rom
56005
56007
56009
```

Figure 9-40. DISPLAY//DEVICE Output

9.3.15 DISPLAY//PATH



Displays the search paths in the SESSION window.

Paths are established with FILE//PATH//SET and FILE//PATH//ADD.

There are two types of path.

The **Working Directory** is the main directory, created with FILE//PATH//SET. It is used as the initial directory for all file chooser boxes. Also, whenever a file is created, and the file name is specified without a directory, the file is created in the working directory.

The **Alternate Source Paths** are only used when opening a file for read access, when a file name is specified without a directory. The working directory is searched first, then each of the alternate source directories in turn.

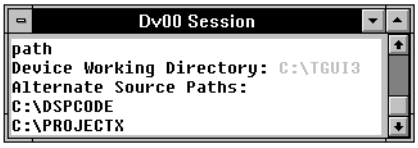


Figure 9-41. DISPLAY//PATH Output

9.3.16 DISPLAY//INPUT FILES

- see DISPLAY//OUTPUT FILES

9.3.17 DISPLAY//OUTPUT FILES

Displays the file assignments for simulated input and output for the current device.

See FILE//INPUT//... and FILE//OUTPUT//... for assignment procedures.

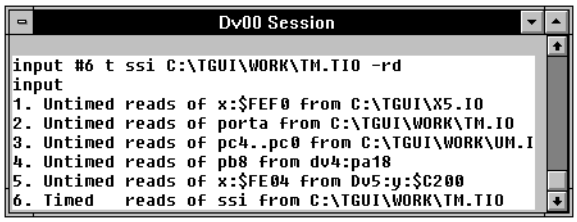


Figure 9-42. DISPLAY//INPUT FILES Output

9.3.18 DISPLAY//REDIRECTED IO STREAMS

- see DISPLAY//IO STREAM STATUS

9.3.19 DISPLAY//IO STREAMS STATUS

IO stream redirection supports stream IO for C programs running on a DSP device. STDIN, STDOUT, and STDERR are supported.

Support may be enabled or disabled (see FILE//IO STREAMS//...), and each of the stream files may be individually assigned to a file on the development host (see FILE//IO REDIRECT//...).

DISPLAY//IO STREAMS STATUS indicates whether stream support is enabled or disabled, DISPLAY//REDIRECTED IO STREAMS lists the stream files and the assignments to files on the host.

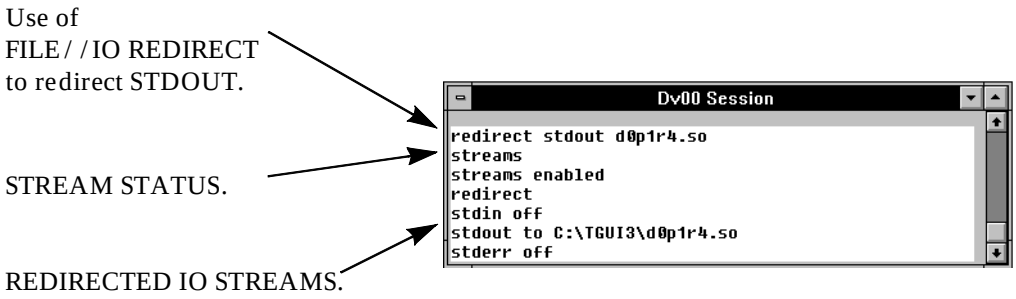


Figure 9-43. DISPLAY//IO STREAMS Output

9.3.20 DISPLAY//LOG FILES

All activity in the COMMAND and SESSION windows may be written to log files. There is only one COMMAND log, but may be a SESSION log for each device. If command activity for different devices is to be logged separately, the old command log must be closed before the command log for the new device can be opened.

DISPLAY//LOG FILES displays a summary of the logging status.

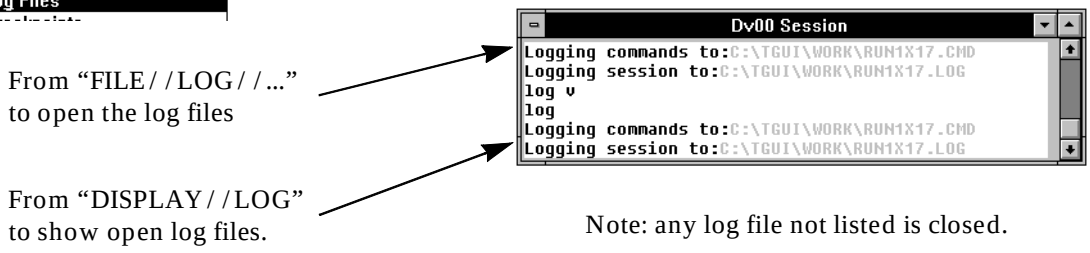
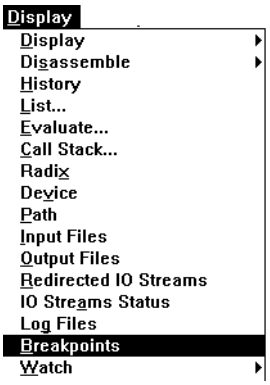


Figure 9-44. DISPLAY//LOG FILES Output

9.3.21 DISPLAY//BREAKPOINTS



Displays all breakpoints set for the current device, listing the breakpoint number, its location, and the action to be performed.

The breakpoint location is listed exactly as entered when the breakpoint was set.

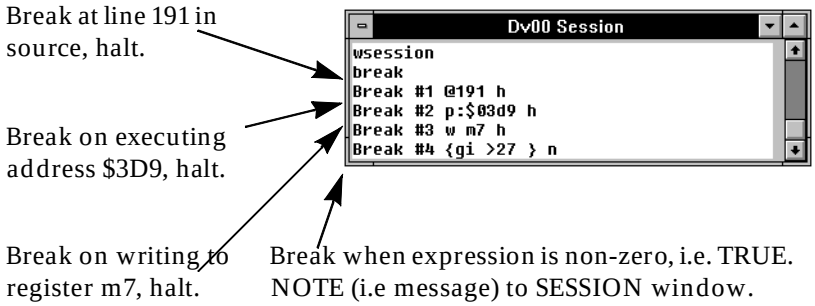
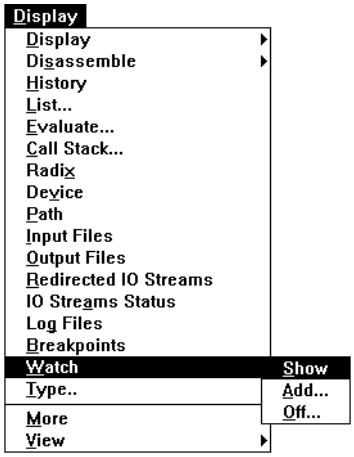


Figure 9-45. DISPLAY//BREAKPOINTS Output

9.3.22 DISPLAY//WATCH//SHOW



DISPLAY//WATCH displays the value of expressions whenever execution is interrupted.

The expression to display is specified with DISPLAY//WATCH//ADD, and may be reviewed with DISPLAY//WATCH//SHOW.

The expression may be specified using register names and assembler labels. If the expression is enclosed in brackets {}, it is interpreted as a C expression, using C variable names. Use MODIFY//UP and //DOWN to navigate the call stack and select the evaluation context for the expressions.

DISPLAY//WATCH//SHOW displays the watch list

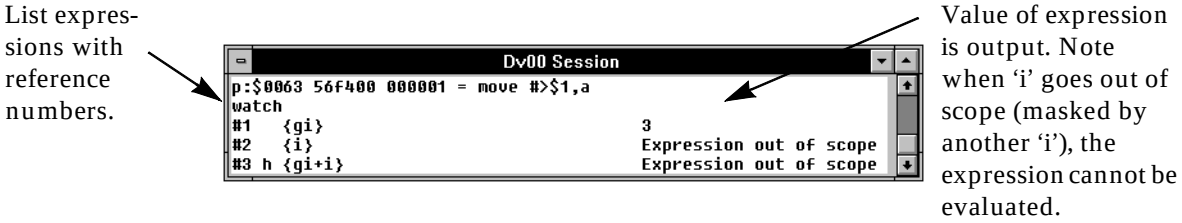


Figure 9-46. DISPLAY//WATCH//SHOW Output

9.3.23 DISPLAY//WATCH//ADD

DISPLAY//WATCH//ADD adds expressions to the watch list. Symbolic references are interpreted as assembler labels and register names, unless the expression is a C expression in brackets {}. The value of the expression is displayed by DISPLAY//WATCH//SHOW, or when execution terminates.

When a C variable goes out of scope, the expression can no longer be evaluated. Use MODIFY//UP and //DOWN to select an evaluation context.

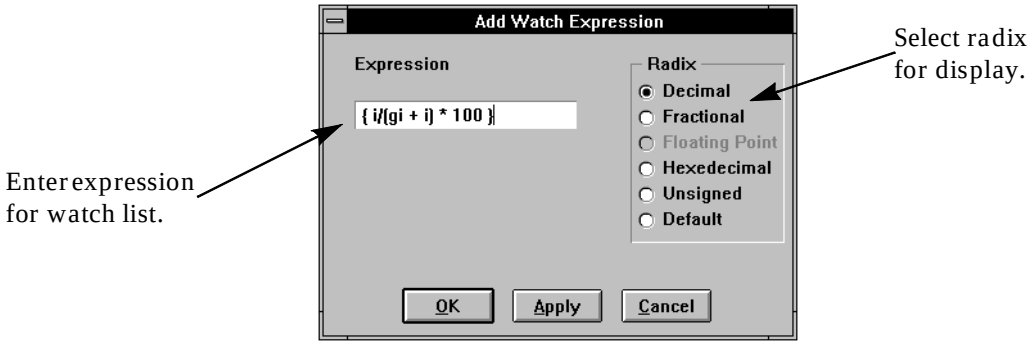


Figure 9-47. DISPLAY//WATCH//ADD Dialog Box

9.3.24 DISPLAY//WATCH//OFF

Removes a DISPLAY//WATCH expression from the list. As the dialog box only lists the reference numbers, it may be helpful to use DISPLAY//WATCH//SHOW first.

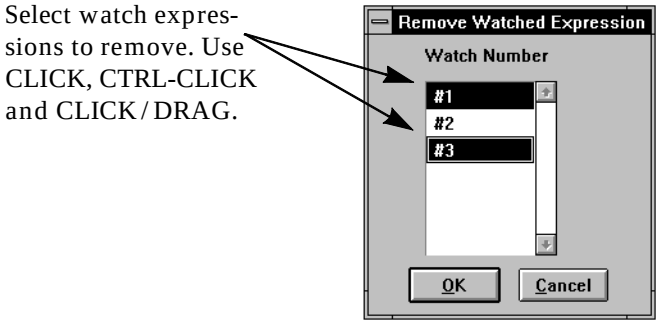
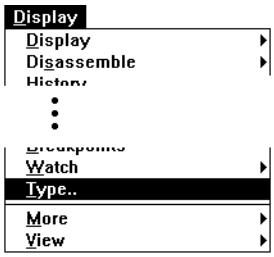


Figure 9-48. DISPLAY//WATCH//OFF Dialog Box

9.3.25 DISPLAY//TYPE



Displays the type of a C variable or expression. Use MODIFY//UP or //DOWN to select the evaluation context.

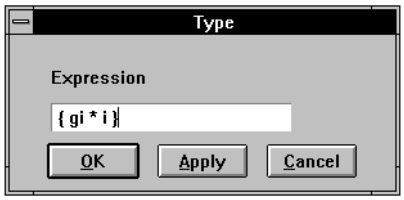


Figure 9-49. DISPLAY//TYPE Dialog Box

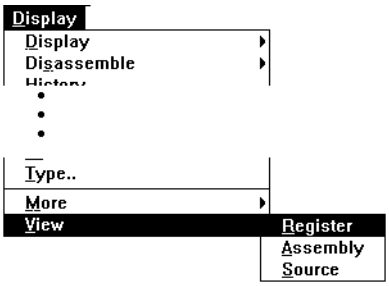
Figure 9-50. DISPLAY//TYPE Output

9.3.26 DISPLAY//MORE

DISPLAY//MORE freezes the SESSION window when it is full until the user responds. Useful when the output from an operation may be longer than the session buffer.

Figure 9-51. DISPLAY//MORE Dialog Box

9.3.27 DISPLAY//VIEW//REGISTER



The DISPLAY//VIEW commands control the type of information displayed in the SESSION window.

REGISTER mode is used to view the output buffer for the current device. This displays the breakpoint memory and register information, commands, error messages, output from the Display menu, etc. This can be considered the normal mode for this window.

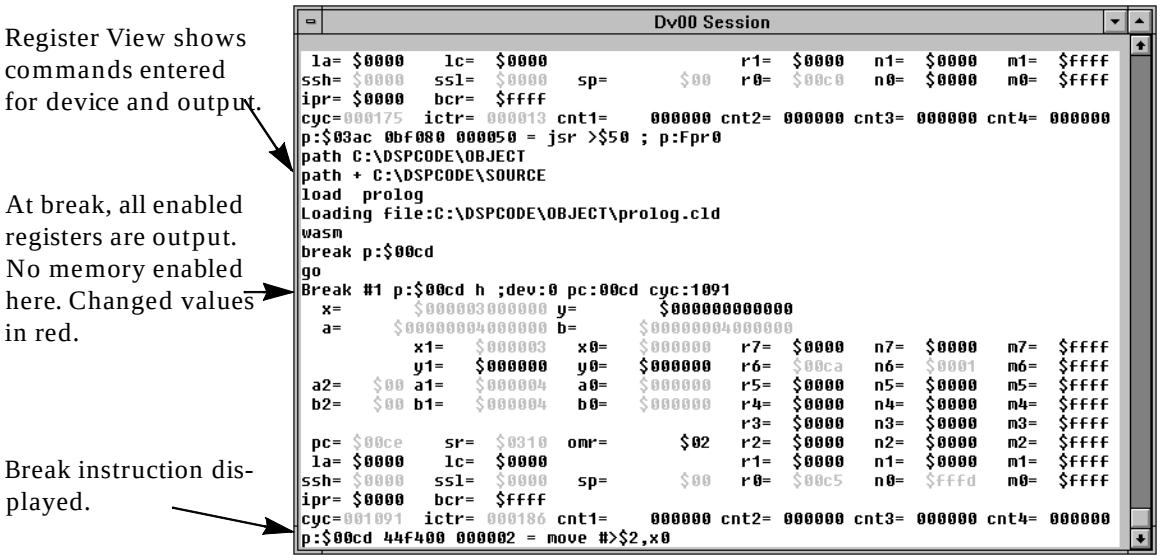
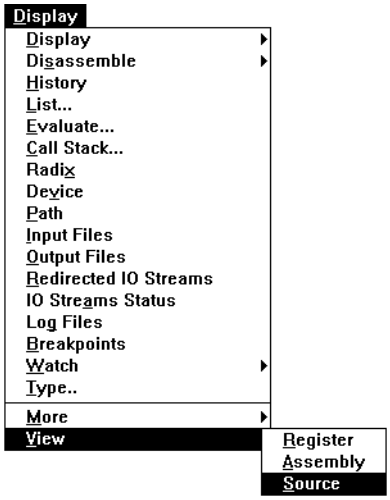


Figure 9-52. SESSION Window - Register View

9.3.28 DISPLAY//VIEW//ASSEMBLY

- See DISPLAY//VIEW//SOURCE

9.3.29 DISPLAY//VIEW//SOURCE



Use the SESSION window to view the ‘p’ memory space as assembly instructions, or to view the program source. The display scrolls to view the entire memory area or source code.

This display does not use the 100-line device output buffer, and is not limited to a scrolling region of 100 lines.

At each break in execution, the window refreshes in the area of the PC, marking the current instruction with the arrow symbol, ‘=>’.

The display is very similar to the ADDRESS and SOURCE windows. However, the SESSION window cannot be used to view, set or clear breakpoints.

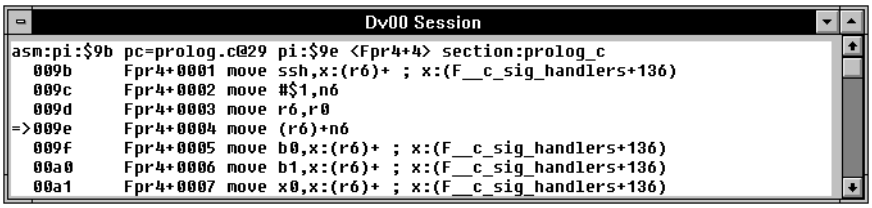
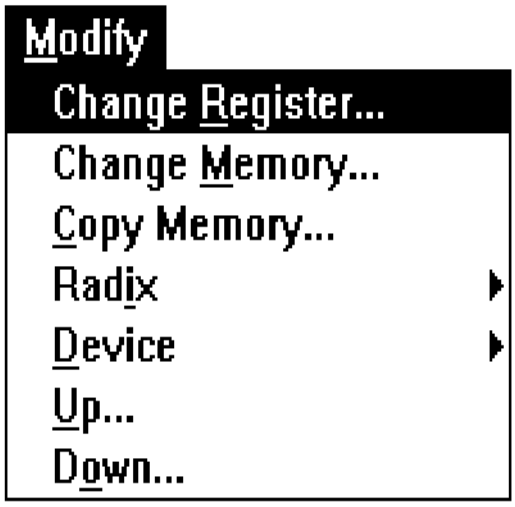


Figure 9-53. SESSION Window, Assembly View

9.4 MODIFY menu



The Modify menu examines and alters many aspects of the development system:

•Change Register: change one or more registers to the same new value.

•Change a single memory location or a block of memory to the same new value.

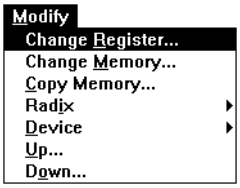
•Copy a single location or a block of memory to another location or block. The destination memory block may but need not be in the same memory space as the source.

•Specify the DEFAULT RADIX and the DISPLAY RADIX. The default radix is used for all input numbers which do not include an explicit radix specifier. The initial default radix is decimal. The display radix specifies how each memory location and register is to be displayed. The initial display radix is hexadecimal.

•Select the current device and set the device type (e.g. set DV05 to be type 56001).

•Select a stack frame from the C call stack as the context for C expression evaluation.

9.4.1 MODIFY//CHANGE REGISTER



MODIFY//CHANGE REGISTER changes the value of one or more registers on the current device.

A dialog box is opened which offers all the registers on the current device in a scrolling list.

Registers may be selected by a single click to select one register, or click-and-drag to select a continuous range of registers. The list scrolls automatically when the dragging reaches either end of the scroll list. Use the control key to add to an existing selection; CTRL-CLICK adds one register, CTRL-CLICK-DRAG adds a range of resisters. Enter a new value in the value field, and click [OK] to change all selected registers.

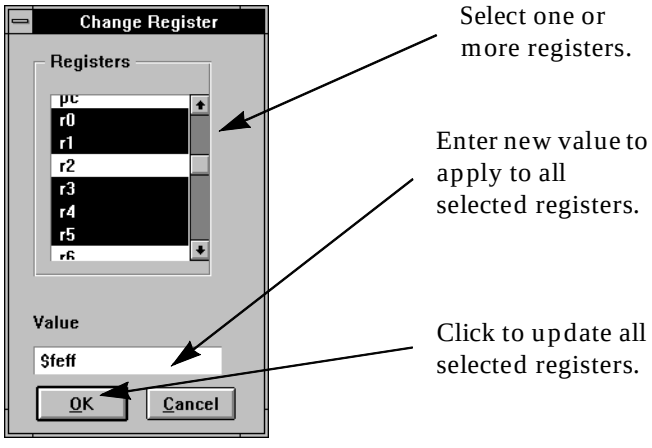
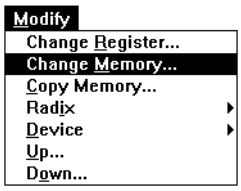


Figure 9-54. MODIFY//CHANGE REGISTER Dialog Box

9.4.2 MODIFY//CHANGE MEMORY



MODIFY//CHANGE MEMORY changes a range of memory locations in one address space on the current device to a new value. All locations are changed to the same value.

Note that addresses are frequently specified in hexadecimal. Use the '$' radix specifier for hexadecimal, or set the default radix to hexadecimal (MODIFY//RADIX//SET DEFAULT).

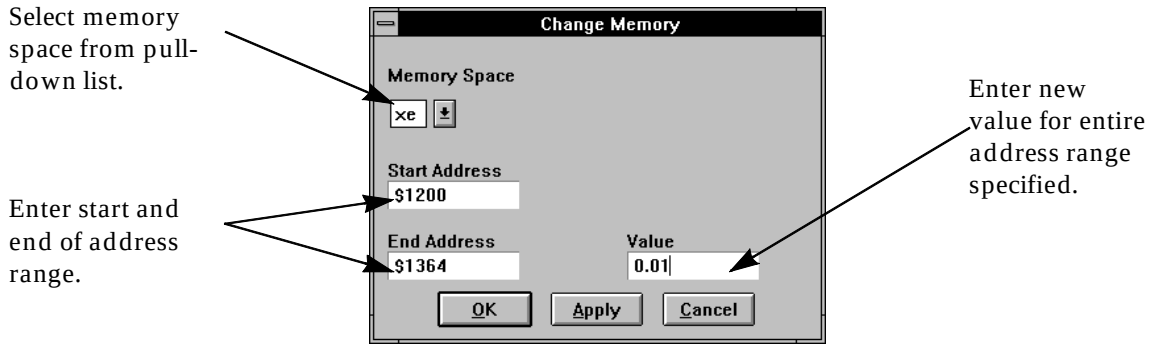
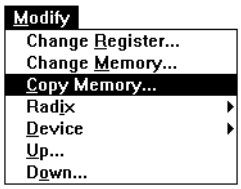


Figure 9-55. MODIFY//CHANGE MEMORY Dialog Box

9.4.3 MODIFY//COPY MEMORY



MODIFY//COPY MEMORY copies one block of memory to another. The source and destination memory maps may, but need not, be the same.

Enter the memory block by selecting the source memory space, and entering the start and end addresses. Enter the destination of the copy with the memory space and start address. The copy will wrap around to the start of memory if it reaches the end.

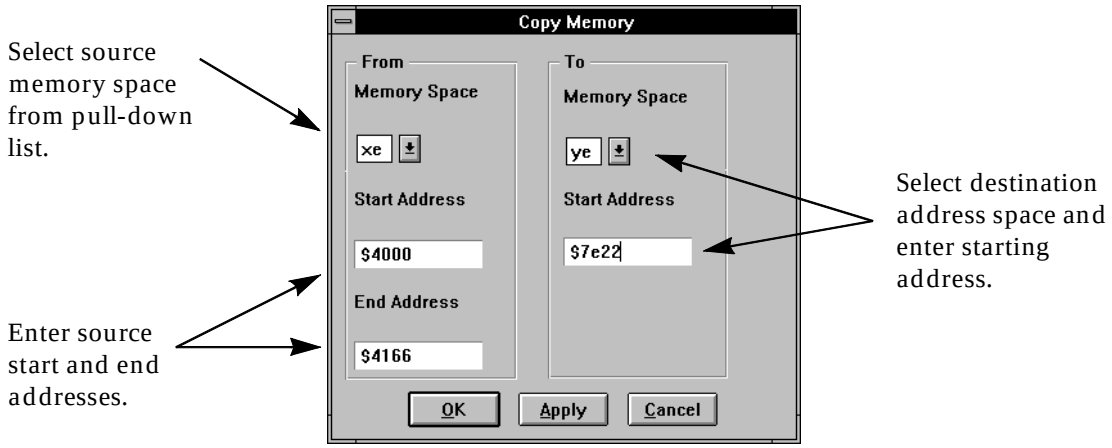
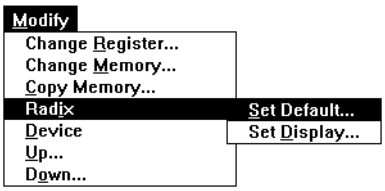


Figure 9-56. MODIFY//COPY MEMORY Dialog Box

9.4.4 MODIFY//RADIX//SET DEFAULT



MODIFY//RADIX//SET DEFAULT specifies the radix used on all input fields unless the input value includes a radix operator. The radix operators are listed in the table below. Note the Radix Operator is used as a prefix to the input value.

The initial default radix is Decimal.

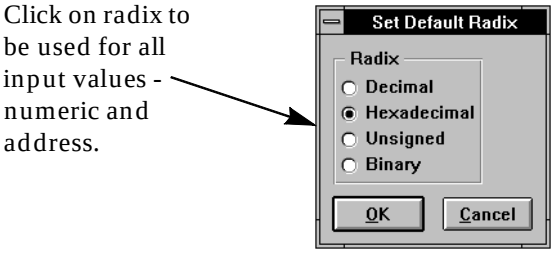
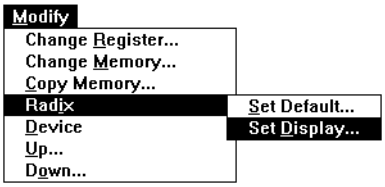


Figure 9-57. MODIFY//RADIX//SET DEFAULT Dialog Box

9.4.5 MODIFY//RADIX//SET DISPLAY



MODIFY//RADIX//SET DISPLAY specifies the radix used when registers or memory locations are displayed. Each register or memory location may have its own display radix. Thus a location which contains a counter may be set to display in decimal, a bitmask may display in binary, etc.

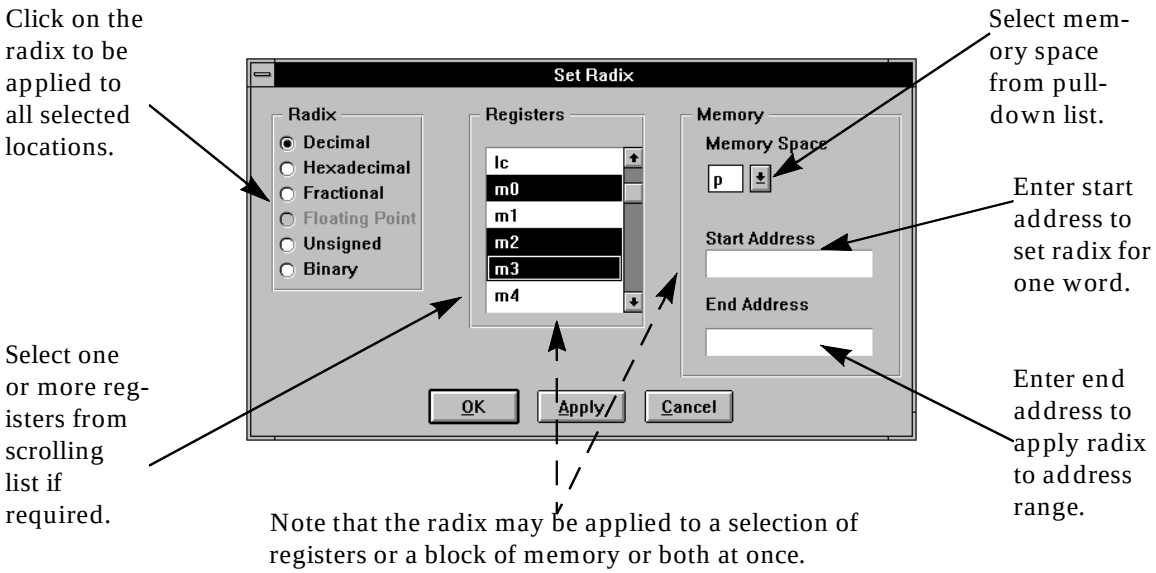
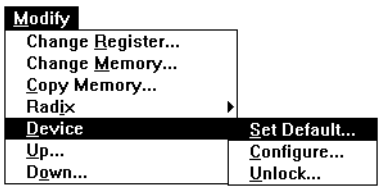


Figure 9-58. MODIFY//RADIX//SET DISPLAY Dialog Box

9.4.6 MODIFY//DEVICE//SET DEFAULT



MODIFY//DEVICE//SET DEFAULT selects a DSP device as the current target device. All device-oriented operations will be applied to this device until another device is selected

.

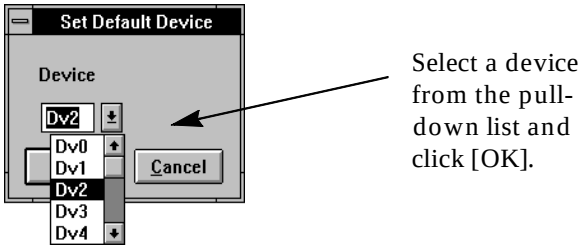
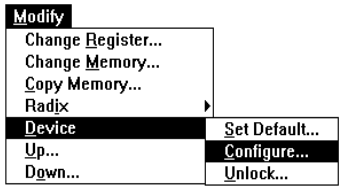


Figure 9-59. MODIFY//DEVICE//SET DEFAULT Dialog Box

9.4.7 MODIFY//DEVICE//CONFIGURE



MODIFY//DEVICE//CONFIGURE allows information to be specified about the DSP devices in use. If a device is not specified, the current default device is assumed:

•TYPE - Which particular member of the DSP family is in use. This automatically adds a device to the system. Initially only Device 0 is considered part of the system.

•ON - Device is turned on, able to execute instructions.

•OFF - Device is temporarily unable to execute instructions. Memory and register contents is retained.

•Remove - Device is no longer considered to be part of the system. All data is lost.

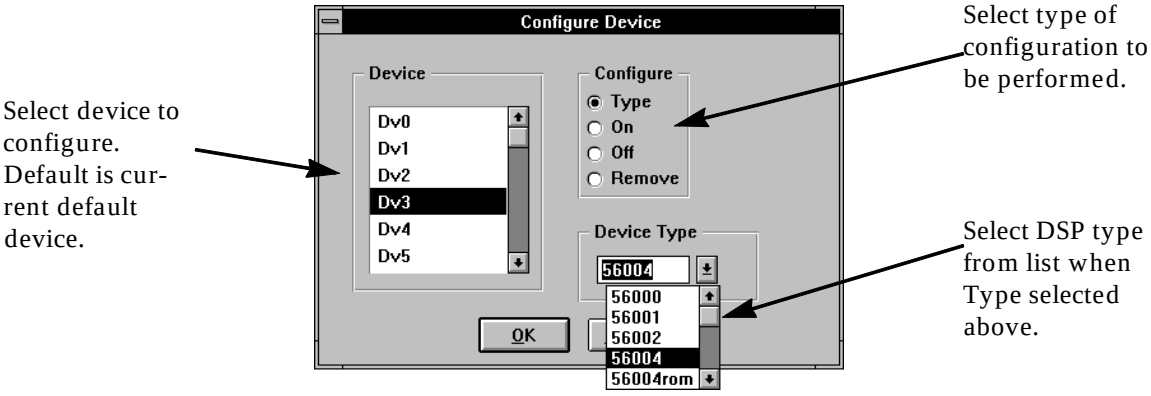
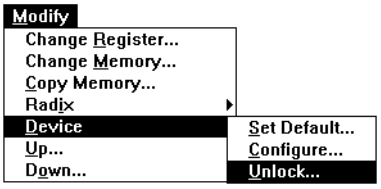


Figure 9-60. MODIFY//DEVICE//CONFIGURE Dialog Box

9.4.8 MODIFY//DEVICE//UNLOCK



The development system may contain hidden device types. A password is required to activate such devices. A password is not required for devices which are not hidden.

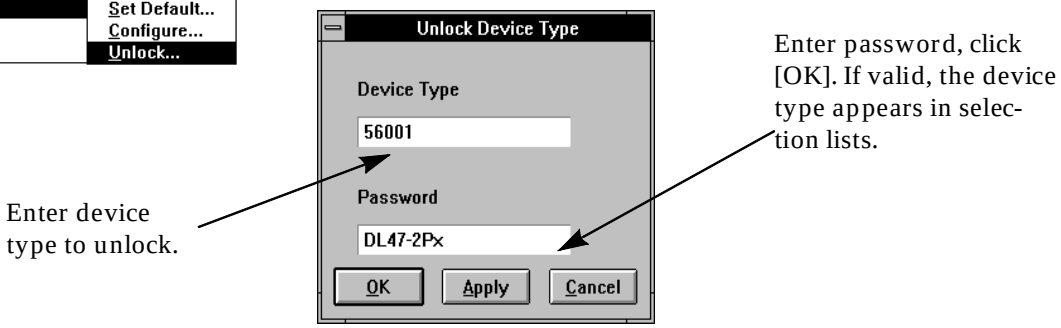
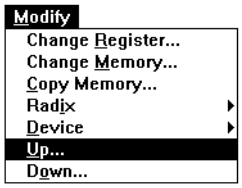


Figure 9-61. MODIFY//DEVICE//UNLOCK Dialog Box

9.4.9 MODIFY//UP, MODIFY//DOWN



Modify UP and DOWN are used to select the context to be used for evaluating C expressions with DISPLAY//EVALUATE, DISPLAY// WATCH, and the WATCH window. The potential problem arises because of the rules of scope for C. Since each function can have its own variable, say 'i14', it may be necessary to specify which function's 'i14' is to be referenced.

As each function is called, a stack frame is created, containing the variables belonging to that function. The stack frame for the current function is stack frame 0, the calling function has frame 1, and so on back to the main program, at frame (say) 7.

DISPLAY//EVALUATE returns the value which would be returned at the current execution point. If a variable in a calling function is required for the expression which is masked by an identical variable in the current function, the required variable is inaccessible. Hence the need to be able to select the required stack frame for the evaluation context.

MODIFY//UP shifts the evaluation context towards the main program by increasing the frame number, MODIFY//DOWN shifts towards the current function by decreasing it.

MODIFY//UP and //DOWN work similarly with the WATCH window and DISPLAY// WATCH. If an expression cannot be evaluated because it is 'Out of Scope' select the original context to evaluate the expression again.

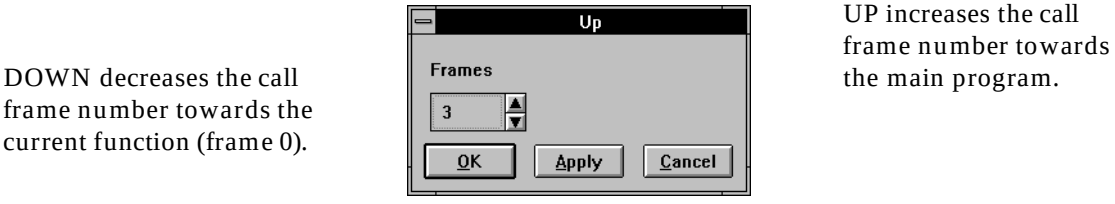
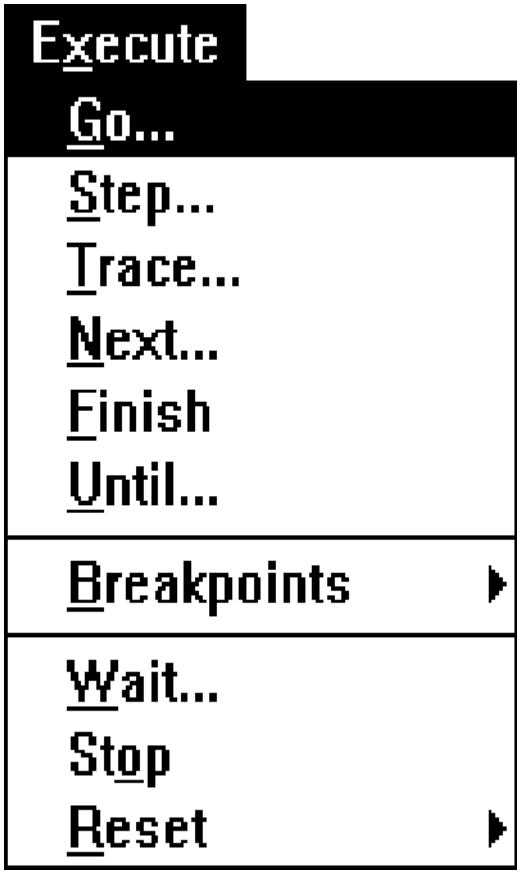


Figure 9-62. MODIFY//UP Dialog Box

9.5 EXECUTE menu



The Execute menu controls the execution of programs on the target device:

•Go lets the program run until a breakpoint or other event interrupts execution. Options are available to specify the execution start address and the way that breakpoints (if set) are to be handled.

•Step executes a specified number of instructions, cycles, or lines of code. If a function call is executed, Step follows the execution through the function.

•Trace executes a specified number of instructions, generating a trace of each instruction executed. After each instruction execution the enabled registers and memory locations are output to the SESSION window.

•Next executes a specified number of instructions or lines of code, skipping over all function calls.

•Finish executes to the end of the current function, terminating after the RTS instruction is executed.

•Until specifies a temporary breakpoint and executes until that (or optionally, any other) breakpoint is met.

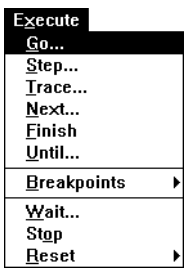
•Breakpoints allows the setting and clearing of breakpoints. A breakpoint is an event (e.g. executing a particular instruction, expression value non-zero) and an action (e.g. increment counter, stop execution).

•Wait pauses, either indefinitely, until a timer has expired, or the user cancels the wait. This is useful in Macro files to freeze the screen for examination.

•Stop stops execution and returns control to the user.

•Reset is used to reset the device registers, to change the mode of a device, or to reset the entire Simulator state.

9.5.1 EXECUTE//GO



EXECUTE//GO opens the GO dialog box to control program execution. There are options controlling the starting address and the way breakpoints (if any have been set) are to be handled. These options are summarized in the illustration below.

The program is allowed to run free from the specified starting point until it is stopped by one of several events. These include user action (EXECUTE/ /STOP, STOP LIGHT button), until the program hits a breakpoint specified to stop program execution, or until the program executes an instruction which ends execution, such as STOP or an illegal instruction).

During execution, the bottom line of the main display shows program progress, with the executing device, the PC, and the cycle count, updated every 1,000 cycles.

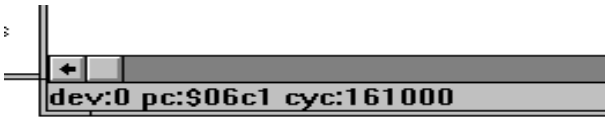


Figure 9-63. Execution Cycle Count Display

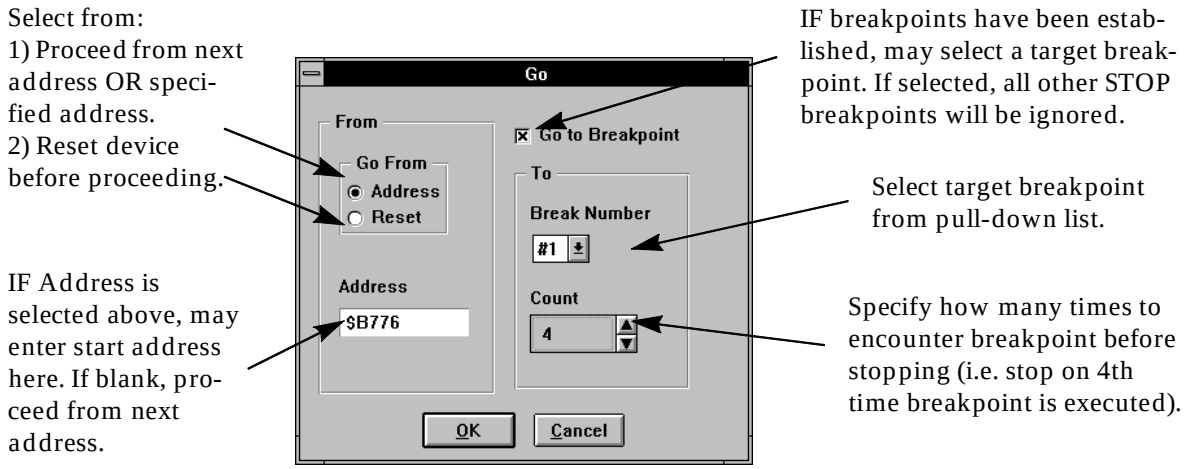


Figure 9-64. EXECUTE//GO Dialog Box

9.5.2 EXECUTE//STEP

- see EXECUTE//TRACE

9.5.3 EXECUTE//NEXT

- see EXECUTE//TRACE

9.5.4 EXECUTE//TRACE

EXECUTE//STEP executes a specified number of cycles, instructions or lines of code. If a function call is encountered, counting of instructions (etc.) will continue during execution of the function.

EXECUTE//NEXT does not offer cycles as an execution option, and called functions are not counted in the execution steps.

EXECUTE//TRACE outputs all enabled registers and memory locations after each instruction execution.

A check box is provided to control breakpoint operation. If left blank, breakpoints will not halt program execution.

At end of execution, the SESSION window displays the values of all registers, memory locations, and expressions which have been enabled (DISPLAY//DISPLAY//REGISTER, DISPLAY//DISPLAY//MEMORY, DISPLAY//WATCH).

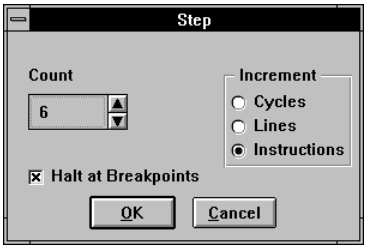
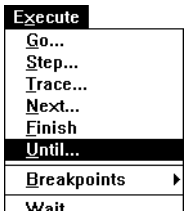


Figure 9-65. EXECUTE//STEP Dialog Box

9.5.5 EXECUTE//UNTIL



EXECUTE//UNTIL executes the program to a specified location. The location may be specified as a program line number, an address, or a label. This sets a temporary breakpoint which is cleared when execution terminates.

Line numbers and labels may only be used if debug information has been loaded from a COFF file (see FILE//LOAD//MEMORY COFF).

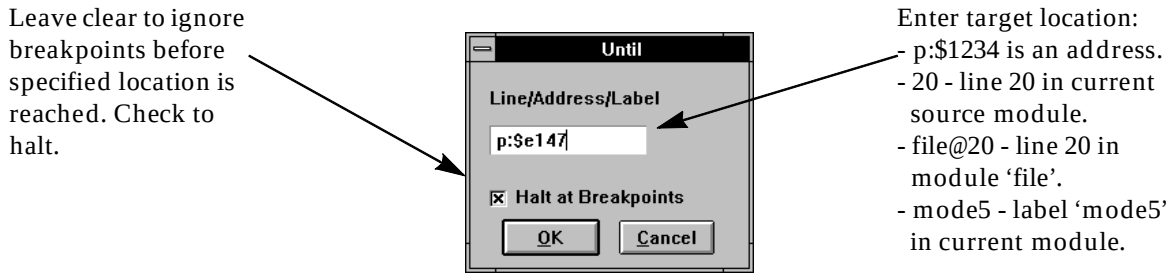


Figure 9-66. EXECUTE//UNTIL Dialog Box

9.5.6 EXECUTE//FINISH

Program executes until the end of the current function. The RTS instruction is executed before execution stops. Breakpoints are handled as normal.

If a function is called during a Finish operation, it executes as normal, but the exit from that function does not end execution.

9.5.7 EXECUTE//BREAKPOINTS//SET

Set breakpoint and specify action to be taken when breakpoint is met.

Available options will vary with DSP type, type of breakpoint and action selected.

Breakpoints are enabled when set, and may be disabled. Breakpoints are listed in the BREAKPOINT window, and are indicated in the ASSEMBLY window with blue highlighting on the address when enabled.

More than one breakpoint may be set on the same location, so that more than one action may be taken.

When the dialog box opens, the first available breakpoint number is offered. Breakpoint numbers do not have to be consecutive, and may be allocated for convenience. For example, it may be convenient to allocate breakpoints so that one function uses breakpoints 1 to 10, another uses 11 to 20, and so on. The BREAKPOINT window (see WINDOWS//BREAKPOINT) will then list all the breakpoints in function A together, etc.

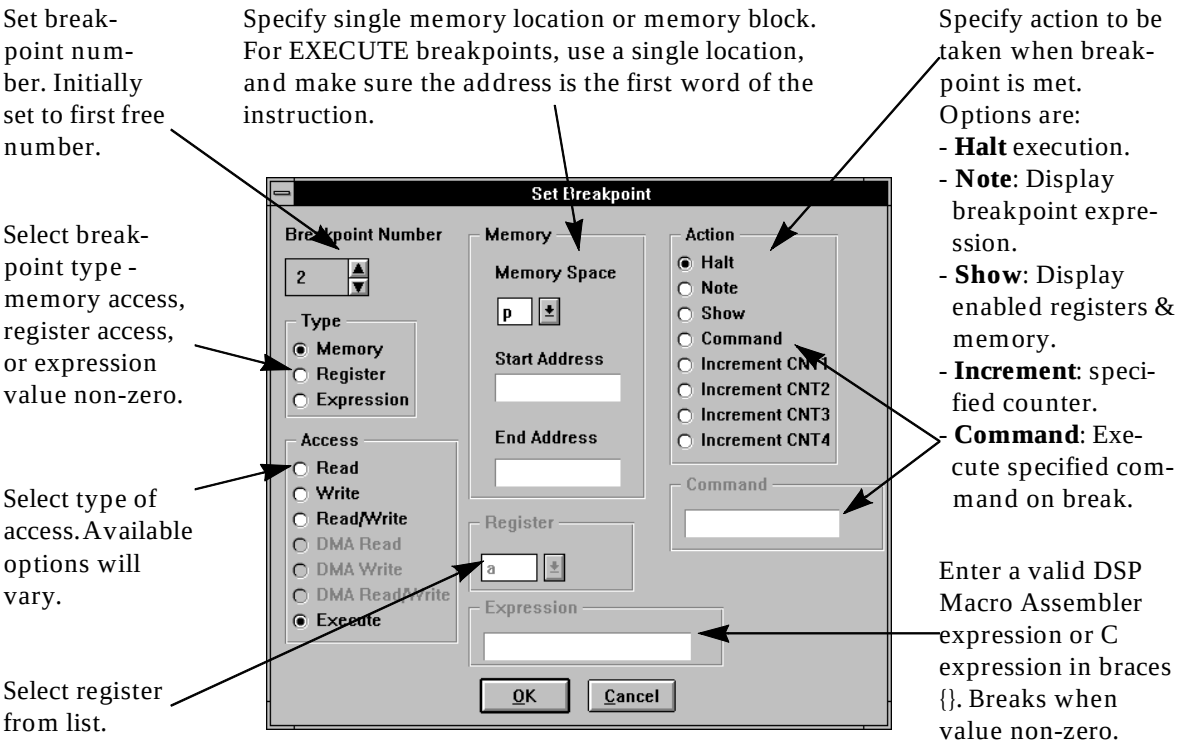


Figure 9-67. EXECUTE//BREAKPOINT//SET Dialog Box

An execute breakpoint location may also be specified by source line number. The source module name may be omitted if there is only one source module.

Figure 9-68. Setting Breakpoint by Line Number

9.5.7.1 Break Processing

During program simulation, breakpoints are checked after each instruction. If a breakpoint condition is found, that is a specified address is accessed in the specified way, or the expression is true, etc., the breakpoint count is checked. If not set, the breakpoint action is taken. If set, and this is the specified occurrence of the breakpoint, the action is taken. Otherwise, program execution continues.

9.5.8 EXECUTE//BREAKPOINTS//CLEAR

Removes a breakpoint. Select the breakpoint or breakpoints from the pull-down list, and click [OK] to clear. CLEARed breakpoints can only be reinstated by recreating with EXECUTE//BREAKPOINT//SET.

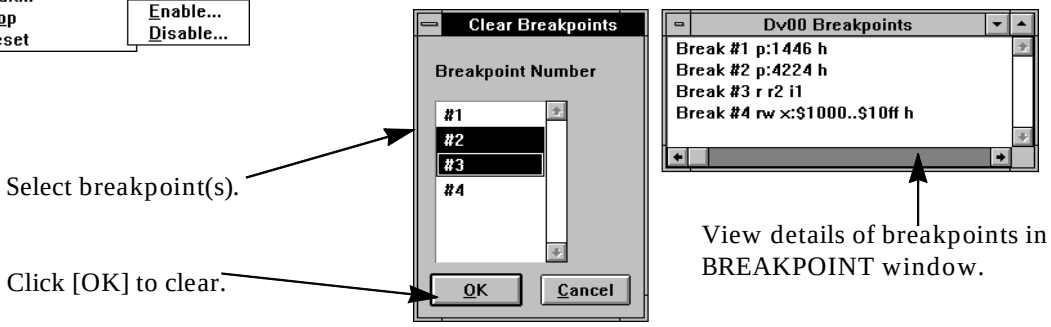
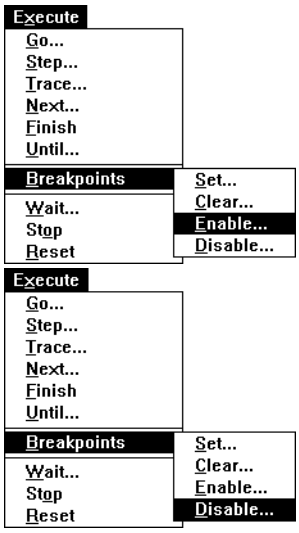


Figure 9-69. EXECUTE//BREAKPOINT//CLEAR Dialog Box

9.5.9 EXECUTE//BREAKPOINTS//ENABLE, DISABLE



Breakpoints may be disabled and enabled. DISABLE temporarily deactivates the selected breakpoints, ENABLE reinstates them.

While disabled, they have no effect on DSP program execution, and do not cause any of the actions associated with the breakpoint.

The dialog boxes for disable and enable are identical in appearance and operation apart from the title. Only the Enable dialog box is shown here.

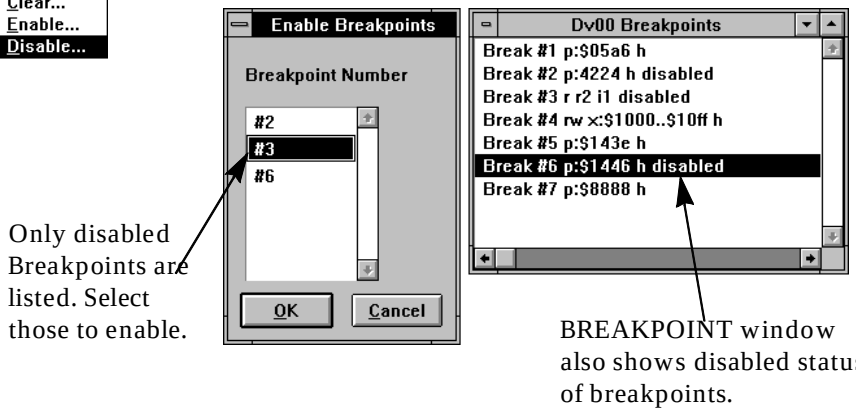
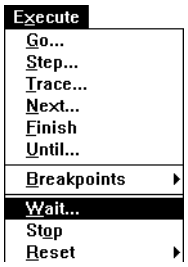


Figure 9-70. EXECUTE//BREAKPOINTS//ENABLE Dialog Box

9.5.10 EXECUTE//WAIT



The WAIT command pauses for a number of seconds, or forever if no count specified. Pause may be ended by pressing the [Cancel] button, or hitting <enter>.

Wait is useful in macro files (FILE//MACRO), where it freezes the display while details are examined.

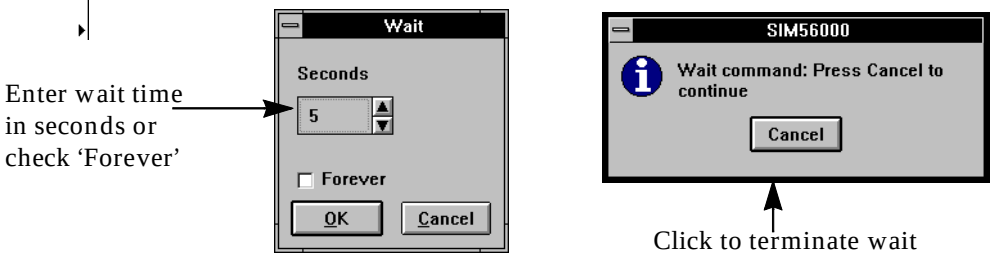
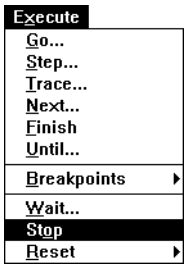


Figure 9-71. EXECUTE//WAIT Dialog Box

9.5.11 EXECUTE//STOP

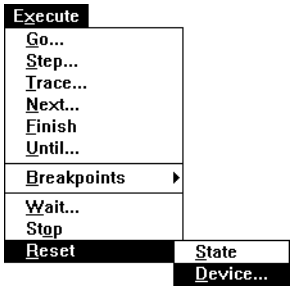


EXECUTE//STOP interrupts execution of the DSP program or macro execution. Control is returned to the user interface.

It may be used to regain control of a program which has failed to reach a breakpoint as expected, is looping or is in some other way running out of control.

Any temporary breakpoint set by EXECUTE//UNTIL is cleared.

9.5.12 EXECUTE//RESET...



EXECUTE//RESET//DEVICE resets the device registers for the current device. In addition, the operating mode for the device may be specified.

The device mode is selected with the radio buttons. Once set to a mode, that mode is the initial state for future reset operations.

This operation is analogous to a device reset caused by the RESET pin. The RESET button in the main window performs a RESET//DEVICE operation, but does not set the device operating mode.

EXECUTE//RESET//STATE resets the entire Simulator state to the start-up condition. Memory is initialized, breakpoints are cleared, and all simulated I/O and logging is stopped.

9.6 WINDOWS menu

The WINDOWS menu provides access to the windows which allow monitoring and control of the development process. These windows display information such as the contents of registers and memory, are updated automatically at each break in execution, and may be moved and re-sized to provide a convenient working environment.

Many of the windows are multi-function, for example the ASSEMBLY window, which displays the code in the vicinity of the PC, permits editing the code with the single-line assembler, and sets and clears breakpoints.

Some windows may be opened many times. With some of the windows, such as the BREAKPOINT window, which lists the breakpoints which have been set in a particular DSP device, a window may be opened for each device. The MEMORY window, however, which displays a block of memory and may be scrolled through the entire address range of the memory space chosen, may be opened as many times as desired to show different memory areas at the same time.

Table 1: Summary of Window Functions

Window	Function	Notes
ASSEMBLY	Display and edit memory contents, set and clear breakpoints, follow program execution.	One per Device
SOURCE	Display source program.	One per Device
REGISTER	Display and modify register contents. Registers arranged in alphabetical order and grouped by peripheral.	Multiple
MEMORY	Display and edit contents of memory. Memory type may be selected, scroll bars access entire range of selected bank of memory.	Multiple
STACK	Display stack contents. Indicates current top of stack. Max 15 entries.	One per Device
CALLS	Display **C** function call stack.	One per Device

Table 1: Summary of Window Functions

Window	Function	Notes
WATCH	Display expressions selected for Watching. Erase with double-click.	Multiple
LIST FILE	Examine any text file.	Multiple
INPUT	Display simulated input assignments.	One per device
OUTPUT	Display simulated output assignments.	One per device
BREAKPOINTS	Display breakpoints set in code. Enable and disable with double-click.	One per Device
COMMAND	Display command history. Retrieve, edit and re-submit commands. Command help. Error message display.	One, shared for all functions
SESSION	Echo commands submitted to Emulator/Simulator and Display output. Each device has its own buffer, only currently selected device is shown.	One, switched between devices and functions
TILE	Arrange open windows in tile pattern.	PC only
CASCADE	Arrange open windows in cascade pattern.	PC only

9.6.1 WINDOW//ASSEMBLY

Opens the ASSEMBLY window for the current device. If it is already open, but hidden or minimized, it is restored and brought to the front.

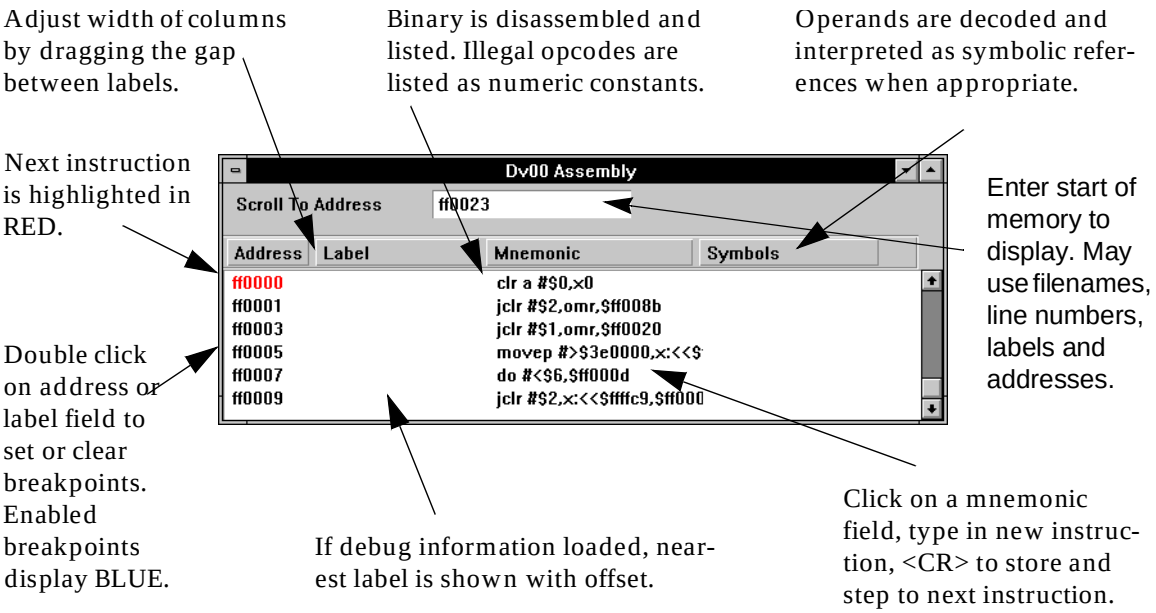
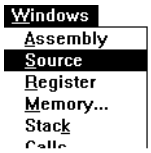


Figure 9-72. ASSEMBLY Window

The ASSEMBLY window displays the memory in the vicinity of the program counter, PC. The scroll bar gives access to the full program memory. As the program executes, the display is updated at each break in execution. The next instruction to be executed is always displayed, highlighted in red.

Breakpoints may be cleared, and Halt breakpoints set by a double-click on an address or label field. Enabled breakpoints are displayed in blue.

9.6.2 WINDOWS//SOURCE



The SOURCE window displays the source code for the executing program. The source code may reside in the directory containing the object module, or any or the directories specified in the path (see FILE//PATH...). The window automatically tracks the PC, displaying the corresponding source line highlighted in red. The scroll bar may be used to scan the whole source file, but the display will revert to the current line with each execution step.

A halt execution breakpoint may be set with the SOURCE window. Double-click on a statement to set or clear the breakpoint. The breakpoint is added to the breakpoint list, displayed in the breakpoint window and highlighted blue in the ASSEMBLY window. The presence of the breakpoint is not indicated in the SOURCE window.

If no source code is available for the executing code, the window shows a message giving the current PC, and indicating that no source is available.

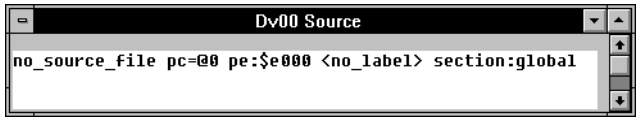


Figure 9-73. SOURCE Window (no source)

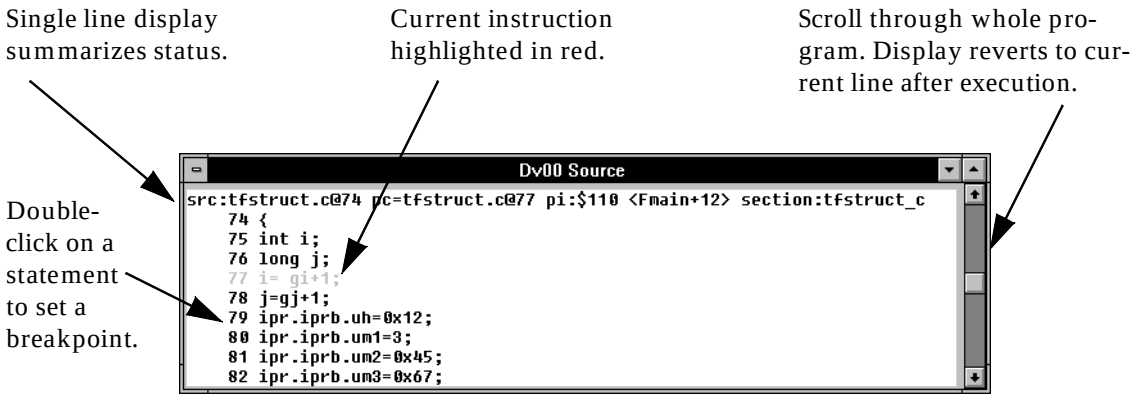


Figure 9-74. SOURCE Window (source file present)

9.6.3 WINDOWS//REGISTER

The REGISTER window displays and modifies a group of registers for the current device. To display registers for another device, first make that the current device and open the REGISTER window. Multiple windows may be opened for each device.

A dialog box allows the selection of the register set to be displayed. Each register window may display the registers for the core or any one peripheral.

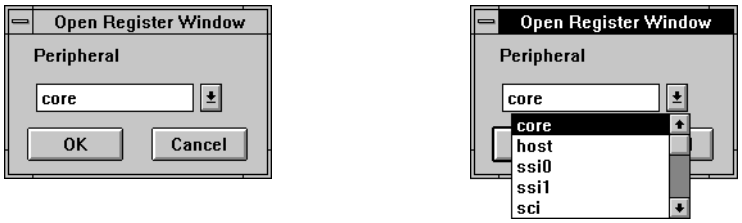


Figure 9-75. Register window peripheral group selection

The selected registers (core registers or registers for the specified peripheral) are arranged in alphabetical order. The window may be resized and scrolled to select which registers are displayed. To display registers which are not conveniently displayed in one window at the same time, open another window and adjust each one to the required range of registers.

The display is updated each time the device enters user mode and returns to debug mode.

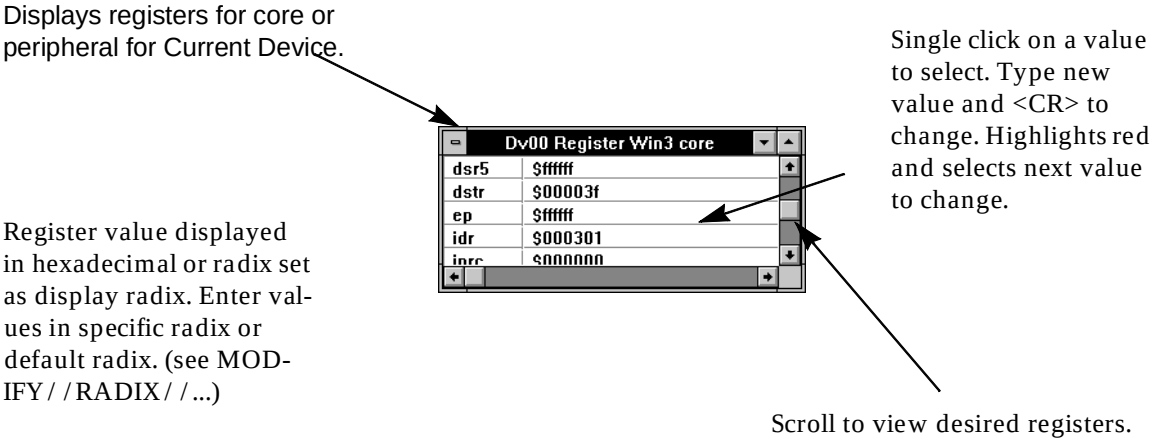


Figure 9-76. REGISTER Window

To change a register, click on it once, type in the new value, and store the value with <CR>. The new value will be displayed in red, and the next register will be highlighted for modification.

9.6.4 WINDOWS//MEMORY

The MEMORY window displays and optionally changes the contents of memory. Each memory window displays a contiguous block of memory from one of the address spaces. Select the address space from pull-down list in the dialog box.

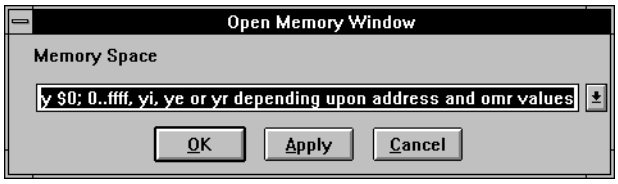


Figure 9-77. WINDOWS//MEMORY Dialog Box

Resize the window to adjust the size of the memory area displayed. The columns in the display adjust automatically to fit the width available. The full range of the memory space selected may be viewed with the scroll bar.

To change memory, click on a location, enter the new value, and store with <CR>. The next location is selected for modification.

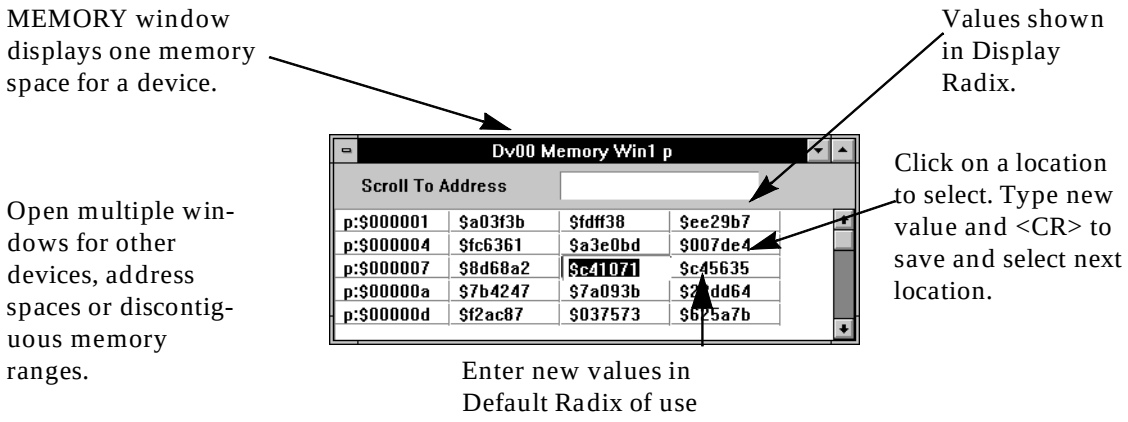
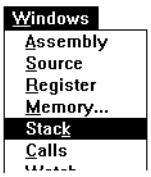


Figure 9-78. MEMORY Window

9.6.5 WINDOWS//STACK



The STACK window displays the hardware stack. May be re-sized and scrolled to view as much or as little as required.

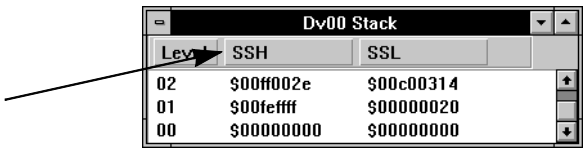
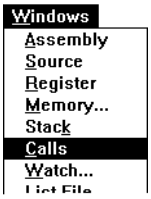


Figure 9-79. STACK Window

The hardware stack is used by the subroutine call instructions, interrupt handling and by some other instructions. In C functions, the return address is put on the stack by the JSR instruction, but then removed and incorporated into the C stack frame. Thus the return address only uses the hardware stack temporarily. Different conventions may be used by assembler programs.

9.6.6 WINDOWS//CALLS



The CALLS window tracks C function calls. Each function call adds another stack frame, each return removes one. Entry #0 is the most nested function, that is the top entry on the stack, the highest number is the main() function.

Each entry has a nesting level number, the PC return address (i.e. the address after the function call), and the name of the function. The top level represents the entry to the debug monitor, and so indicates the next instruction to be executed.

The call stack also indicates the context to use for evaluating C expressions. As each function may have its own copy of a named variable, it may be necessary to indicate which instance is required. A double-click on a stack level selects it as the expression context for DISPLAY//EVALUATE. See also MODIFY//UP and MODIFY//DOWN.

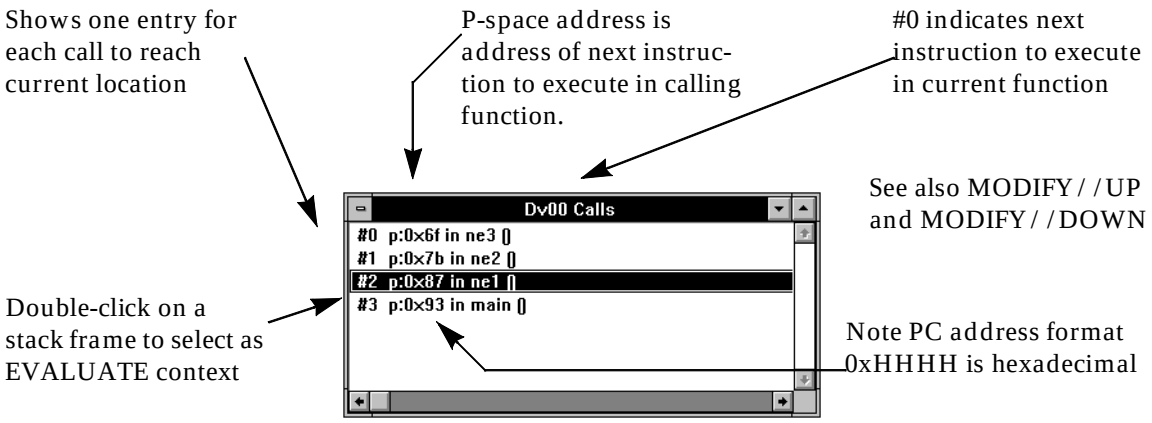


Figure 9-80. CALLS Window

9.6.7 WINDOWS//WATCH

The WATCH window displays the values of any expression. This can be the contents of a memory location or register, or any arbitrary value which need not be calculated during program execution at all. C expressions may be used, enclosed in braces {}.

Symbolic references may be used if symbols have been loaded from the object module.

The values are re-calculated and output at each break in execution.

A C expression which refers to C variables can only be evaluated in the context in which the watch is established - that is while all the variables used in the expression are in scope. So if one (or more) of the variables in an expression goes out of scope (either because a function call or return from a function), the value is replaced with the message “Expression out of scope”. When all elements of the expression are back in scope, the value is again displayed.

An expression which has gone out of scope because of function a call may be evaluated and displayed by selecting the stack frame for the evaluation context. See MODIFY//UP and MODIFY//DOWN. The stack frame assignment remains in effect only until the next instruction is executed. An expression out of scope because of function exit cannot be evaluated until the function is next invoked, as its variables no longer exists.

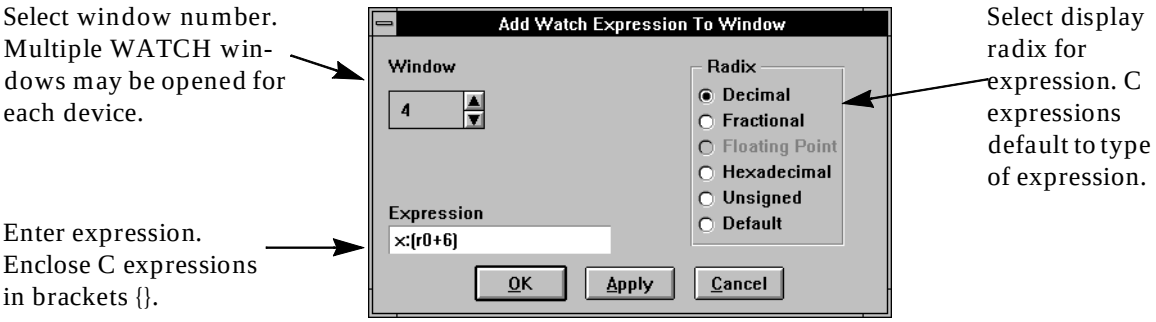


Figure 9-81. WINDOWS//WATCH Dialog Box

9.6.8 WINDOWS//LIST FILE

Views any ASCII file without leaving the development environment.

A standard File Chooser dialog box is opened. Select an ASCII file for viewing. The LIST FILE window is opened, showing the start of the file. The line number appears at the start of each line. The window may be re-sized and scrolled to view the whole file.

Note that the whole file is read when the window is opened, which may take some time with large files.

You may open as many LIST FILE windows as you wish. This may be a convenient way of scanning source files, SESSION window log files (which may be viewed without first closing the log), etc.

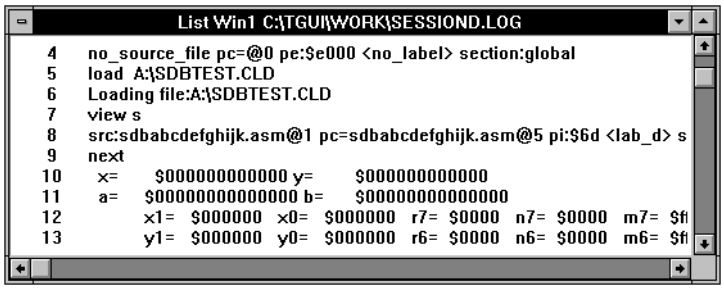


Figure 9-82. LIST FILE Window

9.6.9 WINDOWS//INPUT

Displays all simulated input which has been assigned for the current device. An INPUT window may be opened for each device.

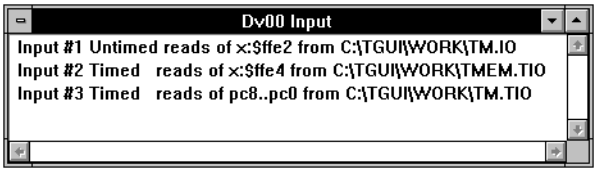


Figure 9-83. INPUT Window

9.6.10 WINDOWS//OUTPUT

Displays all simulated output which has been assigned for the current device. An OUTPUT window may be opened for each device.

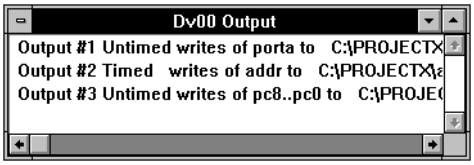


Figure 9-84. OUTPUT Window

9.6.11 WINDOWS//BREAKPOINTS

Displays and enables and disables breakpoints set for the current device. A BREAKPOINT window may be opened for each device.

Breakpoints may be set and cleared by:

- Double-click on ASSEMBLY window address field
- Double-click on source line in SOURCE window
- EXECUTE//BREAKPOINT//SET or //CLEAR menu

Breakpoints set by clicking on the source window are identified by the line number. Breakpoints set by clicking on the ASSEMBLY window have the address. Breakpoints which have been disabled are marked with the word ‘disabled’; all other breakpoints listed are enabled.

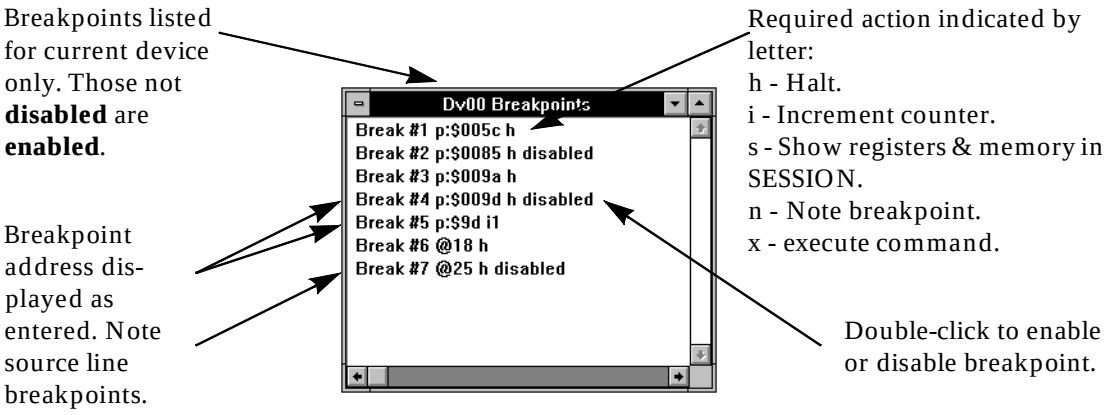


Figure 9-85. BREAKPOINT Window

9.6.12 WINDOWS//COMMAND

The COMMAND window provides the main interface between the user interface and the rest of the system:

- User may enter commands directly.
- All commands generated by the GUI are entered via the COMMAND window.
- The command history is displayed and may be retrieved, edited and re-submitted.
- Summary help is available for all commands.
- Commands executed may be written to a log file - see FILE//LOG//COMMANDS.

The command history buffer holds the last ten commands. If the last command is repeated exactly, the duplicate is not stored. The default size of ten commands may be changed during installation.

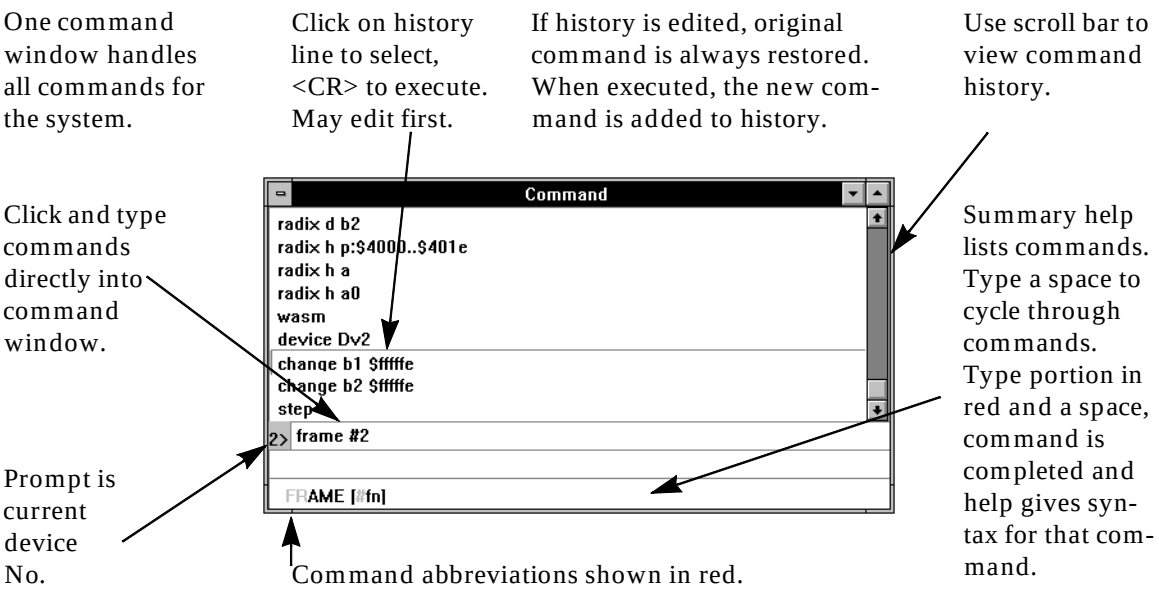
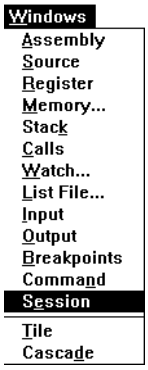


Figure 9-86. COMMAND Window

9.6.13 WINDOWS//SESSION



The SESSION window provides the main output from the development system. The Display menu directs most of its output to the SESSION window, and controls its operation.

Items output to the SESSION window include:

- All commands input via the COMMAND window are echoed.
- All output from commands is displayed.
- Output from many Display menu operations.
- Views of source code and assembly code.
- Registers and memory locations enabled for display at breakpoints and after execution.
- Error messages are sent to the SESSION window.

The last 100 lines written to the SESSION window may be viewed with the scroll bar. The size of this buffer may be set during installation. Some operations may write more than 100 lines to the SESSION window. The Display menu has a MORE feature, which pauses the display every ‘windowful’, allowing the display to be examined, before accepting the next section of output. See DISPLAY//MORE...

There is only one SESSION window, but a separate output buffer for each device. Output from each device is written to its own buffer, but only activity for the current device is displayed in the SESSION window. When another device is made the current device, the SESSION window is refreshed with the buffer for that device.

Output to the session window may be logged to a file - see FILE//LOG//SESSION. A separate log file may be established for each device.

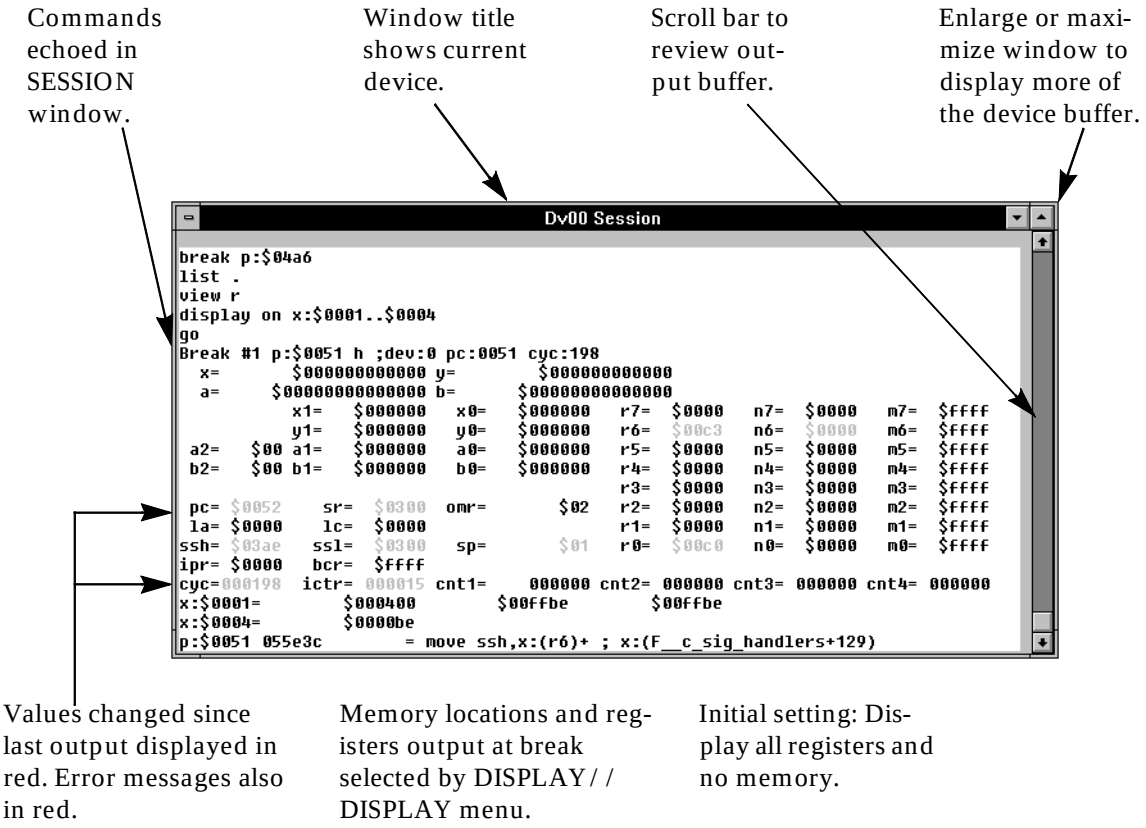


Figure 9-87. SESSION Window

9.6.14 WINDOWS//CASCADE (Windows only)

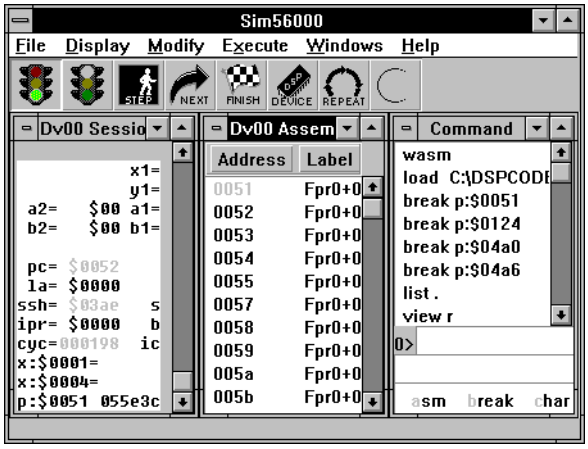
9.6.15 WINDOWS//TILE (Windows only)

The Microsoft Windows environment has two features to arrange windows tidily, Tile and Cascade.

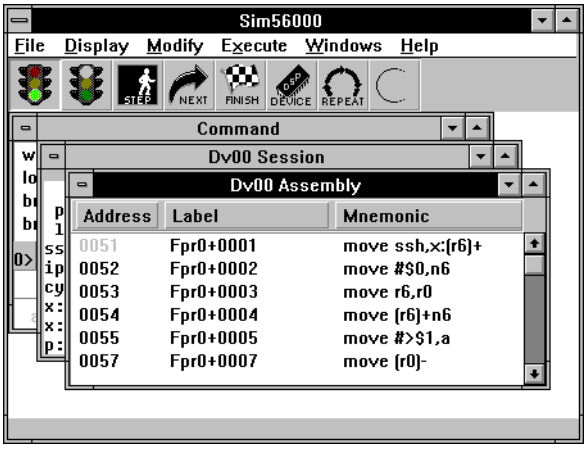
TILE divides the main window into roughly equal areas and places one open window in each tile. All windows are visible, but not all are large enough to be useful.

CASCADE makes all the windows the same size, but usually larger than TILE, and staggers so that the top window can be seen, and the title bar of all other windows is visible.

Both of these techniques simplify the process of locating a window lost on the desktop under other windows.



TILED window



CASCADED window

Figure 9-88. TILED and CASCADED Windows

9.7 THE TOOL BAR

The Tool Bar is located in the main window just below the menu bar. It comprises a number of buttons providing a convenient way of performing frequently-used functions.

9.7.1 GO Button

The GO button starts program execution from the next address. All breakpoints will be acknowledged.

This button is equivalent to EXECUTE//GO from current address, with no target breakpoint.

9.7.2 STOP Button

The STOP button interrupts DSP program execution and returns control to the user. The message ‘SIMULATION ABORTED’ appears in the SESSION window.

The STOP button may also be used to stop execution of a macro command file. A dialog box confirms execution has been terminated.

This button is equivalent to EXECUTE//STOP.

9.7.3 STEP Button

The STEP button executes one execution step. If the source window is open, tracking the program source, STEP executes one line of code. Otherwise, STEP executes one instruction. On encountering a JSR instruction, STEP proceeds with the first instruction of the function, and steps through it.

This button is equivalent to EXECUTE//STEP with a count of 1.

9.7.4 Next Button

The NEXT button executes one execution step. If the source window is open, tracking the program source, NEXT executes one line of code. Otherwise, NEXT executes one instruction. On encountering a JSR instruction, NEXT allows the function to execute, and stops after the RTS instruction.

This button is equivalent to EXECUTE//NEXT with a count of 1.

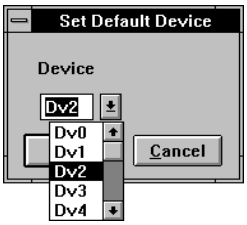
9.7.5 FINISH Button

FINISH allows the current function to execute to completion. Control returns to the user after executing the RTS instruction. It is not affected if another function is encountered during a FINISH operation, execution continues to the end of the current function.

FINISH is equivalent to EXECUTE//FINISH.

9.7.6 Device Button

The DEVICE button opens the ‘Set Default Device’ dialog box. This selects the current default device, to which all commands will be directed until further notice. This button cannot be used to configure, or enable and disable devices.



The DEVICE button is equivalent to MODIFY//DEVICE//SET DEFAULT.

9.7.7 REPEAT Button

The REPEAT button repeats the last command in the history buffer, listed in the COMMAND window.

This button is equivalent to clicking on the last command in the history buffer in the COMMAND window and pressing <CR>.

9.7.8 RESET Button

The RESET button generates a reset command for the current device.

It is equivalent to EXECUTE //RESET, with the device mode unchanged.

INDEX

— A —

— B —

— C —

— D —

— E —

— F —

— G —

— H —

— I —

— J —

— L —

— M —

— N —

— O —

— P —

— Q —

— R —

— S —

— T —

— U —

— V —

— W —