DIGITAL SIGNAL PROCESSOR (DSP)

Application Development System (ADS)
User's Manual

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

This document (and other documents) can be viewed on the World Wide Web at http://www.motorola-dsp.com.

OnCE is a trademark of Motorola, Inc.

© MOTOROLA INC., 1989, 1997

Order this document by **DSPADSUM/AD**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION   1

GENERAL INFORMATION

1.1   INTRODUCTION

The Motorola Application Development System (ADS) is a four component development tool for designing real-time signal processing systems. The ADS consolidates complex hardware and software development tools within a low cost workstation environment using a well supported Operating System. By providing a solid foundation for application development and test, the ADS significantly reduces development costs and time-to-market. The versatile ADS not only allows rapid initial development, but also supports comprehensive testing of prototype designs.

The four ADS components are:

- **Host-Bus Interface Board**—There are two types of boards available: 16-bit ISA bus (for PC-compatibles and HP7xx workstations) and SBus (for Sun and SPARC workstations)

- **Command Converter (CC)**—This is a universal design that supports all ADMs.

- **Control, Development, and Debugging Software**—This software is available in several versions: DOS-compatible (6.0 or later), Windows-compatible (3.1 or later and Windows95), Sun OS-compatible (Rel. 4.1.1 or later), Solaris-compatible (Rel. 2.5 or later), and HPUX-compatible (Ver. 9.x only).

- **Application Development Module (ADM)**—This module supports development and test using a specific DSP chip. Consult your local Motorola distributor, a Motorola semiconductor sales office, or the source for the latest information—the Motorola DSP home page on the Internet (http://www.motorola-dsp.com) to identify currently available ADMs.

Motorola DSPs have a common OnCE™ module that allows the development tools to have identical features. Using the concept of a common serial debug port, one set of tools has been designed which allows the user to communicate with any of the architectures using a single Command Converter. In some Motorola DSPs, this module uses a dedicated OnCE serial port to access the internal module. In other Motorola DSPs, the internal OnCE module is addressed using the IEEE Joint Test Action Group (JTAG) 4-wire Test Access Port (TAP) Boundary Scan Architecture protocol. The tools software can use either the direct OnCE serial port or the JTAG serial port.

This manual describes the installation, use, and functional description of the control system that interacts with the target DSP.

Note: The DSP CLAS Design-In Software Package is a recommended companion product for the ADS. The CLAS software package runs on PC-compatibles, Macintosh (not supported by the ADS), HP7xx Workstation, or a Sun-4 Workstation. The CLAS package includes the DSP Simulator Program and the DSP Macro Cross Assembler Program which are compatible with the ADS user interface program.

1.2 GENERAL DESCRIPTION

The ADS provides a tool for designing, debugging, and evaluating DSP based systems. It consists of three hardware circuit boards, as illustrated in **Figure 1-1**, and two software programs. The hardware circuit boards are the Host-Bus interface, Command Converter, and the ADM. The two software programs are the ADS user interface program which is executed on the host computer and the Command Converter monitor program. **Figure 1-1** illustrates the ADS being used as a hardware evaluation tool or software accelerator. The ADM card has a 14-pin connector which provides an access point for the Command Converter JTAG/OnCE interface.
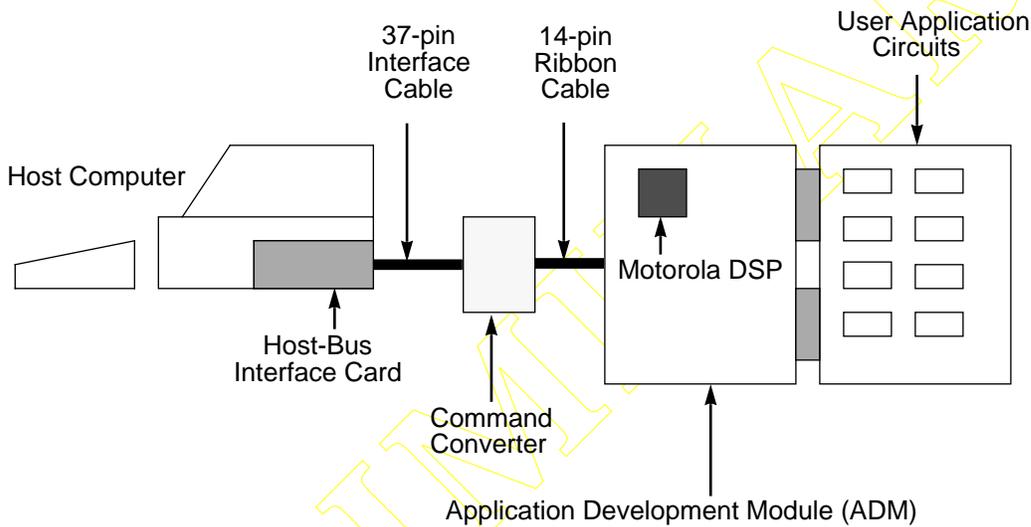
Figure 1-1 Application Development
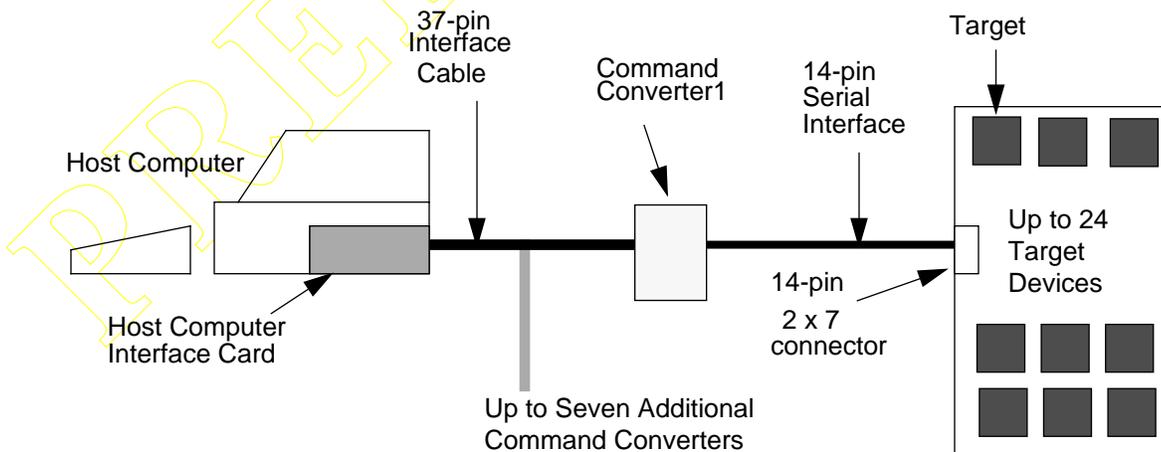
Figure 1-2 Target Circuit Emulation

Figure 1-2 on page 1-4 illustrates how the ADS can be used as an emulator for a defined target system for which the user needs to debug the hardware or software. Here the user must provide an access point on the target hardware for a 14-pin JTAG/OnCE interface cable, which may be as simple as a 2 row × 7 set of test points. **Section 5** provides complete details of the pinout of the JTAG/OnCE interface cable.

The software program provides the routines necessary for the user to communicate with the target DSP on the ADM or the target application. This program has a group of powerful commands, which enables the user to perform a variety of tasks. Operating system command calls may be made from within the program, or temporary exits to the operating system may be made without disturbing current setups to the target DSP.

The format for invoking the non-windowed version of the ADS program is:

```
ADSxxxxx [macro command filename]
```

The format for invoking the Windowed version of the ADS program is:

```
GDSxxxxx [macro command filename]
```

Note: The individual ADM User's Manuals specify which version of the software to load (e.g., DSP56301ADM uses the ADS56300 or GDS56300 software).

The macro command filename (default.cmd extension) is an optional parameter. The macro command file should contain a sequence of commands that the user wishes to execute upon ADS start up and prior to command entry from the keyboard. Macro commands may be nested (a macro command file may call another macro command file) to any level. The macro commands may be easily generated from within the ADS program using the log command. Refer to the **LO**G command for further details. For users who are on a host computer which invokes the ADS using a mouse, it may not be easy to invoke the user interface program with a macro command file. To solve this problem the ADS searches for a specific file named "startup.cmd" in the same directory from which the ADS is invoked. If it finds such a file it is opened as a macro command file and the commands are executed prior to checking for the optional macro command file argument.

The HOST-BUS to ADM interface board provides a physical link between the HOST computer and the ADM via a parallel data and control bus cable. The parallel data path is used for high speed data transfers. The control bus signals enable the HOST computer to reset, interrupt, and send commands to ADMs simultaneously or sequentially.

The ADM is the basic platform for evaluating the DSP. It contains a DSP chip with a JTAG/OnCE interface connector to configure it as a slave to the HOST computer or as a stand-alone unit. In the slave configuration, the user controls the DSP processor and is able to interrogate its status. This enables the user to debug hardware and software easily. In the stand-alone configuration, a user program resident in ROM controls the ADM and may be used as a prototype system for an end product.

1.3   OPERATING ENVIRONMENT

The ADS hardware and software is currently supported on three different host computers:

- PC-compatibles
- Hewlett Packard HP7xx Workstations
- Sun-4 and compatible Workstations

1.3.1   PC-Compatible Requirements

The minimum hardware requirements for the ADS User Interface Program include:

- PC-compatible (486 or Pentium) with 8 M bytes of RAM
- MS-DOS 6.0 (or later), Windows 3.1 (or later), or Windows 95 (or later)
- CD-ROM drive
- Hard drive with 8 M bytes of free space
- Mouse and keyboard
- One 16-bit I/O ISA expansion slot
- Free I/O addresses 100-102 hex, or 200-202 hex, or 300-302 hex.

If the user debug setup involves many assigned disk files, the operating system's limit of the number of open files may be reached. In order to reduce the chance of this situation occurring, it is recommended that your operating system 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 may be open at one time.

1.3.2   Hewlett Packard HP7xx Workstation Requirements

The minimum hardware requirements for the ADSH User Interface Program include:

- HP7xx Workstation running HPUX Version 9.x. (10.x is not supported)
- CD-ROM drive
- Hard drive with 8 M bytes of free space
- Mouse and keyboard
- One EISA expansion slot.

1.3.3   Sun-4 or Compatible Workstation Requirements

The minimum hardware requirements for the ADSF User Interface Program include:

- SUN Operating System Release 4.1.1 or later or SOLARIS Release 2.5 or later
- CD-ROM drive
- Hard drive with 8 M bytes of free space
- Mouse and keyboard
- One SBus expansion slot.

1.4 ADS SOFTWARE FEATURES

- Single/Multiple stepping through DSP programs

- Source level symbolic debug of assembly and C source programs

- Conditional or unconditional software and hardware breakpoints

- Program patching using a Single-Line Assembler

- Session and/or Command Logging for later reference

- Loading and Saving of files to/from ADM Memory

- Macro command definition and execution

- Display Registers and Memory

- Debug commands which support Multiple DSP development

- Hexadecimal/Decimal/Binary/Fractional/Floating Point calculator

- Multiple Input/Output file access from DSP object programs

- On-line help screens for each command and DSP register

- Compatible with the DSP CLAS Assembler & Simulator

- Single Command Converter supports OnCE and JTAG protocols

- Choice of Text-Based or Graphical User Interface

1.5 TEXT-BASED USER INTERFACE

The ADS provides a Text-BasedUser Interface and a Graphical User Interface (GUI). This section describes the Text-Based User Interface. Refer to **Section 4** for detailed information about the GUI.

1.5.1 General Description

The Text-Based User Interface provides the fastest user response time. All command entry occurs from a fixed command line on the screen (third line from the bottom). A fixed error line (second line from the bottom) is used to flag any errors in the command line entry. A fixed help line (last line on bottom of the screen) assists in command line entry by displaying the command's optional or required parameters. Additional help and examples can be viewed by typing a "?" at any point during command entry. As each valid command is accepted from the command line, the command and its results are scrolled into the display screen. The last 100 lines of a DOS display screen entry are available for review at any time by typing Pg Up, Pg Dn, Up-Arrow, or Down-Arrow. The Left-Arrow and Right-Arrow keys allow cursor movement on the command line.

1.5.2 Command Entry

Upon entry into the Text-Base User Interface program, several of the available commands are displayed on the help line. The remaining commands may be reviewed by pressing the SPACE bar when the cursor is at the start of the command line. The user interface program requires a minimum number of key strokes to recognize a command. The minimum number of required characters for each command is shown highlighted on the help line. A command may be specified by typing the required characters followed by a space or by typing the entire command word followed by a space. A detailed description of the commands and command syntax is provided in **Section 3**.

Entering the command key strokes followed by a space activates 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 may 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 a session display. Command execution begins when the user types the Enter or Return key. If the specified command is not predefined, an attempt is made to interpret it as a macro filename. If a macro file of the same name exists, its commands are executed. Macro commands are a convenient way to group a series of commands that might be executed often under one command name.

The ADS user interface software searches for the macro command filename (default.cmd extension). The macro command file is a text file that should contain a sequence of commands that the user wishes to execute. Macro commands may be nested (a macro command file may call another macro command file) to any level. Macro command files may be conveniently created by enabling logging of command entries. This procedure is explained in the documentation of the LOG 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 Return key.

Command line editing is supported for command entry corrections. The cursor may be moved on the command line by using the Left-Arrow and Right-Arrow keys. The Back-Arrow key on the upper right of the keyboard will backspace and delete the previous character. The Del key will delete the following character. The Ins key may be used to toggle between insert and overwrite modes of character entry. The Ctrl-C or Ctrl-Break keys may be used to abort the execution of a display command. The Ctrl-S key allows the current screen output to be frozen for closer examination. The Ctrl-X key acts as a toggle to disable or enable Command Converter service requests. Disabling Command Converter service requests freezes execution of multiple "show" or "note" breakpoints and allows user interface commands to be entered and the display to be examined. Enabling Command Converter service requests resumes Command Converter execution.

1.5.3   Display Modes

The ADS supports three display modes: Register, Assembly, and Source. These modes determine the ADS display at the termination of either the **G**O, **ST**EP, or **T**RACE commands.   The Register display mode causes the display of register and memory locations enabled by the **D**ISPLAY 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 ADS **V**IEW command, or by toggling among the display modes using the Ctrl-W key entry (hold down Ctrl and press w). In addition, ADS commands which display registers or memory, or otherwise create display to the register display window will select the Register display mode; and the ADS **L**OAD and **LI**ST commands will switch from the Register display mode to the Source display mode.

1.6   GETTING STARTED

After following the installation instructions in **Section 2** you should be ready to begin a debug/evaluation session. In order to communicate with the target DSP it must be put into the Debug mode of operation. When first entering the ADS user interface program and the user is just starting a debug session, the user interface program automatically resets the Command Converter.

A good starting reference when initially starting a debug/evaluation session is to reset the Command Converter, user interface program and target DSP into the Debug mode of operation by issuing a FORCE S (system reset) command. If the user exits the user interface program and has put the target DSP into the User mode of operation by issuing a GO command or by toggling the target DSP reset pin, the FORCE B (put the DSP into the Debug mode from the User mode) or FORCE R (put the DSP into the Debug mode from the Reset state) command must be executed so the target is forced into the Debug mode. There are status flags in the ADS user interface program and in the Command Converter monitor that are used to determine whether the target is in a User mode of operation or in the Debug mode ready to communicate with the JTAG/OnCE port. This is necessary so that if the user exits the ADS user interface program and the target is in the Debug mode, the ADS user interface program will initialize its debug status flag correctly by reading the Command Converter monitor debug status flag. For more information on the status flags refer to the **C**FORCE command in **Section 3**.

When power is applied to the ADS system via the 37-pin connector, there are two reset circuits that are activated. The Command Converter has a reset circuit which will put the DSP56002 controller into its monitor program and will wait for commands from the host computer. Any time the power is removed from the target and/or Command Converter the FORCE S command should be executed. This will insure that the user initializes the system into a deterministic state.

Section 4 discusses in detail the functional description of the system interaction between the ADS user interface program, Command Converter monitor program and the target JTAG/OnCE port. A good tip to remember for OnCE-based systems is that whenever the target is put in the Debug mode of operation, its DSO pin ($\overline{\text{DE}}$ pin for newer DSPs) is toggled low. This pin is tied to the $\overline{\text{IRQB}}$ pin of the Command Converter. If power is removed from the target with the 14-pin JTAG/OnCE cable connected to the Command Converter, the DSO line will look like a low signal. This will cause the Command Converter to request service from the host computer.

Note: A good rule is to remove the 14-pin JTAG/OnCE cable prior to powering down the target system when using separate power from the Command Converter.

SECTION 2

PREPARATION AND INSTALLATION

2.1   HOST COMPUTER INTERFACE CARD

This chapter covers the installation instructions for the different host computer platforms. Schematics and bill of materials for the different host computer interface cards can be found in **Section 5**.

2.2   PC-COMPATIBLE TO COMMAND CONVERTER INTERFACE

The interface between the Command Converter and the ADS User Interface Program is handled by a circuit board that resides in one of the PC-compatible motherboard system expansion slots. A single PC-compatible interface card can control up to eight Command Converters.

2.2.1   Installing the PC-Compatible Interface

CAUTION

Before removing or installing any equipment in the PC-compatible computer, turn off the power and disconnect the power cord.

Refer to the appropriate Installation and Setup manual for your PC-compatible for instructions on removing the system cover.

Jumper group JG1 selects the interrupt asserted on the host processor by the Host Interface Card when the target DSP device makes a Service Request (by reaching a breakpoint, for example).

Note: The ADS software does not support interrupts. No jumper should be placed on JG1 when used with the ADS software.

Jumper group JG2 specifies the Host Interface card I/O address. The Host Interface Card supports 16-bit I/O addresses, and uses three consecutive addresses from the specified address. The starting address may be configured, with jumper pairs A8–A15 in JG2, to any multiple of $100, up to $FF00. Place a jumper over a pair of pins to set that

address bit to 0; remove the jumper to set the address bit to 1. Address bits A0–A7 are not decoded. In Figure 2-1, the selected address range is $100–$102.

Note: Although the Host Interface Card supports 16-bit addressing, the ADS software only supports address ranges $1XX, $2XX, and$3XX.

Once you have ensured that the selected address does not confilict with another expansion card installed in the motherboard you may install the Host Interface Card. **Figure 2-1** illustrates the physical locations of JG1 and JG2.
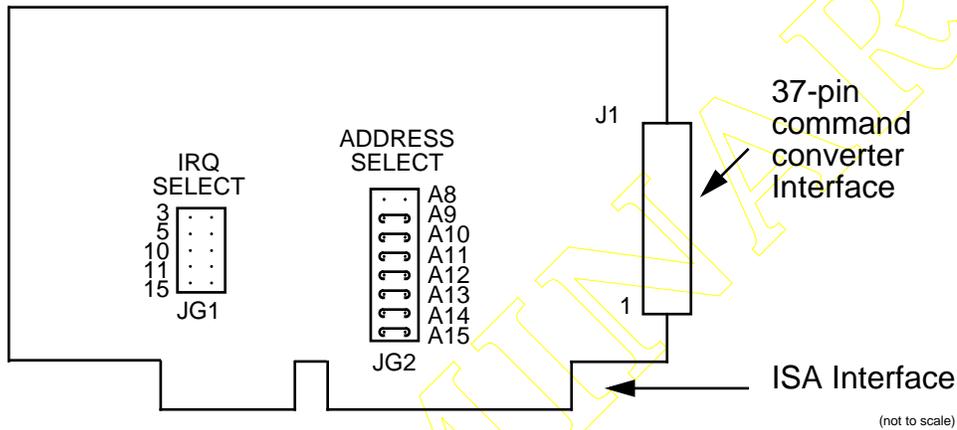
Figure 2-1 PC-compatible Interface Card Jumper Group Locations

The Host Interface Card resides in the PC-compatible I/O bus and is the particular address block with which the ADS User Interface Software communicates. The Host Interface Card address block may be changed to start at one of three addresses as follows:

Table 2-1 PC-compatible I/O Addresses

PC-compatible I/O Address	PC-compatible Peripheral	JG1
100–102 (default)	Undefined	A8 open, all other pairs linked
200–202	Game Port	A9 open, all other pairs linked
300–302	Prototype Port	A8, A9 open, all other pairs linked

If the Host Interface Card address block is changed from the default $1XX_{16}$, the selected address must be specified to the ADS. This may be done in one of three ways:

- Set the environment variable ADMADDR:>SET ADMADDR=200

- Issue the ADS command **host**:0> host io 200

- Use the **-d** option on the ADS command line:>ADS56300 -d 200

To install the Host Interface Card properly, position its front bottom corner in the plastic card guide channel at the front of the PC-compatible chassis. Keeping the top of the Host Interface Card level and any ribbon cables out of the way, lower the card until the card connectors are aligned with the PC-compatible system board expansion slot connectors. Using an evenly distributed pressure, press the Host Interface Card straight down until it seats in the expansion slot.

Secure the Host Interface Card to the PC-compatible chassis using the bracket retaining screw. Refer to the PC-compatible Installation and Setup manual for instructions on reinstalling the cover.

The Host Interface card is factory configured for address decoding at $100–$102 of the PC-compatible I/O Address Map, which are undefined peripheral addresses.

Note: Jumper JG2 should be left disconnected.

The PC-compatible interface card is factory configured for address 100 and no interrupts.

2.3   INSTALLING THE PC-COMPATIBLE SOFTWARE

There are two debugger programs available. A Text-Based User Interface program that provides information in one scrolling window has been used since the inception of the ADS. A Graphical User Interface program (GUI) that supports multiple windows, menus, and many dialog boxes is also available for use in the Microsoft Windows environment.

2.3.1   Defining Environment Variables

The following sections specify environment variables which may need to be defined to establish the correct operating environment for the ADS user interface software. These environment variables may be defined during system startup by adding lines to the file **c:\autoexec.bat**, which applies to all versions of DOS and WINDOWS. Some Windows95™ installations may not use the **autoexec.bat** file. If it does not exist, it may be created and the **SET** commands inserted.

The general form of the **SET** command is:

```
SET  symbolname=value
```

Use the names and values from the sections below, and do not use spaces around the '=' sign.

2.3.2 All Versions of User-Interface Program

If the I/O address of the Host Interface Card is changed from the default setting of $1xx_{16}$, the ADS user interface program must be informed of the address to access the card. This may be done with the environment variable **ADMADDR**:

```
set admaddr=200
```

2.3.3 Text-Based User Interface Program Installation

The ADS DOS user interface program exceeds 300 K bytes in size and dynamically allocates memory. The DOS version of the ADS user interface program uses an extended memory manager, DOS/4GW, supplied with the Watcom 386 C compiler. DOS/4GW is based on Rational Systems' DOS/16M 16-bit Protected-Mode support. This program is called during the ADS user interface program startup, and must be locatable using the DOS PATH environment variable. Refer to the dos4gw.doc file on the Motorola Tools CD.

In almost all cases, DOS/4GW programs can detect the type of machine that is running and automatically choose an appropriate Real mode to Protected mode switch technique. For those few cases in which this default setting does not work the dos4gw.exe program, supplied on the Tools CD, uses the environment variable DOS16M in order to choose an appropriate real- to protected-mode switch technique. In case the default operation does not work on your computer, change the switch mode settings with the following command:

```
set DOS16M=value
```

Do not insert a space between DOS16M and the equal sign. The README file on the Tools CD gives more information on the use of different PC-compatible machines and the value used for those machines. The following procedure shows you how to test the switch mode setting:

1. Before running DOS/4GW applications, check the switch mode setting by running the PMINFO program and note the switch setting reported on the last line of the display. PMINFO.EXE is provided on the Motorola Tools CD. If PMINFO runs, the setting is usable on your machine.

2. If you changed the switch setting, add the new setting to your autoexec.bat file.

3. In order for the virtual memory capability to operate properly, the PC's environment variables must have a defined variable DOS4GVM, with options to define virtual memory parameters. If the DOS4GVM environment variable does not exist, the virtual memory capability does not operate.

The possible parameters are:

- MINMEM—The minimum amount of RAM managed by the VMM. Default is 512 KB.

- MAXMEM—The maximum amount of RAM managed by the VMM. Default is 4 MB.

- SWAPNAME—The swap file name. Default name "DOS4GVM.SWP" on current drive.

- DELETESWAP—Specifies that the swap file should be deleted.

- VIRTUALSIZE—The size of the virtual memory space. Default is 16 MB.

Use the following format for the DOS4GVM environment variable:

```
set DOS4GVM= [option[#value]] [option[#value]]
```

A "#" is used with options that take values since the DOS command shell will not accept "=". As an additional example, the following line in your autoexec.bat file will enable an 8 MB virtual memory swap file with automatic deletion of the swap file:

```
set DOS4GVM=deleteswap maxmem#8192
```

2.3.4 Using Default Settings

If you set DOS4GVM equal to 1, the default parameters are used for all options. In this case the swap file will be called DOS4GVM.SWP and will be given a size of 16 MB. Also note that you should not have to use the DOS16M environment variable for PC compatible 486-based machines. The only line required for the autoexec file is:

```
set DOS4GVM=1
```

2.3.5 Graphical User Interface (GUI) Program Installation

The GUI requires the Microsoft WIN32S to be installed on systems running Microsoft Windows 3.1. The WIN32S software is distributed on the CD. This software is loaded by invoking SETUP on \WIN32S\DISK1.

To install the development software, run SETUP in the WIN directory. See the README file for details.

2.4   SUN 4 TO COMMAND CONVERTER INTERFACE

The Motorola SBus/ADS Interface is an interface board that is designed to be installed in an SBus slot on a Sun SPARCstation or compatible workstation. The board provides a parallel communication path between the workstation and a Motorola DSP development system.

2.4.1   Installing the Sun-4 Interface

The Motorola SBus/ADS Interface is delivered ready to install in your SBus system. There are no user configurable jumpers or hardware configurable options. Please consult the "SPARCstation xxx Installation Guide" or the board installation instructions supplied with your SBus system for installation details. Following is a summary of the Instructions in the Sun manual:

1. Turn off power to the system, but keep the power cord plugged in. Be sure to save all open files and then the following steps should shut down your system:

    ```
    hostname% /bin/su

    Password: mypasswd

    hostname# /usr/etc/halt
    ```

    Wait for the following messages:

    ```
    Syncing file systems... done

    Halted

    Program Terminated

    Type b(boot), c(continue), n(new command mode)
    ```

    When these messages appear, you can safely turn off the power to the system unit.

2. Open the system unit. Be sure to attach a grounding strap to your wrist and to the metal casing of the power supply. Follow the instructions supplied with your system to gain access to the SBus slots.

3. Remove the SBus slot filler panel for the desired slot from the inner surface of the back panel of the system unit. Note that the Motorola SBus/ADS Interface board is a slave only board and thus will function in any available SBus slot.

4. Slide the SBus board at an angle into the back panel of the system unit. Make sure that the mounting plate on the SBus board hooks into the holes on the back panel of the system unit.

5. Push the SBus board against the back panel and align the connector with its mate and gently press the corners of the board to seat the connector firmly.

6. Close the system unit.

7. Connect the 37-pin ADM interface cable to the SBus/ADS Interface board and secure.

8. Turn power on to the system unit and check for proper operation.

2.4.2   Software Installation

Note: In the instructions that follow, ADSxxx represents the name of your particular system, such as ADS56000, ADS96000, etc.

The distribution CD included with the ADS package contain the "mdsp" SBus device driver for the ADS as well as the ADSxxx user interface program for the ADS. The following steps will allow you to install the device driver and run the user interface.

1. Copy all of the software from the distribution CD onto your system using uncompress and tar. See the readme file for details

2. Install the driver with the following commands:

```
hostname# cd adsxxx/driver
hostname# make install
```

3. After a successful installation, you should see a module status message indicating that your module was successfully loaded and giving its ID. To see this status at any time, issue the "modinfo" command on SOLARIS, "modstat" for SunOS.

4. If the ADSxxx driver module was loaded properly, you should be ready to run the ADSxxx user interface:

```
hostname# cd ../bin
hostname# adsxxx
```

If, for any reason, you wish to uninstall the driver, use the following commands:

```
hostname# cd adsxxx/driver
hostname# make unload (SunOS 4.x)
hostname# make uninstall(SOLARIS 2.x)
```

2.5   HP7XX TO COMMAND CONVERTER INTERFACE

The Motorola HP7xx Interface uses the same ISA card which is installed in the PC-compatible. It is controlled by an HPUX device driver. The HP7xx computer must have an ISA slot available to plug this card into. Older versions of the HP700 series computer did not have ISA expansion slots. The ISA board provides a parallel communication path between the HP workstation and a Motorola DSP development system. For details on jumper configurations of the ISA card, refer to **Section 2.2** on page 2-3 of this chapter.

The HP7xx device driver and user interface program support multiple host interface cards in a system. Therefore when reading the software installation instructions keep in mind that the device driver name(s) must be different for each card installed in a system.

2.5.1   Installing the HP-7xx Interface

Before installing one or more ISA cards, a sequence of steps must be followed to shut down your system. If you are using HP VUE carry out steps 1 through 4 of **Section 2.5.1.1**. If you are using the HP-UX command line shell carry out steps 1 through 4 of **Section 2.5.1.2**.

To shut down your computer you must first be logged in as "root". Save all open files prior to shutting down, and always follow the proper shutdown procedure before turning off the power to your workstation. Failure to do so could cause damage to files.

Note: "hostname#" represents the system prompt, (i.e., it is not to be entered as part of the command).

2.5.1.1   HP VUE Shutdown

1. Use the HaltSystem application, located in the System_Admin file of the General toolbox, by double-clicking on its icon.
2. Click on the "OK, Halt System" button to initiate shutdown.
3. When the message "Halted, you may now cycle power." appears, you may safely turn off the power to your workstation.
4. Go to step 1 of **Section 2.5.1.3** on page 2-11.

2.5.1.2   HP-UX COMMAND LINE SHELL SHUTDOWN

1. Change to the root directory with the following command:

        hostname# cd /

2. Enter this command to initiate shutdown:

        hostname# /etc/shutdown -h 0

3. When the message "Halted, you may now cycle power." appears, you may safely turn off the power to your workstation.

4. Go to step 1 of **Section 2.5.1.3**

2.5.1.3   ISA Card Installation

The following steps should be used to install the ISA card:

1. Remove the power cord from both the wall socket and the unit.

2. Remove the power supply cover plate (see Figure 1 below), located on the rear of the unit and marked "TO ACCESS EISA PULL THIS HANDLE", and gently slide the EISA Adapter Card Assembly (hereafter called "assembly") out of the unit.

3. Remove the blank EISA slot cover from the assembly. The cover is simply a piece of metal that covers the hole when there is no card installed, and is held in place by a single screw. Be sure to save this screw as it will be used to secure the card.

4. Carefully slide the ISA card into the assembly, making sure that the connector pins on the assembly meet up properly with those on the card, and that the cable socket is positioned fully within the hole. Use the screw saved from the previous step to securely attach the card to the assembly.

5. Carefully slide the assembly back into the unit. Press firmly on all four corners to ensure that the connectors on the front of the assembly fully engage with those inside the unit.

6. Reinstall the power supply cover plate, and then reconnect the power cord to both the unit and the wall socket. Also connect the 37-pin ribbon cable from the ISA card to the ADM board.

7. The system may now be restarted.

Power supply cover plate

AC Power Connection   EISA Connector hole
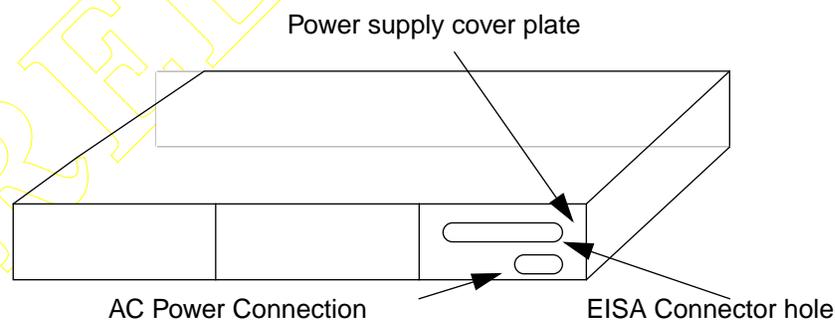
Figure 2-2 HP-7xx Chassis Rear View

2.5.2   Installing the HP-7xx Device Driver

An HP-700 formatted Digital Audio Tape (DAT) contains the necessary files for the installation of the device driver. The device driver supports multiple cards so only one device driver needs to be installed regardless of the number of cards being used. This tape is UNIX tar formatted and should be read in with the following command:

```
tar xvf /dev/rmt/0mn
```

An alternate command would be:

```
tar xvf /dev/rmt/0m
```

The number at the end of the command is the device number and should be changed if the tape drive number is different on the machine being used. The 'n' in the first example keeps the tape from being rewound after it is read. After the files are extracted from the tape a directory named "driver" is created.

The user must be logged in as "root" or superuser to proceed from this point. The following steps should be followed for proper installation of the driver program:

Note: "hostname#" represents the system prompt (i.e., it is not to be entered as part of the command).

The HP device driver for the ADS supports multiple host interface cards (up to three, limited by the addresses that can be selected on the host interface card) installed into a single HP-700 workstation.

1. When the files were installed, a directory called driver was created in the hierarchy. Make this the current directory now by executing the following command, where ADS-PATH is the path to where the ADS files were installed:

   ```
   hostname# cd /ADS-PATH/driver
   ```

2. Copy and change the ownership, group and mode of the device driver library file, libmdsp.a, using the following commands:

   ```
   hostname# cp libmdsp.a /etc/conf

   hostname# chown bin /etc/conf/libmdsp.a

   hostname# chgrp bin /etc/conf/libmdsp.a

   hostname# chmod 444 /etc/conf/libmdsp.a
   ```

3. To install the device configuration file, type the following command (note that it may be necessary to escape the ! by using a backslash \, depending on the shell you are using):

   ```
   hostname# cp  !MOT0010.CFG  /etc/eisa
   ```

   This copies the configuration file into the system EISA configuration directory. Modify the copy of the file in /etc/eisa to reflect the I/O port that the card is configured for (change the PORT entry).

If you are using multiple host interface cards in your HP, make multiple copies of this file in /etc/eisa, changing the last digit of the file name on each copy. Modify each copy to have the correct ID and PORT entries (the ID entry should be the name of the file without the .CFG extension). Each PORT entry should be modified to contain the proper addresses for the additional cards that are supported.

4. Run the EISA configuration program by typing:

   ```
   hostname# /etc/eisa_config
   ```

   At the EISA prompt, type "add !MOT0010.CFG <slot num>", where <slot num> is the slot number in which the card is installed.

   If you have installed multiple host interface cards, repeat the "add" command once for each configuration file and slot. For example, if you created a file !MOT0011.CFG for a card in slot 2, and modified the PORT and ID entries in !MOT0011.CFG to reflect the I/O address of the card in slot 2, you would type "add !MOT0011.CFG 2" to add that card to the EISA configuration information.

   On machines with one EISA expansion slot, the slot number is 1.

   Press "q" to quit the EISA configuration program, and "s" to save.

5. You must now edit the /etc/master file to include the ADS device driver. Make a backup copy of "/etc/master". You will need to change one line and add three new lines in this file. The entries will be in the "Third Party and User Drivers", the alias table, the driver/library table, and the library table.

   Change the first available line of the `Third Party and User Drivers' section to look like the following line. An available line is one which has dashes in the first four entries. Note that the last two entries are already set and are not to be changed. The last entry is the major number for the device. You may use a line which has 38, 39, 40, 42, or 43 for the major number. Make note of the number you choose, as it is used in the next step.

   ```
   Name       Handle     Type     Mask      Block     Char (Do not add this line.)
   mdsp       mdsp       1        1FA       -1        <major>
   ```

   Find the alias table add the following line:

   ```
   mdsp       mdsp
   ```

   Find the driver/library table add the following line:

   ```
   mdsp       libmdsp.a
   ```

   Find the library table add the following line:

   ```
   libmdsp.a  0
   ```

6. Make a device file for the driver by executing the following:

```
hostname# mknod  /dev/mdsp0 c  <major> <minor>
```

The major and minor numbers of the device are given by <major> and <minor>, respectively. N is a user chosen device group number. The ADS user interface software always defaults to device group 0 or mdsp0.

Note: The <major> must be the same major number you used in the previous step. The minor number is 0x4S0000, where 'S' should be replaced by the card slot number in which the card was placed (1 – n). For machines with a single slot, the slot number is 1. For example, slot 4 of a multi-slot machine would be 0x440000

Be sure to change the permissions for this file(s) by typing this command:

```
hostname# chmod  666  /dev/mdsp0
```

If you have installed multiple host interface cards, repeat the ``mknod'' and ``chmod'' steps for each new card. The <major> number for each will be the same, but the <minor> number will reflect the slot in which the card was installed. The device file names are arbitrary, but you should make note of which card corresponds to each device file name. We suggest using a final digit of `N – 1' for slot `N'.

7. Edit the file /usr/sam/lib/kc/drivers.tx and add an entry for the ADS device driver. You may wish to make a backup of the file before editing it. Add the line:

```
mdsp:::Out:Motorola ADS Host Interface Card
```

8. Select and edit a dfile (configuration description file). First, change directories by executing:

```
hostname# cd  /etc/conf
```

A dfile must now be selected. Your current system dfile should be called either dfile, dfile.SAM (for a kernel that has been configured with the HP System Administration Manager), or a unique name given by you if you have altered your kernel configuration by hand. Use the dfile appropriate for your system. If you aren't sure which version to use, or if your choice doesn't work, then you can use the file created by the command

```
hostname# /system/TOOL/get_kdfile  /hp-ux > dfile.current
```

to get your current system dfile.

Make a backup copy of the selected dfile and edit the selected dfile by adding the following lines to the top of it:

```
* Motorola DSP ADS Device Driver

mdsp
```

9. Generate the files needed to rebuild the kernel with your new device driver. Execute this command, replacing dfile-name with the dfile you selected in the previous step

```
hostname# /etc/config dfile-name
```

10. Generate a new kernal object file in the current directory using the following command:

```
hostname# make  -f  config.mk  XOBJS=libmdsp.a
```

11. If the build finished successfully, your new kernel may now be installed. Make a backup of the current kernel by executing the following copy command:

```
hostname# cp  /hp-ux  /hp-ux.pre-ads
```

12. The final step is to Install the new kernel in the root path and reboot the system. Note the period before the first slash in the first argument of the cp command:

```
hostname# cp  ./hp-ux  /hp-ux
```

To reboot the system using the new kernel type the following command:

```
hostname# exec  reboot
```

During boot-up, you should see the following message displayed on the screen if the installation was successful:

```
"Slot  <slot-num>: Motorola DSP ADS Host Interface Card Initialized",
```

where <slot-num> is the previously selected slot number in which the card is installed.

The ADS Device Driver is now installed and ready for use. By default, the ADS program attempts to open the device file dev/mdsp0. If you want to use a different device (i.e., different host interface card), you can specify the device file name in an environment variable, or on the command line. The environment variable is ADMADDR. The command line option is `-d', followed by the name of the device file. The `-d' option must come before a command file name.

For example, to use /dev/mdsp2, you could set an environment variable before invoking the ADS software using the following statements:

```
# setenv ADMADDR /dev/mdsp2
```

```
# adsXXXXX [command-file] (where XXXXX is the device)
```

or you can invoke the ADS software with a command line argument such as:

```
# adsXXXXX -d /dev/mdsp2 [command-file]
```

2.6 CONFIGURING THE COMMAND CONVERTER

The universal Command Converter supports both the OnCE and JTAG serial protocols. The universal Command Converter may be identified by the surface-mount DSP56002 which controls its operation. The monitor program resides in SRAM, and is downloaded by the ADS software during Command Converter initialization and reset operations.

The Universal Command Converter has two user-configurable jumper groups, **JG2** which selects the device number, and **JG3** which selects the power source for the OnCE/JTAG buffers.

2.6.1 Selecting the Command Converter Device Number

The Command Converter JG2 jumper group selects which device that particular Command Converter will respond to commands from the user interface program. The following table describes the device address select option:

Table 2-2 Command Converters Rev 4, 5 Device Number Selection

Device Address	JG2
0 (default)	1–2, 3–4, 5–6
1	1–2, 3–4
2	1–2, 5–6
3	1–2
4	3–4, 5–6
5	3–4
6	5–6
7	no jumpers

Note: All Command Converters are factory configured for device address 0.

2.6.2 JTAG/OnCE Port Buffer V_{CC}

In order to provide support for low voltage DSPs, a CMOS buffer exists between the DSP56002 controller and the target OnCE/JTAG interface cable. This buffer has its Vcc pin connected to **JG3, Pin 2**. See **Table 2-3** for CMOS buffer V_{CC} configuration.

Table 2-3 CMOS BUFFER V_{CC} CONFIGURATION

V_{CC} Source	JG3
Supply V_{CC} from Host System (+5 V)	1–2
Supply V_{CC} from Target System (default)	**2–3**

Note: The Universal Command Converter is factory configured for the JTAG/OnCE buffers to be powered from the target system (JG3 2–3).

2.6.3 Command Convertor Monitor Firmware Upgrades

The monitor code for the Command Converter Revision 6, tailored for the target DSP family in use, is provided with the ADS software, and downloaded automatically into the command converter during ADS system initialization and Universal Command Converter reset. If a revision is issued for the monitor firmware, an environment variable must be defined to specify the filename of the revised monitor. The specified file will be loaded into the Command Converter instead of the standard monitor program. The variable which must be defined is **CC56000**, where '56000' is replaced with name of the DSP family in use, and the defined value is the fully-specified filename of the revised monitor software. For example:

- DOS, in AUTOEXEC.BAT:

  ```
  SET CC56300=C:\ADS\REVISIONS\MONITOR.LOD
  ```

- UNIX, with C shell, in .login or .cshrc:

  ```
  setenv cc56300 /ads/revisions/monitor.lod
  ```

- UNIX, with Bourne shell, in .profile:

  ```
  cc56300=/ads/revisions/monitor.lod
  export cc56300
  ```

specifies that, for the ADS56300 or GUI56300, the standard DSP56300 family Universal Command Converter monitor code is to be replaced by the code in the file ADS\REVISIONS\MONITOR.LOD.

To verify that the monitor file is loading correctly, start the ADS program and enter the following commands:

```
force s
display v
```

The monitor revision should be 5.05. If the error 'Unable to reset Command Converter' is issued, make sure the correct path is specified in the definition of CC56x00.

SECTION 3

USER INTERFACE COMMANDS

3.1   INTRODUCTION

This section describes the ADS user interface commands in detail. There are examples on how to invoke command line arguments associated with each command and there are examples on debugging compiled C source programs written with the Motorola GNU based C Compiler. Also, the user interface program is designed to communicate with multiple devices in a system.

3.2   COMMAND OVERVIEW

There are forty-five commands available in six functional categories: target memory/register modification, file I/O, target DSP address execution control, C source code debug, miscellaneous tasks, and Command Converter tasks. Since the ADS interface program supports multiple Application Development Module connections, a device argument is included in most of the commands. This device argument designates which ADM Board or target DSP is to be addressed. If no device argument is entered, the default device will be addressed. The user interface program will default to device number 0 upon entry, but may be changed to any one of eight possible default devices.

3.2.1   Memory/Register Display/Modification Commands

There are six program/data memory display/change commands available which allow the user to **A**SSEMBLE (**A**SM), **C**HANGE, **CO**PY, **DI**SASSEMBLE, and **D**ISPLAY registers or memory. A **WAT**CH list may be used to display a variable whenever single stepping or program execution is halted.

3.2.2   File I/O Commands

There are five program/data file I/O commands available which allow the user to **I**NPUT or **O**UTPUT data to/from a target, **L**OAD macro-assembler object module programs, **LO**G commands and/or display entries, and **S**AVE program/data memory to files.

3.2.3   Target Program Execution Commands

There are eight program execution commands available which allow the user to set **B**REAK conditions in program memory, instruct a target to **G**O to a program address and begin execution, **F**ORCE a reset or program halt on a target, **ST**EP x number of instructions before displaying register and memory changes, and **T**RACE through a program one instruction at a time. For symbolic debug capabilities, the **N**EXT command operates essentially the same as the **ST**EP command except that if the DSP opcode being executed calls a subroutine or macro, execution continues until return from the subroutine or macro. The **U**NTIL command has the effect of setting a temporary breakpoint at a specified address, executing until a breakpoint is encountered, then clearing the temporary breakpoint. The **FI**NISH command proceeds until an RTS opcode is encountered for the current subroutine.

3.2.4   C Source Code Debug Commands

There are seven C source code debug commands available. The user may use **WH**ERE to display the C function call stack. The user can then use **UP**, **DO**WN and **FR**AME to traverse the call stack. The user may **RED**IRECT data from stdin/stdout/stderr to files when **STR**EAMS are enabled and the user may also **D**ISPLAY the data type of a variable, function or C expression. **Section 3.8** gives examples on how to use these commands using example C programs provided on the ADS software distribution media.

3.2.5   Command Converter Commands

There are eight commands associated with the Command Converter board. These commands allow the user to **CC**HANGE, **CD**ISPLAY, **CL**OAD, **CS**AVE, **CST**EP, **CT**RACE, or execute a OnCE command sequence (**CG**O) in the Command Converter Y memory. This allows users the ability to write and debug their own command sequences for the OnCE debug port. The **CF**ORCE command is used to reset or interrupt the Command Converter.

3.2.6   Miscellaneous Commands

There are eleven miscellaneous task commands available which allow the user to **E**VALUATE expressions in five different radices, select a new default **DE**VICE, **Q**UIT the User interface program and return to the operating system or get **H**ELP for command

line entry. Also a default **P**ATH may be defined for a target, a new default **R**ADIX may be selected, execute SYSTEM commands or **W**AIT a specified number of seconds before proceeding to the next command. The **HO**ST command allows the host computer interface card address to be changed. The **LI**ST command displays a specified source file when symbolic debug is in effect. The **V**IEW command allows selection of the simulator display mode-source, assembly, or register.

3.3   COMMAND SYNTAX

The command descriptions in **Section 3.6** 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 special punctuation is not used for command entry but rather to describe the forms of the command. Capitalized WORDS indicate command keywords. Command keywords may be entered in either upper or lower case. The portion of the command keyword shown in **BOLDFACE** represents the minimum portion of the keyword that the user must type. The portion of the keywords not in boldface may be typed if desired, but is not required by the user interface. The user interface types 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 the following section. 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 solidus '/' is used to separate entries in lists of alternate command parameters. The user may only enter one of the parameters in the list. The solidus is not entered as a part of the command. For example, when entering the "**LO**G" command, **lo**g c filename and **lo**g s filename are valid entries, but not **lo**g 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, **co**py (from)p:0..10 (to) x:5 should be entered as **co**py p:0..10 x:5. 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 be repeated if desired. For example, when entering the "**D**ISPLAY" command, multiple registers may be specified for display on the same command line.

The following symbolic address forms may be used in any ADS command in place of an absolute address:

> `@linenumber` The line number in the current file.
>
> `@filename@linenumber` The line number in the specified filename.
>
> `@symbolname` The symbol in the current section.
>
> `@sectionname@symbolname` The symbol in the named section.

The @ character preceding the symbols may be omitted in most contexts, but is required when the symbol name is the same as an ADS register or peripheral name. The ADS retains each symbol's associated memory space so that it is not necessary to type the memory space, for example "X:" preceding a symbol label which is associated with X memory in the assembly source program.

The disassembler provides symbolic information for the memory addresses being accessed. This includes jump or branch target addresses as well as data memory accesses. All addressing modes are supported. The register indirect modes generate a label based on the current value of the referenced register. The symbolic information is displayed as a label, or label plus offset, in a comment following the disassembled instruction.

Refer to **Section 3.7.2** and **Section 3.7.3** for details on supported C expression syntax.

3.4　COMMAND PARAMETERS

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

- **address** = **P**:location∕**X**:location∕**Y**:location

- **address_block** = address..location∕address#count

- **address_qualifier** = DSP56300∕DSP56600 hardware breakpoint address qualifier

- **action** = **I1**∕**I2**∕**I3**∕**I4**∕**I5**(increment CNTn)∕**H**(halt)∕**N**(note)∕**S**(show registers)∕**T(expression)**Test expression for true condition

- **#bn** = (break number) decimal integer constant in the range 1 to 99

- **count** = positive integer expression in range 1 to $7FFFFFFF

- **dev_num** = specific device number to be addressed for a given command.

- **dev_list** = dvx,dv0..x,dvx,y,z = one or more targets to be addressed for a given command. For example, device group 0 to 4 is expressed as **dv0..4**, whereas the device set of 1, 3, and 5 is expressed as **dv1,3,5**

- **expression** = any arithmetic expression valid for the Assembler; in addition, the register names may be used in the expression

- **filename** = any valid pathname for the operating system in use

- **JTAG/OnCE_type**= JTAG∕OnCE hardware breakpoint type

- **location** = integer expression; it will be mapped into the DSP address range (0 to $FFFF). (0 to $FFFFFF for DSP56300 or $FFFFFFFF for DSP96002)

- **OnCE_type** = OnCE hardware breakpoint type

- **pathname** = any valid pathname for the operating system in use

- **radix** = **%** for binary value, **`** for decimal value, **$** for hexadecimal value

- **reg** = see ADM reference manual for register names pertaining to the particular DSP being evaluated

- **reg_block** = reg..reg

- **reg_group** = **ALL**∕**CORE**∕**IO**∕**STACK**∕**OnCE**

Note:　Check the DSP *User's Manual* for a description of peripheral registers in the particular DSP being evaluated. Use the "**HELP REGS**" command to get a description of the full set of registers in the DSP for which you are designing.

3.5 COMMAND SUMMARY

- Target Memory/Register Modification

    – **A**SM [**dev_num**] [B(byte wide)] [(beginning at) address] assembler_mnemonic

    – **C**HANGE [**dev_list**] [reg[_block]/address[_block] [expression]]...

    – **CO**PY (from)[**dev_num**] address[_block] (to) address

    – **D**ISPLAY [**dev_list**] [**V**(ADM user interface program Version)/**W**(executing targets)

    – **D**ISPLAY [**dev_list**] [**ON**/**OFF**] [reg[_block/_group]/address[_block]]..

    – **DI**SASSEMBLE [**dev_list**] [B(byte wide)][address[_block]]

    – **WAT**CH [**dev_list**] [#wn] [radix] reg/addr/expression/{c_expression}

    – **WAT**CH [**dev_list**] [#wn] **OFF**

- File I/O

    – **I**NPUT [**dev_list**] #(file number)... **OFF**

    – **I**NPUT [**dev_list**] [#(file number)] address **TERM**/filename [**-rd**/**-rf**/**-rh**/**-ru**]

    – **L**OAD [**dev_list**] [**B**(byte wide) address_offset] (from) **filename**

    – **L**OAD [**dev_list**] [**S**(state)/**M**(memory-only)/**D**(debug symbols-only)] (from) filename

    – **LO**G [**dev_list**] [OFF] **V**(source status)/**C**(commands)/**S**(session)] [filename]] [**-O**/**-A**/**-C**]

    – **O**UTPUT [**dev_list**] [#(file number)] **OFF**

    – **O**UTPUT [**dev_list**] [#(file number)] address filename/**TERM** [**-rd**/**-rf**/**-rh**/**-ru**] [**-O**/**-A**/**-C**]

    – **S**AVE [**dev_num**] **S**(state)/address_block... filename [**-O**/**-A**/**-C**]

- Target Execution Control

    – **B**REAK [**dev_list**] [#bn...] [**OFF**/**E**(enable)/**D**(disable)]

    – **B**REAK [**dev_list**] [#bn] EOF [#fn] [t(expression)] [count] [action]

    – **B**REAK [**dev_list**] [#bn] swbp_type address [t(expression)] [count] [action]

- Hardware Breakpoints

  • For DSP56300 and DSP56600 families:

  BREAK [**dev_list**] [#bn [access] [JTAG∕OnCE_type] [addr_qual] address [break_qual [addr_qual] address] [**t**(expression) [count] [action]]

  • Other families:

  BREAK [**dev_list**] [#bn] [access] [type] [address[_range] [count] [action]

  – **FI**NISH [**dev_list**]

  – **F**ORCE [**dev_list**] **R**(reset to Debug mode)∕**B**(break)∕**RU**(reset to User mode∕**S**(system)

  – **G**O [**dev_list**] [(from) location∕**R**(reset)] [#bn] [:(occurrence)count]

  – **N**EXT [**dev_list**] [count] [**LI**(source lines)∕**IN**(instructions)]

  – **ST**EP [**dev_list**] [count] [**LI**(source lines)∕**IN**(instructions)]

  – **T**RACE [**dev_list**] [count] [**LI**(source lines)∕**IN**(instructions)]

  – **U**NTIL [**dev_list**] addr∕line_number∕address_label

- C Source Code Debug

  – **DO**WN [**dev_list**] [count]

  – **FR**AME [**dev_list**] [#frame-number]

  – **RED**IRECT [**dev_list**] **STDIN OFF**∕file

  – **RED**IRECT [**dev_list**] **STDOUT**∕**STDERR OFF**∕file [**-A**∕**-O**∕**-C**]

  – **RED**IRECT [**dev_list**] [**OFF**]

  – **STR**EAMS [**dev_list**] [**E**NABLE∕**D**ISABLE]

  – **TY**PE [**dev_list**] {c_expression}

  – **UP** [**dev_list**] [count]

- Command Converter Control

  – **CC**HANGE [**dev_list**] [**FLAG**∕**XPTR**∕**YPTR**∕address[_block]] [(to) expression]

  – **CD**ISPLAY [**dev_list**] [**FLAG**∕**XPTR**∕**YPTR**∕address[_block]]

  – **CF**ORCE [**dev_list**] **R**(reset)∕**B**(break)∕**D**(Debug mode)∕**U**(User mode)

  – **CG**O [**dev_list**] [address]

- **CL**OAD [**dev_list**] filename
- **CS**AVE [**dev_num**] address_block filename [**-O**∕**-A**∕**-C**]
- **CST**EP [**dev_list**] [count]
- **CT**RACE [**dev_list**] [count]

- GUI Windows

  - **WA**SM [**dev_list**] [**OFF**]
  - **WB**REAKPOINT [**dev_list**] [**OFF**]
  - **WC**ALLS [**dev_list**] [**OFF**]
  - **WCO**MMAND [**OFF**]
  - **WH**ERE [**dev_list**] [[+∕-]n]
  - **WI**NPUT [**dev_list**] [**OFF**]
  - **WL**IST [win_num] **OFF**∕file
  - **WM**EMORY [**dev_list**] [win_num] space [addr]
  - **WM**EMORY [win_num] [**OFF**]
  - **WO**UTPUT [**dev_list**] [**OFF**]
  - **WR**EGISTER [**dev_list**] [win_num] [**OFF**]
  - **WSE**SSION [**OFF**]
  - **WS**OURCE [**dev_list**] [**OFF**]
  - **WST**ACK [**dev_list**] [**OFF**]
  - **WW**ATCH [**dev_list**] [win_num] [#wn] [radix] reg∕addr∕expression
  - **WW**ATCH [**dev_list**] [win_num] [#wn] [**OFF**]

- Miscellaneous

  - **E**VALUATE [**dev_list**] [**B**(bin) **D**(dec)∕**F**(flt)∕**H**(hex)∕**U**(Uns)] expression∕{C expression}
  - **DE**VICE [**dev_list** [device_type∕**ON**∕**OFF**∕**X**]]
  - **DE**VICE [**dev_num**] [cc_num] [tms_num] [chain_pos] [device_type]
  - **DE**VICE cc_num tms_num chain_pos IR count
  - **H**ELP [**dev_num**] [command∕reg] (Alternative syntax: command ?)
  - **HO**ST [IO PC IO addr] [TIMEOUT value]

- **LI**ST [**+**/**-**/**.**/addr]

- **P**ATH [**dev_list**] [pathname]

- **P**ATH [**dev_list**] + pathname[,pathname...]

- **P**ATH [**dev_list**] -

- **Q**UIT [**E**(enable)] [**D**(disable)]

- **R**ADIX [**dev_list**] [**B**(bin)/**D**(dec)/**H**(hex)/**F**(Flt)/**U**(Uns)]
  [reg[_block]/address[_block]]...

- **SY**STEM [-C(continue immediately)] [system_command [argument_list]]
  (non-GUI only)

- **UN**LOCK dev_type password

- **V**IEW [**A**(assembly)/**S**(source)/**R**(register)]

- **W**AIT [[**dev_list**] B(break)]/count(seconds)]

- **Ctrl-X** (toggle ADM service requests on/off)

- **Ctrl-S** (screen scroll freeze)

- **Ctrl-C** (abort display command)

- **Ctrl-F** (insert next command in command circular buffer on command line)

- **Ctrl-B** (insert last command in command circular buffer on command line)

- **up-arrow**/**down-arrow** (scroll display screen)

3.6   DETAILED COMMAND DESCRIPTIONS

The following subsections provide a detailed description for each of the user interface Commands.

3.6.1   ASM—Single Line Interactive Assembler

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

The **A**SM command invokes a single-line interactive DSP Assembler program allowing the user to create or edit DSP object code programs in memory using assembly language mnemonics. Each source line is immediately converted into the proper machine language code and stored in ADM or target system memory. The source line entry is not saved. The address parameter is optional. The beginning address may be in any of the three (p, x, or y) memory maps of the DSP.

Note: The Y memory is only valid for DSP56000 and DSP96002 family members. If no address is specified, assembly begins in the p (program) memory space using the current program counter value as the beginning address.

Invoking this command causes existing 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 or edit the existing object code. 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.

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.

If the interactive Assembler is invoked with the GUI version of the ADS, 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.
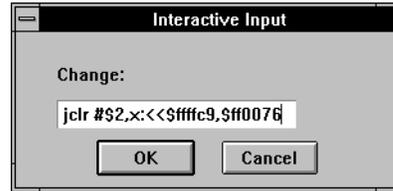
Figure 3-**1** Interactive Assembler Dialog Box

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.

Example 3-**1**   **A**SM Command Examples

---

aₛₘ **p:$50**

Start interactive Assembler at program memory address 50 hex of the current default device.

aₛₘ

Start interactive Assembler at current program counter value of the current default device.

aₛₘ **myfile.asm@7**

Start interactive Assembler at the address corresponding to myfile.asm line 7.

aₛₘ **dv3 p:10**

Start interactive Assembler at target address #3 program memory address 10.

aₛₘ **nop**

Overwrite the instruction at the current pc with the specified instruction.

aₛₘ **x:0 add #<2,a**

Store assembled instruction in specified data memory location. This feature may be useful for patching overlaid programs where overlays are copied from data to program memory before execution.

aₛₘ **dv3 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.

3.6.2 BREAK—Set, Modify, or Clear Breakpoint

BREAK [**dev_list**] [#bn] [**OFF**/**E**(enable)/**D**(disable)]

Software Breakpoints

BREAK [**dev_list**] [#bn] swbp_type address [t(expression)] [count] [action]

DSP56300/DSP56600 Families Hardware Breakpoints

BREAK [**dev_list**] [#bn] [access] [JTAG/OnCE_type] [addr_qual] address
[break_qual [addr_qual] address] [**t(expression)** [count] [action]

Other Families Hardware Breakpoints

BREAK [**dev_list**] [#bn] access] [OnCE_type] [address[_block]] [count] [action]

The **B**REAK command enables or disables breakpoints which causes a user's DSP program execution to halt and enter the Debug mode of operation. There are two basic types of breakpoints that may be entered, hardware breakpoints and software breakpoints.

Breakpoint list rules:

1. The default breakpoint will always be hardware of program core fetch if no break type is entered and a single address is used, example "break p:50". For the DSP56300 and DSP56600 families, the access type defaults will always be read for program memory and read/write for data memory.

2. If more than one hardware breakpoint is entered in the breakpoint list, only the last hardware breakpoint entered will be enabled, all other hardware breakpoints will be disabled.

3. Break counts may be used for hardware and software breakpoints. If no count is entered it defaults to a value of 1. Software breakpoint counts may also be set with the **G**O command.

4. Software breakpoints should only be set on opcode addresses and not operand addresses. In order to help prevent setting a breakpoint on an operand address, the user is issued a warning whenever a software breakpoint is set on an illegal opcode. However, if the operand happens to be a legal opcode, no warning will be issued and the user program may not execute properly because a DEBUGCC opcode will reside in the operand location rather than the correct operand.

5. When a breakpoint is defined, it will be enabled by default. The user must specifically disable or turn off the breakpoint number if that breakpoint is not desired.

The OnCE circuitry as described in the User's Manual allows the user to set hardware breakpoints to occur on Program memory addresses or Data memory addresses. A real-time 24 bit breakpoint counter allows the user to stop program execution and to enter the Debug mode of operation after the nth occurrence of entering the breakpoint address. The user has the option of selecting whether the Program memory access is either a read, a write or read/write generated from the Program controller. The hardware breakpoint accesses are as follows:

Table 3-1   Hardware Breakpoint Access

Break Access	Meaning
r	Break on Program or Data memory read
w	Break on Program or Data memory write
rw	Break on Program or Data memory access

Table 3-2 and **Table 3-2** on page 3-15 describe the different hardware breakpoint types for each DSP:

Table 3-2   OnCE Hardware Breakpoint Types

DSP	Break Type	Meaning
All	pcf	Break on any Program core fetch (read only)
All	pcm	Break on Program read (fetch or move - read only)
24, 32 bit	pcfm	Break on Program access (fetch or P move - r/w)
24 bit	pce	Break on executed fetch only (read only)
32 bit	pdma	Break on Program memory DMA accesses
24 bit	pa	Break on Program access (r/w)
16 bit	xab1	Break on X address bus 1 access
16 bit	xab2	Break on X address bus 2 access
24, 32 bit	xa	Break on X data memory access (r/w)
24, 32 bit	ya	Break on Y data memory access (r/w)
32 bit	xdma	Break on X memory DMA access
32 bit	ydma	Break on Y memory DMA access

Table 3-3 JTAG/OnCE Hardware Breakpoint Types

DSP	Break Type	Meaning
16 bit	pce	Break on executed fetch only (read only)
16 bit	pcm	Break on Program fetch or move (r, w, or rw)
16 bit	xab1	Break on X address bus 1 access (r, w, or rw)
24 bit	pa	Break on Program access (r, w, or rw)
24 bit	xa	Break on X data memory access (r, w, or rw)
24 bit	ya	Break on Y data memory access (r, w, or rw)
24 bit	dma	Break on DMA access (r, w, or rw)

Note: Program core fetches are always read only.

Note: DSP56600 hardware breakpoint logic is the same as the DSP56300.

The number of simultaneous hardware breakpoints are limited by on-chip logic. When more than one hardware breakpoint is set the last or highest breakpoint number with a hardware breakpoint set will be the one which is set in the OnCE port. The OnCE circuitry also allows the user to set conditional or unconditional software breakpoints to occur by using the special DEBUGcc instructions. This allows the user to set as many breakpoints as required to debug algorithms. User program execution is halted after the opcode is executed when a breakpoint occurs. A breakpoint occurrence counter may be set so that from one to $ffff occurrences of the breakpoint address(es) in real-time occur before returning to the Debug mode of operation.

The DSP56300 and DSP56600 OnCE circuitry allows the two breakpoint addresses to form a range of inclusive or exclusive addresses. The address qualifiers in the breakpoint statement express whether the breakpoint address placed in the breakpoint comparator is to be less than, greater than, equal to or not equal to the address bus being monitored. The address qualifiers correspond to the breakpoint control tables in the OnCE section in the user manual.

Table 3-4 DSP56300 AND DSP56600 Hardware Breakpoint Address Qualifiers

Break Address Qualifier	Meaning
>	Address bus is greater than breakpoint address
<	Address bus is less than breakpoint address
==	Address bus is equal to breakpoint address
!=	Address bus is not equal to breakpoint address

When enabling both of the DSP56300 or DSP56600 hardware breakpoint registers they form a sequential state machine. This means that the breakpoint counter will not be decremented until the event qualifier is met for both breakpoints. The breakpoints can be coupled so that the event qualifier is triggered when breakpoint 0 and 1 are true, breakpoint 0 or 1 is true, or breakpoint 0 becomes true then breakpoint 1 becomes true.

Table 3-5   DSP56300 and DSP56600 Hardware Breakpoint Event Qualifier

Break Event Qualifier	Meaning
and	Breakpoint 0 and Breakpoint 1
or	Breakpoint 0 or Breakpoint 1
then	Breakpoint1 after Breakpoint 0

Breakpoint 0 implies the first of two possible breakpoints in a breakpoint expression.

There are seventeen types of DEBUGcc software breakpoints based on the Condition Code Register (CCR) bit values. **Table 3-6** describes the command line options for the DEBUGcc breakpoint types.

Table 3-6   Software Breakpoint Types

DSP	Type	Meaning	Condition Code Bit(s)
all	cc or hs	carry clear	C = 0
all	cs	carry set	C = 1
16, 24 bit	ec	extension clear	E = 0
all	eq	equal	Z = 1
16, 24 bit	es	extension set	E = 1
all	ge	greater or equal	N (xor) V = 0
all	gt	greater than	Z or (N (xor) V) = 0
32 bit	hi	higher than	Z or C = 0
16, 24 bit	lc	limit clear	L = 0
all	le	less or equal	Z or (N (xor) V) = 1
16, 24 bit	ls	limit set	L = 1
all	lt	less than	N (xor) V = 1
all	mi	minus	N = 1
all	ne	not equal	Z = 0
16, 24 bit	nr	normalized	Z or (Not N (and) Not E) = 1

Table 3-6 Software Breakpoint Types (Continued)

DSP	Type	Meaning	Condition Code Bit(s)
all	pl	plus	N = 0
16, 24 bit	nn	not normalized	Z or (Not U (and) Not E) = 0
32 bit	vc	overflow clear	V = 0
32 bit	vs	overflow set	V = 1
all	al	always	N.A.

Note: When using software breakpoints the opcode at that address will be replaced by one of the software breakpoints chosen. Therefore it is wise to set conditional software breakpoints at an address with a NOP opcode.

For the 32-bit DSP96002 there are 21 types of FDEBUGcc software breakpoints based on the Condition Code Register (CCR) and/or the Exception Register (ER) bit values. **Table 3-7** describes the command line options for the FDEBUGcc breakpoint types

Table 3-7 Floating Point Software Breakpoint Types

Type	Meaning	Condition Code Bit(s)
eq	equal	Z = 1
err	error	UNCC or SNAN or OPERR or OVF or UNF or DZ = 1
ge	greater or equal	NAN or (N and ~Z)= 0
gl	greater or less than	NAN or Z = 0
gle	greater, less or equal	NAN = 0
gt	greater than	NAN or Z or N = 0
inf	infinity	I = 1
le	less or equal	NAN or ~(N or Z) = 0
lt	less than	NAN or Z or ~N = 0
mi	minus	N = 1
ne	not equal	Z = 0
nge	not (greater or equal)	NAN or (N and ~Z) = 1
ngl	not (greater or less)	NAN or Z = 1
ngle	not (greater, less or equal)	NAN = 1

Table 3-7   Floating Point Software Breakpoint Types

Type	Meaning	Condition Code Bit(s)
ngt	not greater than	NAN or Z or ~N = 1
ninf	not infinity	I = 0
nle	not (less than or equal)	NAN or ~(N or Z) = 1
nlt	not less than	NAN or Z or ~N = 1
or	ordered	NAN = 0
pl	plus	N = 0
un	unordered	NAN = 1

The host computer program will evaluate the breakpoint expression when a breakpoint occurs and if there is not a test condition, the user will be informed of the target having stopped. If there is a test condition, it must be true before the target is halted. If the test condition is not true, the user interface program will replace the original opcode at the breakpoint address, single-step through it, replace it with the conditional breakpoint opcode, and pass control of the processor back to the User mode from the Debug mode.

If a breakpoint is met during DSP program execution, there are various actions which may be performed. If no action argument is entered, the default action is to halt program execution and display all enabled registers and memory blocks. More than one action argument may be entered at once. The valid action arguments available are listed in **Table 3-8**.

Table 3-8   Breakpoint Actions

Argument	Action.
H	Halt execution—this is the default.
In	Increment counter variable CNTn (n = 1 / 2 / 3).
N	Note—display the breakpoint expression and continue.
S	Show the enabled register/memory set and continue.
T(expression)	Test the expression within the parenthesis. If the expression is true execute the actions following the expression, otherwise continue program execution.

Note: The Ctrl-X key acts as a toggle to disable or enable ADM service requests. Disabling ADM service requests freezes execution of multiple "show" or "note" breakpoints and allows user interface commands to be entered and the registers to be examined. Enabling ADM service requests resumes ADM execution.

A breakpoint expression may be any logical expression that is valid for the DSP Macro Assembler. **Table 3-9** is a list of operators that may be used in the breakpoint expression:

Table 3-9   Expression Operators

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

If more than one breakpoint expression is entered for a breakpoint address, the actions following each expression are executed only if that expression is evaluated as true. If no specific data representation is used in the break address or break expression, the data values will be evaluated using the default radix when an ADM requests the host for breakpoint service. Therefore, the specific data representation should be used when setting breakpoints. It should also be noted that there is a major difference between setting conditional software breakpoints in user memory and using breakpoint expressions to evaluate a breakpoint. The conditional software breakpoint is done in real-time inside the DSP, whereas the breakpoint expression requires stopping the DSP and evaluating the expression from the host computer side. This means that the host computer must interrogate the target DSP to determine if the breakpoint expression is

true. This is not done in real-time. The same holds true for the CNT1-3 counters versus the breakpoint counter register in the OnCE port of the target system. The CNT1-3 counters are software counters in the host computer whereas the breakpoint counter register is hardware and is decremented in real-time.

Example 3-**2**   General Breakpoint Examples for DSPs with OnCE or JTAG/OnCE Ports

break

Display all currently enabled breakpoints for all target DSPs.

break **dv2**

Display currently enabled breakpoints for target DSP address #2.

break **off**

Disable all currently enabled breakpoints for the default target DSP address.

break **dv2 off**

Disable currently enabled breakpoints for target DSP address 2.

break **off 2**

Disable breakpoint number 2 of the current default target address.

break **dv2 p:$100**

Halt DSP program execution of target DSP address 2 and display enabled registers and memory when the DSP instruction at program address 100 hex is reached. This is a hardware breakpoint which will work with OnCE and JTAG/OnCE based DSPs.

break **p:$30 s**

Display enabled registers and memory of the current default target DSP address and continue program execution when the DSP instruction at program address 30 hex is reached. This is a hardware breakpoint which will work with OnCE and JTAG/OnCE based DSPs.

break **dv3 p:$200 t(r0>r1) h**

If the value of R0 is greater than R1 in target DSP address 3 when its DSP instruction at its program counter 200 hex is reached, halt target DSP address 3. To evaluate whether R0 is greater than R1, the host computer will set a hardware breakpoint at address 200 and will interrogate the target DSP every time a breakpoint occurs at that address. This is a hardware breakpoint which will work with OnCE and JTAG/OnCE based DSPs.

break **al @32 t({i>10}) h**

Break if the program reaches line 32 and the C variable "i" is greater than 10.

break **le p:$320 h**

Halt DSP program execution of default target DSP address and enter Debug mode when the Z or (N and V) bits of the CCR are equal to 1 at the address $320 of program memory. This is a software breakpoint, and testing of the Condition Code Register is done real-time.

Example 3-**2**   General Breakpoint Examples for DSPs with OnCE or JTAG/OnCE Ports
(Continued)

break **al p:$320**

Halt DSP program execution of default target DSP address and enter Debug mode
unconditionally at the address $320 of program memory. This is a software breakpoint and
must be placed in SRAM.

break **r xa x:300**

Halt DSP program execution of the default target DSP address and enter Debug mode when a
read access of X data memory address 300 occurs. This is a hardware breakpoint which will
work with OnCE and JTAG/OnCE based DSPs.

Example 3-**3**   General Breakpoint Examples for DSPs with OnCE Ports

break **rw pcfm p:$100 $20**

Halt DSP program execution of the default target DSP address and enter the Debug mode when
the 32nd occurrence of a read or write access of a program core fetch or move occurs at address
$100.

break **rw pce p:$250**

Halt DSP program execution of default target DSP address and enter Debug mode when a read
or write access of program memory address 250 hex occurs.

Example 3-**4**   General Breakpoint Examples for DSPs with JTAG/OnCE Ports

break **r xa > x:104 and < x:110**

Halt DSP program execution on default target DSP when a read access of X memory address
range 105 to 109 occurs 1 time.

break **rw pa == p:104 or == p:110**

Halt DSP program execution on default target DSP when a read or write access of program
memory address 104 or 110 occurs 1 time.

3.6.3 CCHANGE—Change Command Converter Memory

CCHANGE [**dev_list**] [**FLAG**/**XPTR**/**YPTR**/address[_block]] [expression]

The **CC**HANGE command allows the memory examination or modification of the OnCE Command Converter P, X or Y data memory spaces of the DSP56002. This command is useful for users who wish to design and debug their own OnCE command sequences. The command sequence description with respect to the Command Converter monitor program is outlined in 5.1.

The XPTR is Command Converter x memory location 4 and is used to point to the x memory area where values read from the target OnCE are to be stored. The YPTR is Command Converter x memory location 2 and is used to point to the y memory area where sequences are to start from when issuing a **CG**O command.

Note: Command Converter X memory addresses 0 to 7F hex are reserved for use by the Command Converter monitor. These locations should not be changed by the user. For more details on the usage of these locations refer to the monitor program source listing. P memory locations 0 to 1B0 hex are reserved for the monitor which is boot loaded from the Command Converter EPROM.

Example 3-5   **CC**HANGE Command Examples

cchange **dv2 x:0**

Display the current value of X:0 of Command Converter #2 memory and prompt the user for a new value. Subsequent values may be displayed or changed by entering a carriage return. To exit this interactive mode, use the escape key.

cchange **y:0..$10 $0**

Change y:0 to y:$10 of the default Command Converter to a value of 0.

3.6.4 CDISPLAY—Display Command Converter Flags and Memory

CDISPLAY [**dev_list**] **FLAG**/**XPTR**/**YPTR**/address[_block]...

The **CD**ISPLAY command allows the user to examine the Command Converter flag register, Y memory pointer, or the P, X or Y memory values used to transfer OnCE serial command sequences to the target DSP. All values will be displayed in hexadecimal.

Command converter X memory locations 0 to 10 hex are used for temporary storage of flags and constants as defined in 5.1. To display the Command Converter X or Y memory pointers, command line arguments have been added to the **CD**ISPLAY and **CC**HANGE commands.

Example 3-6   CDISPLAY Command Examples

cdisplay **flag**

Display the Command Converter flag register. The flag register is used to store status bits of whether the target DSP is in the Debug mode or User mode, as well as other Command Converter monitor flags.

cdisplay **y:0..$10**

Display the Command Converter Y memory space which is used for the OnCE command sequence transfers to the target DSP.

cdisplay **xptr**

Display the X memory pointer, which is used to save values read from the OnCE port when executing OnCE serial sequences.

3.6.5 CFORCE—Assert Reset or Break on Command Converter

CFORCE [**dev_list**] **R**(reset)/**B**(break)/**D**(Debug mode)/**U**(User mode)

The **CF**ORCE command is used for forcing a hardware reset or hardware interrupt on a Command Converter. The **D** option can be used to force the Command Converter into the Debug mode in the event that the target has entered the Debug mode by some means other than through the ADS program (such as a DEBUG instruction in the user code). The U option can be used to force the Command Converter into the User mode in the event that the target has entered the User mode by some means other than through the ADS program (such as a push button reset or power-on reset). When using the **U** or **D** arguments, internal flags of the user interface program are also set or cleared.

CAUTION

Placing the Command Converter in Debug mode when the target is NOT in Debug mode can cause improper behavior of the ADS system.

Example 3-7   **CF**ORCE Command Examples

cforce **dv1 r**
Force a hardware reset on Command Converter #1.

cforce **b**
Force an interrupt on the default Command Converter.

cforce **d**
Force the default Command Converter into the Debug mode.

cforce **u**
Force the default Command Converter into the User mode.

3.6.6 CGO—Execute OnCE Sequence

CGO [**dev_list**] [(from) address]

The **CG**O command allows the user to execute OnCE command sequences in the DSP56002 controller's Y memory. This command is useful for debugging user defined OnCE serial command sequences which will be used in a target system. A sequence memory pointer resides in the DSP56002 controller's internal X memory at address 2. This pointer is used as the start location and may be changed using the **CC**HANGE command.

Example 3-8 CGO Command Examples

cgo

Execute the OnCE sequence of the default Command Converter starting at the current address in the Command Converter PTR.

cgo **$10**

Change the Command Converter PTR to hex 10 and execute the OnCE sequence of the default Command Converter starting at that address.

3.6.7   CLOAD—Load OnCE Command Sequence

CLOAD [**dev_list**] **filename**

The **CL**OAD command is used for loading a user defined OnCE serial command sequence into the Command Converter internal Y memory. The file must be in DSP object module format (OMF) and have a .lod suffix name. Refer to **Appendix A** for further details on OMF files. This command allows the user to write a OnCE command sequence using the Command Converter monitor program OnCE sequence format as described in 5.1.

Example 3-9   **CL**OAD Example

cload **onceseq.lod**
Load the file "**onceseq.lod**" into the Y memory of the default Command Converter.

3.6.8   CSAVE—Save Command Converter Memory to a File

CSAVE [**dev_num**] address_block filename [**-o**/**-a**/**-c**]

The **CS**AVE command allows the user to save the Command Converter X or Y data memory to a disk file. This is useful when debugging user defined OnCE command sequences using the Command Converter monitor program sequence format.

If a file currently exists with the filename specified the user will be prompted for an action of either appending the data to the file, overwriting the file, or aborting the command.

The selection of the file action may be included in the command line using the **-o** (overwrite), **-a** (append), or the **-c** (cancel) argument. This is useful when executing macro command files.

Example 3-10   **CS**AVE Command Examples

csave **dv0..3 y:0..$20 onceseq.lod**

Save the contents of Command Converters 0, 1, 2, and 3 Y memory addresses 0 to hex 20 to a file named "**onceseq.lod**".

csave **x:$10#10 newdata.lod**

Save the contents of the default Command Converter's X memory addresses hex 10 through hex 1A to a file named "**newdata.lod**".

3.6.9 CSTEP—Step through OnCE Sequence

CSTEP [**dev_list**] [count]

The **CST**EP command allows the user to execute a group of OnCE serial sequences before displaying the OnCE register contents. This gives the user the opportunity to write and debug a OnCE command sequence using the Command Converter monitor program sequence format as described in **Section 5 Functional Description**.

Note: The OPDBR and OPILR registers always display the last values stored after executing a **G**O, **ST**EP or **T**RACE command or after servicing a breakpoint. The values of these registers will not reflect the changes made to them when executing the **CG**O, **CST**EP, or **CTR**ACE when doing a display of the OnCE registers.

Also, it is important to remember that writing to the OPDBR register is in effect manipulating the DSP program controller. Whenever 2-word opcodes are being written to the OPDBR, it is best to **CTR**ACE or **CST**EP 2 before displaying registers.

Example 3-11   CSTEP Command Examples

cstep

Execute one OnCE serial command of the default Command Converter's Y memory pointed at by its YPTR. The OnCE registers will be displayed after the command is executed.

cstep **$10**

Execute hex ten OnCE serial commands of the default Command Converter's Y memory pointed at by its YPTR. The OnCE registers will be displayed after the hex 10 commands have all been executed.

A macro command file can help in single stepping through user defined OnCE sequences and displaying results. The display of registers after a **CST**EP or **CT**RACE were not implemented because of the nature of having to access the OPDBR register to retrieve the register values. An example of a macro file would be the following:

```
cstep 2                    ;execute 2 OnCE commands then show XPTR and YPTR
cdisplay x:80..90 y:80..9f ;display the Command Converter x and y memory
display                    ;display the target registers
```

3.6.10 CTRACE—Trace through OnCE Sequence

CTRACE [**dev_list**] [count]

The **CT**RACE command allows the user to single step through a OnCE command sequence in the Command Converter Y memory pointed at by the Command Converter YPTR. This enables the user to write and debug OnCE command sequences using the Command Converter monitor program sequence format.

Note: The OPDBR and OPILR registers always display the last values stored after executing a **G**O, **ST**EP or **T**RACE command or after servicing a breakpoint. The values of these registers will not reflect the changes made to them when executing the **CG**O, **CST**EP, or **CT**RACE when doing a display of the OnCE registers.

Also, it is important to remember that writing to the OPDBR register is in effect manipulating the DSP program controller. Whenever 2-word opcodes are being written to the OPDBR, it is best to **CT**RACE or **CST**EP 2 before displaying registers.

Example 3-12   **CT**RACE Command Example

```
ctrace 10
```

Execute 10 OnCE serial commands of the default Command Converter and display the OnCE register contents after each command is executed. The serial commands reside in the Command Converter Y memory and are pointed at by the Command Converter YPTR register.

A macro command file can help in single stepping through user defined OnCE sequences and displaying results. The display of registers after a **CST**EP or **CT**RACE were not implemented because of the nature of having to access the OPDBR register to retrieve the register values. An example of a macro file would be the following:

```
ctrace 2                         ; single step 2 OnCE commands and show XPTR and
                                 ; YPTR after each trace.
cdisplay x:80..90 y:80..9f       ;display the Command Converter x and y memory
display                          ;display the target registers
```

3.6.11   CHANGE—Change Register or Memory Value

CHANGE [**dev_list**] [reg[_block]/address[_block] [expression]]...

The **C**HANGE command allows register or memory examination or modification. Memory blocks may be initialized to a particular value by including an end address. If the command is entered without a value, the register or memory location of the current default target DSP address will be displayed with its current value on the command line and the user will be prompted for a new value. Multiple changes may be specified in a single command line. Each specified destination(block) must be followed by the value of 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. To change the register or memory location contents and display the next register or location, type the new value followed by carriage return. Subsequent or previous memory locations or register names can be examined and changed if required by typing, respectively, **Up-Arrow** (Ctrl-U) or **Down-Arrow** (Ctrl-N). Typing a new value followed by Up- or Down-arrow does not change the open location. Pressing the **Esc** key causes the interactive **C**HANGE command to terminate.

<div style="border:1px solid black; padding:1em;">

CAUTION

Users should be aware that some peripheral registers contain handshake bits that change state when they are read. Reading these registers can interfere with the proper operation of the peripheral when returning to the user's program.

</div>

If interactive change mode is entered with the GUI version of the ADS, 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.
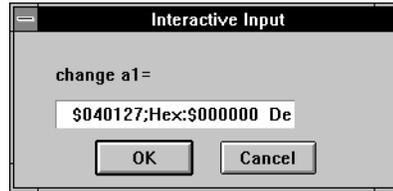
Figure 3-2 Interactive Change Dialog Box

The Session and Command windows are written during interactive change operations. Both windows display the original **C**HANGE command, the Session window display each change as it is applied.

Example 3-13 **C**HANGE Command Examples

change

Display the current default target DSP address's register values individually starting with register a and prompt the user for new values.

change **x:$55**

Display x memory location hexadecimal 55 of the current default target DSP address and prompt the user for a new value. Subsequent or previous memory locations may be examined and changed using the up arrow key for the previous address and down arrow for the next address.

change **dv3 pc**

Display the current value of the program counter on target DSP address 3 and prompt the user for a new value. Subsequent or previous register values may be examined and changed using the up arrow key for the previous address and down arrow for the next address.

change **p:$20 $123456**

Change the current default target DSP address's p memory address hexadecimal 20 to hexadecimal 123456.

change **r0..r7 0 p:$30..$300 0 x:$fffe $55 pc '100**

Change the current default target DSP address's registers r0 to r7 to 0, p memory addresses 30 hex to 300 hex to 0, x memory address fffe hex to 55 hex and the program counter to 100 decimal.

change **dv1,3,5 r0..r7 0 p:$1000..$2000 0**

Change target DSP addresses 1,3 and 5 registers r0 to r7 to 0 and their program memory addresses 1000 hex to 2000 hex to a value of 0.

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.

3.6.12 COPY—Copy a Memory Block

COPY (from)[**dev_num**] address[_block] (to) address

The **CO**PY 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. This command allows the copying of program or data blocks within a single target DSP device. To transfer information from one target device to another, use the **S**AVE command to create an object file, which may then be loaded into the destination device.

Example 3-14 **CO**PY Command Examples

copy **p:0..30 p:100**

Copy the data in current default target DSP address program memory starting at 0 and ending at 30 to program memory starting at 100.

copy **dv3 p:0..30 dv2 p:100**

Copy the data in target DSP address number 3 program memory starting at 0 and ending at 30 to target DSP address number 2 program memory starting at 100.

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 **x:$100..$200 x:$150**

Copy the data in the current default target DSP address X memory starting from hex $200 down to hex $100 and put the data in the current default target DSP address X memory starting at hex $250 down to hex $150. Whenever the addresses of the source and destination overlap, the source end address will be used and the addresses will be decremented rather than incremented.

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**.

3.6.13   DEVICE—Select Default target DSP address

DEVICE [**dev_list** [**device_type**/**ON**/**OFF**/**X**]]

DEVICE [**dev_num**] [**chain_num**] [**tms_num**] [**chain_pos**] [**device_type**]

DEVICE **cc_num tms_num chain_pos IR count**

The **DE**VICE command allows the user to:

- Select the current device for command input and session output

- Activate one or more of the target devices controlled by the ADS.

- Specify the device type of each target device

- List the type and status of each device

- Specify the position of devices in a JTAG chain

- Specify non-Motorola devices in a JTAG chain

- Enable and disable each device

- Deactivate a device and deallocate all associated structures

The command line prompt displays the number of the currently selected device. At start-up, device DV0 is activated and selected as the current device.

- device_type specifies which type of DSP is being emulated. If omitted, a default value will be selected, depending on the device family in use. Use **DE**VICE command for a list of supported device types.

- ON makes the specified device(s) active for program execution

- OFF suspends program execution for the specified device. The state of the device is not otherwise changed.

- X deactivates the device and discards all associated structures. If the X parameter is used for the current device, another device will become the current device. At least one device must be activated at all times; the last device may not be deactivated.

- JTAG parameters—The ADS supports up to eight command converters on a development host. Each Command Converter supports one JTAG chain which may service up to twenty-four devices. The **DE**VICE command associates each device in any position in the JTAG chain with an ADS device number (dvn). The ADS only performs debugging operations on Motorola DSP devices. However, to support target systems incorporating other devices, the **DE**VICE command also permits the specification of the JTAG instruction register length so such devices may be handled correctly.

- **DVn**—Specifies the device number to be used to access the device described by the remainder of the parameters. (0..31)

- **CCn**—Specifies the Command Converter to which the device is connected (0..7)

- **TMSn**—Specifies which TMS (Test Mode Select) line controls this device (0..1); Command Converter revision 6 only supports TMS0.

- **POSn**—Specifies which position the device occupies in the JTAG chain. The device connected directly to TDO from the Command Converter is position 0. $(0 \le n)$

- **IR n**—Specifies the length of the instruction register for unsupported devices. $(2 \le n)$

Note: The defaults for the above parameters are as follows: CCn defaults to the device number DVn and all other parameters default to zero. This maintains compatibility with previous versions.

Example 3-15   DEVICE Command Examples

---

device

Display all activated target DSP addresses and device types, and lists all supported family members.

device dv0..2 on

Activate target DSP address 0, target DSP address 1, and target DSP address 2.

device dv0,3 off

Deactivate target DSP address 0 and target DSP address 3.

device dv2

Select target DSP address 2 as the default target DSP for command entry.

device dv1 x

Deactivate target DSP address 1 and discard all associated data structures. If this was the selected device, select another.

device dv12 cc3 pos2 56301

Specifies that device DV12 refers to a DSP56301, which is controlled by Command Converter #3, occupying the third (0,1,2...) position in the chain. TMS0 is used by default.

device cc3 tms0 pos1 ir 3

The device on Command Converter 3, TMS chain 0, position 1 is not to be used in this development session. It has an instruction register 3 bits wide. Note that no device number may be specified, and that all fields are required.

Note: The instruction register length must not be specified for a device which has been allocated an ADS device number.

3.6.14   DISASSEMBLE—Single Line Disassembler

DISASSEMBLE [**dev_list**] [**B**(byte wide)][address[_block]]

The **DI**SASSEMBLE command allows the user to review DSP object code in its assembly language mnemonic format. All invalid opcodes will display "DC" for define constant. The b (byte-wide) parameter constructs the instruction words by taking one byte from each word of memory, starting from the specified address.

Example 3-**16   DI**SASSEMBLE Command Examples

disassemble

Disassemble a page of instructions pointed at by the user interface program disassembler counter of the current default target DSP address. A counter maintains the last instruction disassembled so subsequent instructions may be disassembled by merely entering a carriage return to execute the same instruction again.

disassemble **p:0..20**

Disassemble program memory address block 0 to 20 of the current default target DSP address.

disassemble **dv2 x:$50#10**

Disassemble ten instructions of the target DSP address 2 starting at x memory map 50 hex.

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 **b y:$1000#$40**

Disassemble forty 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.

3.6.15 DISPLAY—Display Register or Memory

DISPLAY [**W**(executing targets)/[**dev_list**] **V**(ADS user interface program version)]

DISPLAY [**dev_list**] [**ON**/**OFF**] [reg[_block/_group]/address[-_block]]...

The **D**ISPLAY command allows the user to examine the contents of a register group and/or memory block in the radix specified by the **R**ADIX command. The default display radix is hexadecimal. It may also be used to enable or disable particular registers or memory locations for automatic display when executing debug commands.

Entering the command with no parameters will cause the display of all enabled registers and memory blocks. Registers and memory blocks may be enabled or disabled by entering the command with one of the "enable" keywords (i.e., **ON** or **OFF**) prior to the register and/or memory list. The keywords have the following meaning:

- **ON** = Enable display of the following registers and memory locations.
- **OFF** = Disable display of the following registers and memory locations.

Entering the **D**ISPLAY command with only a register or memory list causes immediate display of the listed registers and memory locations without affecting their "enable" status. Several register group names have been predefined and may be used in the display list to enable, disable or display all of the registers in the group. The list of group names available depends on the target device. For a list of the peripheral names available with a particular ADS system, use the command **H**ELP periph.

CAUTION

Some peripheral registers contain handshake bits that change state when they are read. Reading these registers can interfere with the proper operation of the peripheral within the user program.

.

Example 3-17 DISPLAY Command Examples

display

Display all currently enabled registers and memory of the current default target DSP address.

display **on**

Enable all programming model registers for display on the current default target DSP address.

display **p:0..300**

Display p memory addresses 0 through 300 of the current default target DSP address.

display **on p:0..20 x:30..40 x:$100**

Display enable p memory address block 0 to 20, x memory address block 30 to 40 and X memory address hexadecimal 100 of the current default target DSP address.

display **dv2 p:30..50**

Display p memory address block 30 to 50 of target DSP address 2.

display **w**

Display all target DSP addresses that are currently executing user programs.

display **v**

Display the user interface program and Command Converter monitor program revision numbers.

display **on host**

Display enable the host peripheral registers.

display **on all**

Display enable all programming model and peripheral registers.

display **X:0..$100**

Display the values of x memory locations 0 to 100 hex.

3.6.16 DOWN—Move Down the C Function Call Stack

DOWN [**dev_list**] [n]

The **DO**WN command is used to move down the call stack. It can be used in conjunction with the **WH**ERE, **FR**AME, and **UP** commands to display and traverse the C function call stack. After entering a new call stack frame using **DO**WN, that call stack frame becomes the current scope for evaluation. In other words, for C expressions, the **E**VALUATE command acts as though this new frame is the proper place to start looking for variables.

Example 3-18   DOWN Command Examples

down
Move down the call stack by one stack frame.

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

3.6.17 EVALUATE—Evaluate an Expression

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

The **E**VALUATE 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 **E**VALUATE command line, the current default radix (specified by the **R**ADIX command) will be used. An expression consists of an arithmetic combination of operators and operands. An operand may be a register name, a memory location, or a constant value. For example, A2 evaluates to the contents of register A2. To evaluate A2 as hexadecimal, a dollar sign must precede the value. The same holds for the values A0, A1, A2, B0, B1, and B2.

The order of evaluation of an expression's operators will be associated from left to right. Parenthesis may be used to force the order of evaluation of the expression. A more extensive discussion of the expressions which are valid for the **E**VALUATE command is in the *DSP Assembler Reference Manual* (**Section 3**). The valid symbols for an expression are listed in the description of the **B**REAK command.

Example 3-19   EVALUATE Command Examples

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

evaluate **dv4 r0+p:$1000**
Add the value in r0 register of target DSP address 4 to the value in its program memory address hexadecimal 1000 and display the result using the default radix.

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

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

evaluate **$a0+$b0**
Calculate the sum of hexadecimal a0 plus hexadecimal b0.

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

evaluate **{count+max}**
Evaluate the sum of the C variables "count" and "max", and display the result.

evaluate **{lookup(i)}**
Call the C function "lookup" from the command line, with the argument "i". Display the result of calling the function.

3.6.18   FINISH—Step Until End of Current Subroutine

FINISH [**dev_list**]

The **FI**NISH command executes instructions until a Return-To-System (RTS) instruction is executed within the current subroutine. The ADS interface program simply steps, checking if any instruction is an 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.

Example 3-20   FINISH Command Example

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

3.6.19 FORCE—Assert RESET or BREAK on Target

FORCE [**dev_list**] **R**(reset to Debug mode)/**B**(break)/**RU**(reset to User mode)/**S**(System)

The **F**ORCE command asserts a hardware reset or asserts a debug request on one or more target systems. This command is useful for reinitializing all registers, as well as peripherals to their Reset state or when the user wishes to halt real-time executing of the target DSP to interrogate its registers and/or memory. All communication with the target DSP program model must be done while the target DSP is in the Debug mode of operation. Asserting a debug request is only required once to enter the Debug mode. Exiting the Debug mode is accomplished via a **G**O, **T**RACE, or **ST**EP command, or a **F**ORCE RU.

The B(break) argument asserts a debug request on the DSP and the register values will not be altered. Program execution can be resumed after a break by using a **G**O command. The R(reset to Debug mode) argument asserts the debug request on the DSP after the $\overline{\text{RESET}}$ pin is asserted and continues asserting the debug request until after the $\overline{\text{RESET}}$ pin is de-asserted, thus bringing the DSP into the Debug mode of operation at reset.

The **RU** (reset to User mode) argument asserts the $\overline{\text{RESET}}$ on the DSP pin only and does not assert a debug request, thus bringing the DSP out of reset into the mode specified by the external mode pins (which can be set by the user). The **S** (System) argument causes a **CF**ORCE **R** followed by a **F**ORCE **R** of the devices specified in the command. This command basically resets the Command Converter and then the target putting it into the Debug mode of operation.

Example 3-21   **F**ORCE Command Examples

fORCE **r**

Force a reset on the current default target DSP address. This command will destroy the contents of some registers since a hardware reset initializes them automatically.

fORCE **b**

Force the current default target DSP address into the Debug mode of operation. Program execution will halt and the DSP will enter the Debug mode waiting for command entry from the Host computer via the OnCE debug port.

fORCE **dv1,4,5 b**

Force target DSPs 1, 4, and 5 to halt DSP program execution and enter the Debug mode of operation for user commands.

fORCE **ru**

Reset the default target DSP into the User mode specified by its mode pins.

3.6.20   FRAME—Select C Function Call Stack Frame

FRAME [**dev_list**] [#fn]

The **FR**AME command is used to select the current call stack frame. It can be used in conjunction with the **WH**ERE, **DO**WN, and **UP** commands to display and traverse the C function call stack. After entering a new call stack frame using **FR**AME, that call stack frame becomes the current scope for evaluation. In other words, for C expressions, the **E**VALUATE command acts as though this new frame is the proper place to start looking for variables.

Example 3-22   FRAME Command Examples

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

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

3.6.21   GO—Execute DSP Program

GO [**dev_list**] [(from)location/**R**(reset)] [[#break_number] [:(occurrence)count]]

The **G**O command initiates program execution of DSP code. It can be used to start multiple ADMs or target systems simultaneously. The Command Converter monitor passes control to the user program on the ADM or target system until a breakpoint is reached or a break is asserted with the **F**ORCE command. The **G**O command can be used to resume execution after a force break.

Invoking the command with no argument will start execution at the current program counter value. Enabled break-points are examined at the end of every instruction cycle. If an address argument is included, program execution begins at the address specified. The R(reset) parameter will cause program execution to start from the user defined reset exception.

The optional #break_number 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. The occurrence count may only be used with the break_number parameter.

Example 3-23   GO Command Examples

go **dv0,1**

Start DSP program execution from the current address specified by the program counter of target DSP addresses 0 and 1. The target DSP addresses will stop at the first occurrence of any breakpoint set and report to the user the register results.

go **$100**

Start DSP program execution at program memory address hex.100 of the current default target DSP address.

go **100 #5 :3**

Start DSP program execution at location 100 (default radix) of the current default target DSP address and halt on the third occurrence of breakpoint number 5.

3.6.22   HELP—ADS User Interface Help Text

HELP [**dev_num**] [**command**/**reg**] (or) **command ?**

The **H**ELP command allows the user to review the Application Development System command set or a particular command's description. The **H**ELP line for command entry is designed so that a minimum amount of documentation is required to use the ADS interface program.

Invoking the command with a command name causes a summary of that command's parameters with a brief description and example to be displayed on the screen. If no command name parameter is included, the entire command set is displayed.

The **H**ELP command may also be used to review the register names and bit descriptions of peripheral control registers. Invoking the command with a register name causes the register contents of the default or selected target DSP address to be displayed on the screen.

Help for a particular command can also be obtained by entering the command and a question mark "?".

Example 3-24   HELP Command Examples

help

Display a summary of the available commands with their arguments.

help **a**

Display a summary of the **A**SM command, its arguments, and some examples.

help **dv1 omr**

Display a description of the OMR bits of target DSP address #1.

help **oscr**

Display a description of the OSCR bits of the default target DSP address.

break **?**

Display a summary of the **B**REAK command, its arguments, and some examples.

3.6.23 HOST—Change HOST Interface Address

HOST [**IO PC IO addr**] [**TIMEOUT value**]

The **HO**ST command allows the user to reconfigure the Host Interface card I/O address or set the timeout count for interaction with ADMs or target systems.

The I/O address may be started at $100, $200, or $300 only on the IBM-PC interface.

The TIMEOUT argument is used to change the host timeout value. In order for the user interface to avoid becoming hung up, it limits the time it will wait for the Command Converter to respond. If the Command Converter does not respond within this limit, an error message will be displayed. The default value of the timeout is 1 second.

Example 3-25 **HO**ST Command Examples

host

Display the current Host Interface Card address and timeout count being used.

host **timeout 3**

Change the host timeout to 3 seconds.

host **io $200**

Change the PC Host Interface address to $200.

Note: Changing the Host Interface address should be done only after changing JG1 of the IBM-PC Interface Card, otherwise the target DSP address will not respond to the ADS user interface commands.

3.6.24   INPUT—Assign Input File

INPUT [**dev_list**] [#(file number)] [**OFF**/[address **TERM**/filename [**-rd**/**-rf**/**-rh**/**-ru**]]]

The **I**NPUT command open files which will pass data to an ADM or target system whenever a user program enters the Debug mode of operation via a software breakpoint at a predefined address. Use of the keyword TERM assigns input from a terminal file which is created using a single line editor. The input data is in ASCII and is expressed in hexadecimal (**-rh**) unless decimal (**-rd**), unsigned (**-ru**), or floating point (**-rf**) radix is specified. Any number of files may be open for a target DSP input, up to host computer limits.

To successfully accomplish data transfers the following rules must be followed when the ADS user program calls the monitor file input subroutines.

1. It is necessary that a file be opened using the **I**NPUT command and that the DSP user program have a DEBUG software breakpoint opcode at the program address which is to enter the Debug mode of operation.

2. The file number and the word count which the user wishes to input data from must be in the X0 (R0 for DSP96000) register of the DSP prior to executing the DEBUG instruction.

3. The address where the data is to be put must be in the R0 (R1 for DSP96000) register and the memory map type (P = 0, X = 1, Y = 2) must be in the R1 (R2 for DSP96000) register.

The default path for each target DSP address can be changed with the **P**ATH command. If a transfer error occurs an error message will be displayed on the screen.

The ADS user interface program provides a way to specify repeated input values and sequences very similar to the DSP simulator program. A single data value may be repeated by specifying **#count** following the data item. A group of data items may be indicated by enclosing the group in parentheses. The entire group may then be repeated by placing #count immediately following the closing parenthesis. The parentheses may be nested. A closing parenthesis without a following repeat count will cause the data sequence within the parentheses to repeat forever.

Example 3-**26**   Examples of Input File Data

$ABCF

A single data item $ABDF

1FF#20

Repeat the data item 1FF twenty times.

(CC 50)#5

Repeat the sequence of data pairs CC 50 five times.

There are two levels of terminal data input capability provided by the ADS user interface. If the **I**NPUT command specifies term as the input filename, the ADS program enters a resident editor which allows creation of an input data file. The data file is given a temporary name, termxxxx.io (xxxx=0000-9999), and is saved on the disk at the termination of the **I**NPUT command. The entire contents of the input file may be specified in this manner, including any of the valid fields specified above.

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. Each time a t is encountered, the user will be prompted for a single data value from the terminal. The ADS user interface will read the input data using the radix option specified in the **I**NPUT command. Hexadecimal is the default input radix.

Example 3-**27**   Examples of Terminal Input Within an Input File

t#30

Request the next thirty input values from the user interactively or until an ESCAPE character is entered.

(t)

All input values are to come from the user interactively until an ESCAPE character is entered.

Example 3-28   Example Of Program That Resides on a Target DSP5600x

Input Requirements:
1. Load X0 with file number and word count

    a. For 16-bit devices, the file number will be in the upper byte and word count byte.

    b. For 24-bit devices, the file number will be in the upper byte and word count in the lower 2 bytes.

2. Load R0 with the starting location of the source block.

3. Load R1 with the source memory space:

    a. `MOVE #0,R1`—Move block to P memory space

    b. `MOVE #1,R1`—Move block to X memory space

    c. `MOVE #2,R1`—Move block to Y memory space

4. Execute DEBUG instruction to enter Debug mode of operation.

Code Example:
```
MOVE #$1000c,x0        ;file #1—input a block of 12 words DSP5600x only
MOVE #$0,R0            ;starting address of block is $0
MOVE #1,R1            ;in X memory space
DEBUG                 ;enter Debug mode
```

Example 3-29   Example Of Program That Resides on a Target DSP9600x

Input Requirements:
1. Load R0 with file number in upper byte and word count in lower two bytes.

2. Load R1 with the starting location of the source block.

3. Load R2 with the source memory space:

    a. `MOVE #0,R2`—Move block to P memory space

    b. `MOVE #1,R2`—Move block to X memory space

    c. `MOVE #2,R2`—Move block to Y memory space

4. Execute DEBUG instruction to enter Debug mode of operation.

Code Example:
```
MOVE #$10000c,R0       ;file #1—input a block of 12 words
MOVE #$0,R1           ;starting address of block is $0
MOVE #1,R2           ;in X memory space
DEBUG                ;enter Debug mode
```

Example 3-30 INPUT Command Examples

input

Display currently open input files for the current default target DSP address.

input **dv2 p:200 data.io**

Open "data.io" file for input to target DSP address #2 and place in the target DSP address 2 input list.
Note: p:200 is the address where the DEBUG opcode will reside.

input **dv2 off**

Close all current input data files assigned to target DSP address 2.

input **dv1 #2**

Display the file number 2 input filename for target DSP address 1.

input **p:400 data.io -rf**

Open "data.io" file for input to the default target DSP address as ASCII fractional and place in the default target DSP address input list.
Note: p:400 is the address where the DEBUG opcode will reside.

input **p:600 term**

Create a "term" file for input. The file number defaults to the first available number, one or greater.
Note: p:600 is the address where the DEBUG opcode will reside.

3.6.25   LIST—List Source File Lines

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

The **LI**ST 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 **LI**ST command without a parameter.

Example 3-31   **LI**ST Command 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 **lab_1**
List source or assembly corresponding to symbolic address lab_1.

3.6.26 LOAD—Load DSP Program

LOAD [**dev_list**] [**B**(byte wide) **address_offset**] (from) file

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

The **L**OAD command transfers DSP Macro-Assembler object files to the target DSP memory. The object module format (OMF) is defined in **Appendix A**, the Common Object File Format (COFF) is defined in **Appendix B**. Programs are loaded into the memory map and address specified by each data record. A directory path may be specified with the filename.

If only the file parameter is specified, then the user interface program assumes that the file is an OMF file. The object file may be in either the special ASCII OMF format, or in the DSP COFF format generated by the DSP Macro-Assembler. An OMF file may be created by using the DSP Macro-Assembler, or by using the **S**AVE command, or by some user generated method. If no filename suffix is specified, a OMF format ".lod" file is searched first and if not found, then a COFF format ".cld" file is searched. Loading a COFF format file replaces the target DSP symbolic debug information unless the **M** option is specified.

Programs are loaded into the memory map and address specified by each data record. A directory path may be specified with the filename. The default path for each target DSP address can be changed with the **P**ATH command.

If S is specified as the second of three parameters, the ADS interface program will load filename as an ADS state file. The ADS state file may be created using the **S**AVE s command. Loading the ADS state changes all ADS setups as well as registers and memory, to the previous definition saved in the state filename. If no filename suffix is specified, .adm is assumed.

If **M** is specified as the second parameter, the ADS interface program will load object file filename, .cld or .lod, without modifying the target DSP symbolic debug information.

If **D** is specified as the second parameter, the ADS 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.

If **B** is specified as the second parameter, the third parameter must be an address offset which is added to the OMF data record start address where data is to be loaded. The file must be in OMF and will be loaded byte wide sequentially incrementing the address counter on each byte load. The low order byte will be loaded first and the high order byte will be loaded last in each word. This is similar to the byte wide loading format of the bootstrap loader program on the DSP. This feature allows users to download programs into RAM and debug bootloader or overlay programs.

Example 3-**32** **L**OAD Command Examples

load **\source\testloop.lod**

Load "testloop.lod" file from directory "source".

load **lasttest**

Load "lasttest.lod" file from current directory. If not found look for "lastltest.cld" and load.

load **s lunchbrk**

Load "lunchbrk.adm", ADS state.

load **m test.cld**

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

load **d test.cld**

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

load **b 300 bootprog.lod**

Load the "**bootprog.lod**" file writing the least significant byte first and most significant byte last into each consecutive memory location of the target. The 300 argument is an address offset to be added to the starting location of memory specified in the data record.

3.6.27 LOG—Log Commands and/or Session

LOG [**dev_list**] [**OFF**] [**C**(commands)/**S**(session)] [**filename**] [**-o**/**-a**/**-c**]

LOG [**dev_list**] [**OFF**] **V**(source display status line)

The **LO**G command allows the user to record command entries only, or record all session display output to a file. 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. The user would otherwise only have access to the last 100 lines of output on the terminal. Since the output log files are in ASCII format, they may easily be printed or reviewed using an editor program.

Entering the **LO**G command with no parameters will cause a display of the currently opened log filenames. The keyword **OFF** is used to terminate logging. The **C** and **S** key characters are used to specify whether the logfile will contain only commands (**C**), or all session output (**S**).

The suffixes .cmd and .log are added, respectively, to the commands-only or session filename if no other suffix is specified. The default file path for each target DSP can be changed with the **P**ATH command. If a file currently exists with the filename specified the user will be prompted for an action of either appending the data to the file, overwriting the file, or aborting the command. The selection of the file action may be included in the command line using the **-o** (overwrite), **-a** (append), or the **-c** (cancel) argument.

Example 3-33 **LO**G Command 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**
Log all commands to filename "macro1.cmd".

log **off c**
Terminate command logging.

log **off**
Terminate all logging.

log **c macro1 -a**
Log all commands to filename "macro1.cmd". If a file with that name currently exists cancel the execution of the command.

3.6.28 MORE—Enable/Disable Session Paging Control

MORE [**OFF**]

The **M**ORE 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.

Example 3-**34** **M**ORE Command Examples

mORE
Turn on session display paging control.

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

3.6.29   NEXT—Step Over Subroutine Calls or Macros

NEXT [**dev_list**] [count] [**LI**(lines)/**IN**(inst)]

The **N**EXT command functions the same as the **ST**EP 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 **N**EXT command the specified number of times before execution terminates.

All breakpoints are ignored while the **N**EXT command is executing.

Example 3-35   **N**EXT Command Examples

---

next

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

next **10**

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

next **10 li**

Step over the 10 next source lines (if there is a source file associated with the current program counter).

3.6.30 OUTPUT—Assign Output File

OUTPUT [**dev_list**] [#(file number)... **OFF**]

OUTPUT [**dev_list**] [#(file number)] address filename/**TERM** [**-rd**/**-rf**/**-rh**/**-ru**] [**-o**/**-a**/**-c**]

The **O**UTPUT command opens disk files that will accept data from the target DSP. In order to do this, the user must first open the file for output and assign an address where a DEBUG software breakpoint will occur. Prior to executing the DEBUG instruction the user must load the X0 register with the file number to write to and the transfer count value, R0 with the address of memory and R1 with the memory map type (P=0, X=1, Y=2).

Use of the keyword TERM assigns the output to the display terminal rather than a file. The output data is in ASCII and is expressed in hexadecimal (**-rh**) unless decimal (**-rd**) or fractional (**-rf**), or unsigned (**-ru**) radix is specified. Any number of files may be open for an target DSPs output, up to the host computer limits.

For file output, it is necessary that a file be opened using the **O**UTPUT command. The file number in the output statement must correspond to the file number used in the DSP user program call to the Debug mode of operation. The default path for each target DSP address can be changed with the **P**ATH command.

If a file currently exists with the filename specified, the user will be prompted for an action of either appending the data to the file, overwriting the file, or aborting the command.

The selection of the file action may be included in the command line using the **-o** (overwrite), **-a** (append), or the **-c** (cancel) argument. This is useful when executing macro command files.

To accomplish data transfers successfully, the following rules must be followed when the ADS user program calls the monitor file output subroutines.

Example 3-36   Example Of Program That Resides on a Target DSP5600x

Input Requirements:
1. Load X0 with file number and word count

    a. For 16-bit devices, the file number will be in the upper byte and word count byte.

    b. For 24-bit devices, the file number will be in the upper byte and word count in the lower 2 bytes.

2. Load R0 with the starting location of the source block.

3. Load R1 with the source memory space:

    a. `MOVE #0,R1`—Move block to P memory space.

    b. `MOVE #1,R1`—Move block to X memory space.

    c. `MOVE #2,R1`—Move block to Y memory space.

4. Execute DEBUG instruction to enter Debug mode of operation.

Code Example:

```
MOVE #$1000c,x0      ;file #1—input a block of 12 words DSP5600x only
MOVE #$0,R0          ;starting address of block is $0
MOVE #1,R1           ;in X memory space
DEBUG                ;enter Debug mode
```

Example 3-37   Example Of Program That Resides on a Target DSP9600x

Input Requirements:
1. Load R0 with file number in upper byte and word count in lower two bytes.

2. Load R1 with the starting location of the source block.

3. Load R2 with the source memory space:

    a. `MOVE #0,R2`—Move block to P memory space.

    b. `MOVE #1,R2`—Move block to X memory space.

    c. `MOVE #2,R2`—Move block to Y memory space.

4. Execute DEBUG instruction to enter Debug mode of operation.

Code Example:

```
MOVE #$10000c,R0     ;file #1—input a block of 12 words
MOVE #$0,R1          ;starting address of block is $0
MOVE #1,R2           ;in X memory space
DEBUG                ;enter Debug mode
```

Note: For DSP56300 systems, you must clear Bit 17 of the Status Register (SR) before loading the X0 register with the proper value of file number and word count.

Example 3-**38** **O**UTPUT Command Examples

output

Display all output files currently open for all target DSP addresses.

output **dv2 p:300 admout**

Open file "admout.io" for logging of target DSP address 2 outputs.
Note: p:300 is the address where the DEBUG opcode is to reside.

output **dv2 off**

Close file currently open for target DSP address 2 outputs.

output **p:500 outfile.io -rd**

Open file "outfile.io" for logging of default target DSP address outputs in decimal radix.
Note: p:500 is the address where the DEBUG opcode is to reside.

output **#2 p:700 term**

Open file number 2 for the current default target DSP address as the terminal.
Note: p:700 is the address where the DEBUG opcode is to reside.

3.6.31   PATH—Define File Directory Path

PATH [**dev_list**] [pathname]

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

PATH **-**

The **P**ATH command allows the user to pre-define a directory path for file I/O for each target DSP address. This enables the user to effectively partition data files for each target DSP address in their appropriate subdirectory. Once a file is opened for **I**NPUT, **O**UTPUT, **S**AVE, or **LO**Gging, subsequent changes to the path will not affect the opened file. To change the path, the file must be closed and reopened with the new path name. 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 "**P**ATH **+**" form of the command. Each time the command is issued, the specified pathname, or comma-separated list of pathnames, is added to the current list. When searching for files, the ADS user interface program 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.

Example 3-**39   P**ATH Command Examples

---

path

Display the path for the current default target DSP address.

path **dv1..4 \;**

Define a path to the root directory for target DSP addresses 1, 2, 3, and 4. All subsequent commands with filename arguments will have this path string preceded to the filename when making an operating system call.

path **+ ..\test**

Add pathname "..\test" to the list of alternate source pathnames.

path **+ ..\test,..\help**

Add pathnames "..\test+ and +..\help" to the list of alternate source pathnames.

path **-**

Clear the list of alternate source pathnames.

3.6.32   QUIT—Exit ADS Program

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

The **Q**UIT command passes control back to the operating system and closes all logging files, assignment files, and macro files currently open.

Use the **SY**STEM command to exit the ADS program temporarily.

QUIT enable and **Q**UIT disable control the action taken by the ADS if an error occurs during the execution of a macro command. **Q**UIT enable specifies that the macro command is aborted and the ADS quits immediately with a non-zero exit status. **Q**UIT disable specifies that the ADS does not exit.

Example 3-40   **Q**UIT Command Examples

quit
Close all currently open files and return to the Operating System. Target DSP may be left running until the program is re-entered.

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

3.6.33   RADIX—Change Default Number Base

RADIX [**dev_list**][**B**(bin)/**D**(dec)/**F**(flt)/**H**(hex)/**U**(uns)] [reg[_block]/address[_block]]

The **R**ADIX command allows the user to change the default number base for command entry. 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 (').

Note: The default radix is hexadecimal when the user interface program is initially invoked.

This means that hexadecimal constants must be entered with a preceding dollar sign. Changing the default radix allows the user to enter constants in the chosen radix without typing the radix specifiers before each constant.

The **R**ADIX command also allows the user to select the display radix of registers and/or memory. The default display radix for registers and memory is hexadecimal.

The use of the values of hex A or B require the $ preceding the value, otherwise the values will be evaluated as the contents of the registers A or B, respectively.

Example 3-41   RADIX Command Examples

radix

Display the default radix currently enabled.

radix **h**

Change default radix entry to hexadecimal. Hexadecimal constant entries no longer require a preceding dollar sign, but any decimal constants will require a preceding grave accent (').

radix **f a**

Change the default display radix for the long register a to display a fractional value whenever the a register is displayed.

radix **u x:100..200**

Enable the display radix for X data memory block 100 to 200 to be unsigned when displayed on the screen.

3.6.34   REDIRECT—Redirect stdin/stdout/stderr for C Programs

REDIRECT [**dev_list**] **STDIN OFF**/file

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

REDIRECT [**dev_list**] [**OFF**]

The **RED**IRECT 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.

No stream file redirection occurs while stream option is disabled. See **STR**EAMS command, **Section 3.6.37** on page 3-66.

Example 3-42   REDIRECT Command Examples

---

redirect

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

redirect **DV0,3,4,5 off**

Cancel all stream redirection for specified target devices.

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 **STR**EAMS for more information.

3.6.35   SAVE—Save Memory To File

SAVE [**dev_num**] **S**(state)/address_block... filename [**-o**/**-a**/**-c**]

The **S**AVE command allows creation of an ADS state file from the current ADS state, or creation of an OMF file from specified memory blocks.

If S is specified as the second parameter, an ADS state file is created. It contains the entire ADS 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 ADS Interface program to store and load (see the **L**OAD s command description). The default suffix for an ADS state filename is .adm.

If memory blocks are specified (instead of S) the specified memory areas are stored in Macro_Assembler object module format so the file may be reloaded with the **L**OAD command. The default suffix for an OMF file is .lod. If a filename suffix of ..cld is explicitly specified, a COFF file will be created.

If a file currently exists with the filename specified the user will be prompted for an action of either appending the data to the file, overwriting the file, or aborting the command.

The selection of the file action may be included in the command line using the **-o** (overwrite), **-a** (append), or the **-c** (cancel) argument. This is useful when executing macro command files.

Example 3-43   SAVE Command Examples

---

save **p:0..$ff x:0..$20 session1**

Save all three memory maps to OMF file "**session1.lod**" of the current default target DSP address. Prompt for required action if file already exists.

save **s lunchbrk**

Save the default ADS state to filename "**lunchbrk.adm**". Prompt for required action if file already exists.

save **dv1 s lunchbrk.e1 -o**

Save the target DSP address 1 state to filename "**lunchbrk.e1**". Overwrite the current file "lunchbrk.e1" if it exists.

3.6.36   STEP—Step Through DSP Program

STEP [**dev_list**] [count] [**LI**(source lines)/**IN**(instructions)]

The **ST**EP command allows the user to execute count instructions or C source lines 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 **T**RACE command except that display occurs only after the count number of cycles or instructions have occurred.

Note: The address of the first instruction that is to be executed is in the OnCE Program Address Bus Decode Register (OPABD). If the Program Counter is changed before a **T**RACE or **ST**EP command is issued, the address of the Program Counter Register points to the instruction to be executed.

CAUTION

DSP5616x: When single stepping through a BRKcc instruction and the condition is true, the instruction immediately following the BRKcc instruction will be displayed by the ADS but will not be executed. Instead, the DSP will correctly execute the instruction at LA + 1. Single-stepping Tcc, REPcc or REP instructions with initial loop counter equal to zero may cause incorrect DSP operation.

The main difference between the **T**RACE and **ST**EP commands is the OnCE port trace counter is armed to trace one instruction in the Trace mode. The **ST**EP command arms the trace counter with the count instructions to be executed in real time before re-entering the Debug mode of operation and displaying the enabled registers and memory.

Example 3-44   **ST**EP Command Examples

---

step

Step one instruction at the current target DSP address and display enabled registers and memory blocks.

step **$50**

Execute hex $50 instructions and then display the enabled registers and memory blocks.

step **3 li**

Step over the next 3 source lines (for a source file associated with the current program counter).

step **dv2,5 5**

Execute 5 instructions on target DSP addresses 2 and 5 simultaneously and display the enabled registers and memory blocks of each when they have each completed their 5 instructions.

3.6.37   STREAMS—Enable/Disable Handling of I/O for C Programs

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

The **STR**EAMS 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.

See **RED**IRECT command, **Section 3.6.34** on page 3-63.

Example 3-45   STREAMS Command 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.

3.6.38 SYSTEM—Operating System Access

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

SYSTEM is a non-GUI command that allows operating system commands to be executed. The operating system commands may be executed as subprocesses of the ADS interface program or may be executed independently by temporarily exiting the ADS interface program. Invoking this command with no arguments will cause the ADS Interface Program to pass control to the Operating system but stay resident. To re-enter the ADS Interface Program the EXIT command must be invoked from the operating system command line.

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

When a **SY**STEM command is specified on the system command line, the user is prompted to "Hit return to continue..." before control returns to the ADS. 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 ADS without prompting the user. This may be useful in macro commands, allowing system commands to be used without requiring operator intervention.

Example 3-**46** **SY**STEM Command Examples

system

Temporarily exit the ADS interface program and go to the Operating System. To re-enter the ADS interface program invoke the EXIT command. All previous setups will not be altered.

system **dir**

Invoke the directory Operating System command from within the ADS Interface program:

```
system
dir *.io
del he.io
exit
```

Create a MS-DOS shell and temporarily exit the ADS user interface program. Execute two MS-DOS commands and re-enter the ADS user interface program using the EXIT command. The current state as well as opened files of all target DSP addresses will remain the same.

system **-c del e:\temp*.lod**

Delete the specified temporary files and continue without issuing the continuation prompt.

3.6.39   TRACE—Trace Through DSP Program

TRACE [**dev_list**] [**count**]/[**LI**(source lines)/**IN**(instructions)]

The **T**RACE command gives a snap shot of each instruction during program execution. This single stepping capability displays the enabled registers and memory blocks after each instruction until count instruction or C source lines are decremented to zero. To execute an instruction requires exiting the Debug mode of operation and entering the User mode of operation so that the instruction pipeline may be restored and any result of the traced instruction and subsequent instructions update the pipeline and machine state.

Note: The address of the instruction that is to be traced is in the OnCE Program Address Bus Decode Register (OPABD). If the program counter is changed before a **T**RACE or **ST**EP command is issued, the address of the program counter register will point to the instruction to be traced.

When single stepping through the BRKcc instruction and the condition is true, the instruction immediately following the BRKcc instruction is displayed by the ADS but is be executed. Instead, the DSP correctly executes the instruction at LA + 1. Single-stepping a Tcc, REPcc, or REP instruction with initial loop counter equal to zero may cause incorrect DSP operation.

The main difference between the **T**RACE and **ST**EP commands is the OnCE port trace counter is armed to trace one instruction in the Trace mode. The **ST**EP command arms the trace counter with the count instructions to be executed in real time before re-entering the Debug mode of operation and displaying enabled registers and memory.

Example 3-47   **T**RACE Command Examples

---

trace

Execute one instruction and display the enabled registers and memory blocks.

trace **20**

Execute 20 instructions and display the enabled registers and memory blocks after each instruction.

trace **5 li**

Execute the next 5 lines of source, and display enabled registers and memory blocks after each line.

trace **dv2,4 5**

Execute 5 instructions on target DSP address 2 and target DSP address 4 from their current program counter and display their enabled registers and memory blocks after each instruction.

3.6.40   TYPE—Display The Result Type of C Expression

TYPE [**dev_list**] {c_expression}

The **TY**PE 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.

Example 3-48   TYPE Command Examples

type **{count}**
Display the type and location of the variable count.

type **{0.5+i}**
Display the type of the given expression.

3.6.41 UNLOCK—Unlock Password Protected Device Type

UNLOCK **dev_type password**

The **UN**LOCK command provides password enabling for emulation of unannounced device types. Once unlocked, the device type may be selected with the **DE**VICE command.

Example 3-49 UNLOCK Command Example

unlock **56002 x51-234**

Enable device type **56002** for simulation using the password **x51-234**.

3.6.42   UNTIL—Step Until Address

> **U**NTIL   [**dev_list**] addr/line_number/address_label

The **U**NTIL 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 **ST**EP 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.

All other breakpoints are ignored while the **U**NTIL command is executing.

Example 3-**50   U**NTIL Command 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.

until **lab_2**

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

3.6.43   UP- Move Up the C Function Call Stack

UP [**dev_list**] [n]

The **UP** command is used to move up the call stack. It can be used in conjunction with the **WH**ERE, **FR**AME, and **DO**WN 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 **E**VALUATE command acts as though this new frame is the proper place to start looking for variables.

Example 3-**51**   **UP** Command Examples

up
Move up the call stack by one stack frame.

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

3.6.44 VIEW- Select Display Mode

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

The **V**IEW command changes the ADS display mode. There are three display modes: Assembly, Source, and Register. See Display Modes in **Section 1** for a description of the display modes.

If the **V**IEW 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 following order: Source—Assembly—Register. The same results can be obtained by typing **Ctrl-w**.

Example 3-52 VIEW Command Examples

view

Cycle to the 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.

3.6.45 WAIT—Wait Specified Time

WAIT   [[**dev_list**] **B**(break)]/**count**(seconds)]

The **W**AIT command pauses for count seconds or until the user types any key before continuing to the next command. If the **W**AIT command is entered without a count parameter, the command will terminate only if the user types a key. This instruction is useful when executing a macro file and the current display on the screen needs to be examined before executing further instructions from the macro file. The **B** option causes a pause until all of the specified devices have entered the Debug mode. This option is useful when executing a macro file where the devices must hit breakpoints or complete steps or traces before the next command is accepted.

Example 3-53   **W**AIT Command Examples

wait

Wait for a key stroke from the keyboard before executing any further instructions.

wait **10**

Wait ten seconds before executing another command from the keyboard.

wait **dv0..3 b**

Wait until devices 0,1,2 and 3 have all entered the Debug mode.

3.6.46   WATCH—Set, Modify, View, or Clear Watch Item

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

WATCH [**dev_list**] [#wn] **OFF**

The **WAT**CH 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 **WAT**CH without any parameters, the watch list is displayed.

Example 3-54   WATCH Command 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.

3.6.47   WASM—GUI Assembly Window

WASM [dev_list] [**OFF**]

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

Example 3-**55   WA**SM Command Examples

wasm

Open an assembly window for the current device.

wasm **dv0..1**

Open an assembly window for devices dv0 and dv1.

wasm **off**

Close the assembly window for the current device.

3.6.48   WBREAKPOINT—GUI Breakpoint window

WBREAKPOINT [dev_list] [**OFF**]

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

Example 3-**56**   **WB**REAKPOINT Command Exampels

wbreakpoint
Open a breakpoint window for the current device.

wbreakpoint **dv0,3,4**
Open a breakpoint window for the listed devices.

wbreakpoint **off**
Close the breakpoint window for the current device.

3.6.49   WCALLS—GUI C Calls Stack Window

WCALLS [dev_list] [**OFF**]

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

Example 3-57   **WC**ALLS Command Examples

wcalls

Open a C call stack window for the current device.

wcalls **off**

Close the C call stack window for the current device.

3.6.50 WCOMMAND—GUI Command Window

WCOMMAND [**OFF**]

WCOMMAND is a GUI command that opens the Command window. Only one Command window may be opened even for debugging target systems with multiple DSPs.

The Command window is shared between all target DSP devices. All commands affect the current default device unless specifically addressed to other device(s). The prompt on the command entry line indicates the current default device.

Example 3-**58** **WCO**MMAND Command Examples

---

wcommand
Open a Command window.

wcommand **off**
Close the Command window.

3.6.51   WHERE—GUI C Calls Stack Window

WHERE [dev_list] [[+/-]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.

Example 3-59   **WH**ERE Command 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.

3.6.52 WINPUT—GUI File Input window

WINPUT [dev_list] [**OFF**]

WINPUT is a GUI command that opens an input window. The input window lists all simulated input assignments for the specified device. Multiple device windows may be opened for debugging target systems with multiple DSPs.

Example 3-60   WINPUT Command Examples

winput

Open an input window for the current device.

winput **off**

Close the input window for the current device.

3.6.53   WLIST—GUI List Window

WLIST [win_num] [**OFF**]

WLIST is a GUI command that opens a list window. A list window is used to view a test file within the ADS environment. Multiple list windows may be opened for viewing multiple text files.

Example 3-61   WLIST Command 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.

3.6.54   WMEMORY—GUI Memory Window

WMEMORY [dev_list] [win_num] [space [addr]/[**OFF**]]

WMEMORY is a GUI command that opens a memory window. Multiple device windows may be opened for viewing and changing separate memory areas and spaces at the same time, and for debugging target systems with multiple DSPs.

The memory window is positioned initially to view the address specified by the addr field. The addr field may also be used in conjunction with the space field to select the exact memory space required.

Example 3-62   WMEMORY Command 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 **dv2 win3 x $4100**

Open a memory window for memory space x with a window number of 3 for the current device. Scroll window to display address $4100.

wmemory **off**

Close all memory windows for the current device.

wmemory **win3 off**

Close memory window 3 for the current device.

3.6.55   WOUTPUT—GUI File Output Window

WOUTPUT [dev_list] [**OFF**]

WOUTPUT is a GUI command that opens a file output window. The output window displays the simulated output assignments for the specified device. Multiple output windows may be opened for debugging target systems with multiple DSPs.

Example 3-63   WOUTPUT Command Examples

woutput

Open an output window for the current device.

woutput **dv1 off**

Close the output window for the device dv1.

3.6.56 WREGISTER—GUI Register Window

WREGISTER [dev_list] [win_num] [**OFF**]

WREGISTER is a GUI command that opens a register window. Multiple register windows may be opened to display and change separate blocks of registers at the same time. Multiple device windows may be opened for debugging target systems with multiple DSPs.

Example 3-**64**   **WR**EGISTER Command 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 **dv1 off**

Close all register windows for the device dv1.

3.6.57   WSESSION—GUI Session Window

WSESSION [**OFF**]

WSESSION is a GUI command that opens a Session window. Only one Session window may be opened even for debugging target systems with multiple DSPs. All session output from all target devices is written to the Session window. A message is output to indicate which target device produced the following output.

Example 3-65   WSESSION Command Examples

wsession
Open a Session window for the current device.

wsession **off**
Close the Session window.

3.6.58   WSOURCE—GUI Source window

WSOURCE [dev_list] [**OFF**]

WSOURCE is a GUI command that opens a Source Code window. Multiple device windows may be opened for debugging target systems with multiple DSPs.

Example 3-66   WSOURCE Command Examples

wsource
Open a Source window for the current device.

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

3.6.59 WSTACK—GUI Stack Window

WSTACK [dev_list] [**OFF**]

WSTACK is a GUI command that opens a Device Stack window. Multiple device windows may be opened for debugging target systems with multiple DSPs.

Example 3-67   WSTACK Command Examples

wstack

Open a Stack window for the current device.

wstack **off**

Close the Stack window for the current device.

3.6.60   WWATCH—GUI watch window

WWATCH [dev_list] [win_num] [#wn] [radix] reg/addr/expression

WWATCH [dev_list] [win_num] [#wn] [**OFF**]

WWATCH is a GUI command that opens a Watch window. Multiple device windows may be opened for debugging target systems with multiple DSPs.

Example 3-68   WWATCH Command 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.

3.7 DEBUGGING C PROGRAMS

The ADS user interface 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 ADS capabilities are available while debugging C programs.

3.7.1 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, along with the result type of a C expression.

3.7.2 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 ten 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 ADS software, only the lower 48 bits of register A would be used.

Note: The $ in non-C_expression evaluation is used to designate a hexadecimal value.

3.7.3   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.

3.7.4 Compiling a Program for Debugging

To use the C debugging features included in the ADS 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.

3.8 C DEBUGGING COMMANDS

Certain commands (**WH**ERE, **UP**, **DO**WN, **FR**AME, **STR**EAMS, **RED**IRECT, and **TY**PE) exist specifically for debugging C programs, while other commands (**B**REAK, **E**VALUATE, **FI**NISH, **G**O, **N**EXT, **ST**EP, **T**RACE, **U**NTIL, and **WAT**CH) are useful in debugging C programs, but are also used in assembly language debugging.

To eliminate duplicated functionality, the **E**VALUATE command is used in C debugging as the **C**HANGE, **D**ISPLAY, and **E**VALUATE commands would be used in assembly language debugging. For instance, to display a C variable, **E**VALUATE that variable. To change the value of a C variable, **E**VALUATE an expression that has an assignment to that variable. **E**VALUATE 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."

3.9 EXAMPLE DEBUGGING SESSIONS

This section goes over debugging two simple C programs, one that performs a binary search on a constant array, and one that performs a binary tree traversal, writing each element on the tree out to disk. These two example programs contain examples of many commands useful for C debugging. The two programs are both included in source form with the ADS distribution. These files are binsbad.c, binsgood.c, and trav.c. The file binsbad.c is a version of a binary search program with bugs in it. The file binsgood.c has no known bugs. The file trav.c is used in the second session to illustrate the use of more debugging commands.

3.9.1   Binary Search Example

For brevity, the full text of the source code for the binary search program is not shown here. You should compile the file binsbad.c, start the ADS software, and follow through the instructions in the rest of this section step by step, so that what is happening in each step is clear to you.

Assuming you have already compiled the binsbad.c source file with the debugging flag, started the ADS host software, and reset the DSP, you should now do:

> **l**oad **binsbad.cld**

to load the COFF file that resulted from compiling binsbad.c.

The first thing we will do is step over the program text at a very high level to verify that the program appears to be working. We will then step through in detail and find that there are problems which result in both poor efficiency, and improper execution.

First, set a breakpoint at function "main".

> **b**reak **al main**

Now that you have set a software breakpoint that stops at "main", start execution of the DSP program.

> **g**o

After the breakpoint is reached, we want to view the source code for the program and get some indication of where we stopped when we hit the breakpoint for "main".

> **v**iew **s**

By using the **N**EXT command we can step over the next two lines of source code (the call to the function "find_keyword" and the assignment to the variable "t").

> **n**ext **2**

So, "find_keyword" has been called with "typedef" as an argument. The index returned by "find_keyword" has been assigned to "indx", and the token value for the "indx" item of "key_toks" has been assigned to "t". Now display what value "t" has.

> **e**valuate **{ t }**

You should see the result "enum token_t: Typedef". So, "t" is of type "enum token_t", and it's value is "Typedef", the enumeration value that corresponds to the string "typedef". Now view the source again.

> **v**iew **s**

Repeat the procedure we have just gone through to examine the results of each additional call to "find_keyword".

While viewing the source, display the entire "key_toks" array.

> **e**valuate **{key_toks}**

You should see the display of an entire array of structure items. Display the contents of a single structure, for example the one at index 7.

> **e**valuate **{key_toks[7]}**

You should see the structure entry for "do." Arbitrarily complex expressions can be used. Here is an illustration of the "#" operator.

> **e**valuate **{key_toks#10}**

You should see the display of the first 10 elements of the "key_toks" array. Now, on to debugging. Repeat the procedure for loading the program, so that we can start over from the beginning. Run until you hit the breakpoint at "main".

> **l**oad **binsbad.cld**
>
> **g**o

View the source, and then step into the "find_keyword" function.

> **v**iew **s**
>
> **st**ep

Now, step until the indicator is pointing at the line that begins the while loop.

After stepping to the while loop, use the **WH**ERE command to find out where you are, and how you got there.

> **wh**ere

You should see a list containing the function you are in, and the function "main". This list tells you that the function you are in was called from "main" with the parameter listed. The indicator tells you which level of the call stack is currently selected. By selecting other levels of the stack (with up, down, and frame), you can examine local variables within those functions. You can also view the source for those functions, in which case the indicator will point at the place where you will return to when you get back up to that function. For instance, view the source code again, and then move up a stack level, and then finally down a stack level. Note the difference in the source display as you move up and down levels.

> **v**iew **s**
>
> **up**
>
> **do**wn

Now, examine the values of the variables low and high.

> **e**valuate **{low}**
>
> **e**valuate **{high}**

Since low and high are the bounds of our search within the area, something should immediately strike you as being wrong. In C, array indices begin at zero, and the last

element of an array is indexed by the number of elements in the array minus one. However, in this program, "high" has been set to the number of elements in the array. It is off by one, and this could cause it to access elements that are not a part of this array. Modify the source program so that one is subtracted from the value assigned to "high". See the file binsgood.c if you need help. Recompile the program and repeat the steps necessary to get you back into the "find_keyword" function of the newly compiled program.

After repeating the previous steps you should be back at the beginning of the while loop and should be able to examine the variables "low" and "high" and see that they now have correct values. Now we want to add a few expressions to our watch list so that we can view them as we step over lines.

> **wat**ch **{med}**
>
> **wat**ch **{low}**
>
> **wat**ch **{high}**
>
> **wat**ch **{key_toks[med].keyword}**

Let's go into the register view and step line by line through the source and watch the progress of our variables.

> **v**iew **r**
>
> **n**ext **li**

Repeat the **N**EXT command several times and watch what happens to each of the variables. The form of the **N**EXT command being used (**N**EXT **li**), steps line by line, rather than instruction by instruction (which is the default for the **N**EXT command when not viewing source). Repeat the **N**EXT command until you see "Expression out of scope" as a result of the watch expressions, indicating that you have left the function that you were stepping through. View the source.

> **v**iew **s**

If you were watching carefully while you used next to go over the lines, you might have noticed that "high" was not changing by much each iteration of the loop. Examining the source carefully you'll see "high = high - 1" where it should actually read "high = med - 1". Correct this in the source code and recompile. Repeat the previous steps needed to get into the function "main".

If you next over the lines while viewing source, and stop after the call to "find_keyword" in which "while" is the parameter, you'll notice that the variable "indx" was assigned a value of "-1", indicating that "while" wasn't found in the list.

Tracing through this call to "find_keyword" carefully, it becomes apparent that the expression in the while loop should be "low <= high", rather than "low < high". Correct this mistake, recompile, and then repeat the steps necessary to get into the function "main".

Step into the first call to the "find_keyword" function, after the assignments to "low" and "high", but prior to the beginning of the while loop. Set a breakpoint that will stop execution at line 62 when "low" and "high" have the same value.

break **al @62 t({low==high})**

Recognize that when we set the breakpoint for "main", we simply used its name (since it is a function), but in this case we are using a C expression enclosed in braces. Unless you are specifying a function name to break on, you should use the expression format as above.

Execute the program until you hit the breakpoint.

go

Examine the values of "low" and "high".

Now, experiment on your own to get a better feel for debugging C programs within the ADS software. At some point you will want to do the next tutorial, which will introduce a couple more commands that can be useful in debugging C programs.

3.9.2   Recursive Binary Tree Traversal Example

This example introduces some more C debugging commands, this time stepping through a program that doesn't have any known bugs, to illustrate the result of using the debugging commands.

For brevity, the full text of the source code for the tree traversal program is not shown here. You should compile the file trav.c, start the ADS software, and follow through the instructions in the rest of this section step by step, so that what is happening in each step is clear to you.

Assuming you have already compiled the trav.c source file with the debugging flag, started the ADS host software, and reset the DSP, you should now do:

load **trav.cld**

to load the COFF file that resulted from compiling trav.c.

The file trav.c contains code to open a text file, traverse a binary tree, and write the numbers contained in the nodes out to a file. If there are any problems in doing this, it writes an error message out to stderr and exits. By using the **STR**EAMS command, you can determine whether C input and output are currently being handled by the host software.

streams

By default, input and output are handled by the host software. One reason you might want to disable the handling of input and output by the C program is if you have written

your own send and receive functions that were compiled into your C program, and you have special communication needs that are handled in those routines (like writing out to a peripheral on the DSP).

REDIRECT the output to stderr so that it gets written into a file.

> **red**irect **stderr stderr -o**

This command will open the file stderr.cio on the host side, and force everything that is written to stderr to be written to this file. The -o flag specifies that stderr.cio file should be overwritten if it already exists.

Now, set a breakpoint at main, run until that breakpoint is hit, and view the source.

> **b**reak **al main**

> **g**o

The main routine contains calls to create_tree_node to create nodes for our tree. It builds a simple tree, and then calls write_tree_to_file to write the tree out.

Step over all the calls to create_tree_node.

> **n**ext **8**

Use the **E**VALUATE command to add yet another node onto this tree. This illustrates the ability to call C functions from the command line.

> **e**valuate **{tree->right->right->right = create_tree_node(9)}**

Now, step over the call to write_tree_to_file, and the tree will be written out to the file.

> **n**ext

Examine the file "treefile.txt" to see the results.

SECTION   4

GRAPHICAL USER INTERFACE

4.1   INTRODUCTION

This chapter describes the DSP ADS Graphical User Interface (GUI). Use of each operation is described, using illustrations of the windows, dialog boxes, and expected output results from the operation. Important features are indicated on each illustration.

4.2   HOST SYSTEM REQUIREMENTS

The graphic interface version of the DSP ADS requires the following minimum system configuration:

- **Sun Workstation**—Any SPARCstation 2 or above, with at least 10 MB of free disk space, or

- **Hewlett Packard Workstation**—Any HP7xx series workstation, with at least 10 MB of free disk space, or

- **PC-compatible Computer**—An 486 or later system, minimum 8 MB RAM, color SVGA display at 800 by 600 resolution or better, approximately 10 MB free disk space. A high resolution SVGA display (1280×1024) with a 17" monitor will give a more productive working environment.

4.3   PLATFORM SPECIFICS

The operation of the ADS under the GUI 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 ADS 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.

4.4 GENERAL WINDOW BEHAVIOR

Under Windows, all the GUI 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.
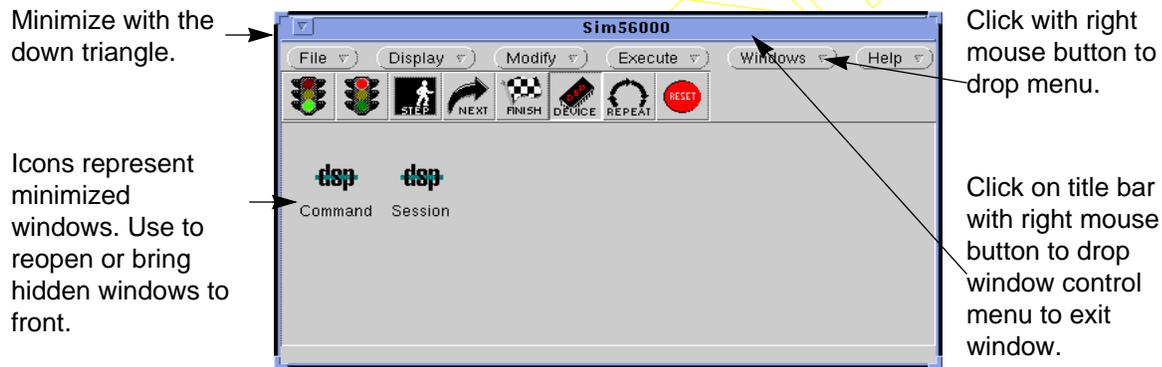
Figure 4-1 Main Window for Sun SPARCstation 2

4.4.1 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 Unix 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

Pin if required.

Scroll horizontally through directories— parents to left, subs to right.

Select a directory in one column gives list of files/directories in next.

Scroll through list of files and directories.

Enter file name if desired.
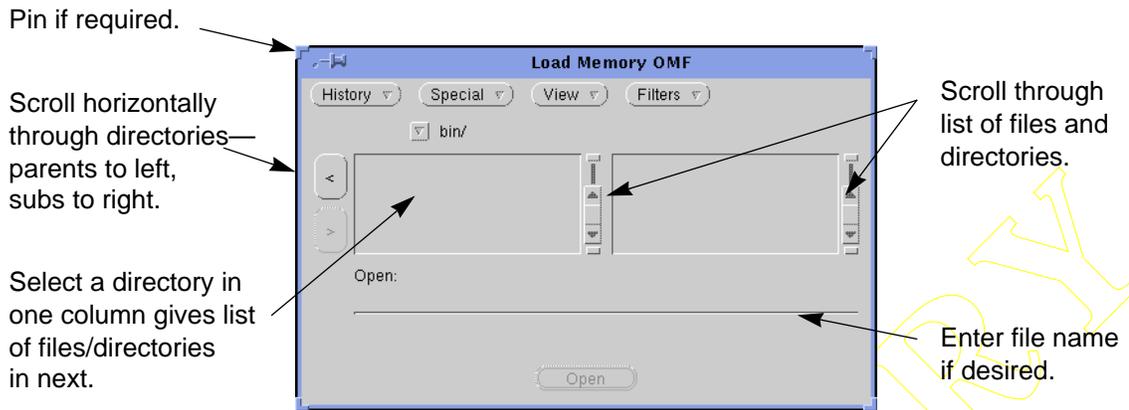
Figure 4-2 Sun File Chooser Dialog Box

Type name or wildcard filter.

Scroll through files or directories.

Click on required file.

Select files to view from pull-down list.

[OK] to open, [Cancel] to dismiss.

Open folder is current directory; above is path, below is sub-directories.

Select required drive.
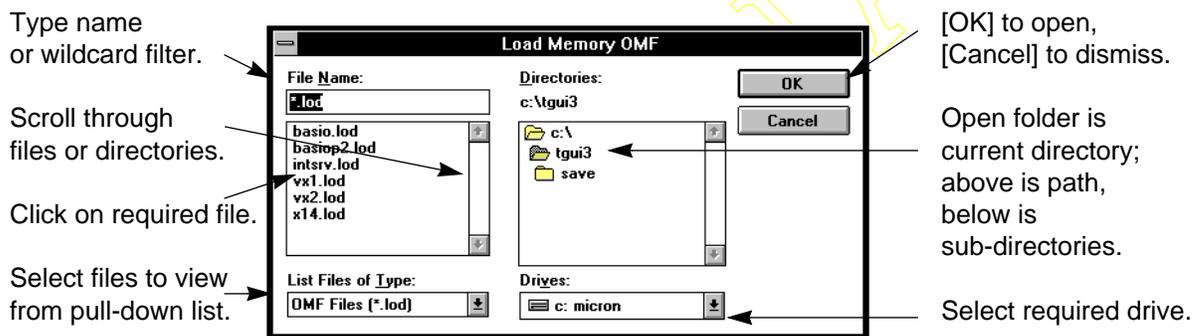
Figure 4-3 Windows File Chooser Dialog Box

4.4.2 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 repeated operations.

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.
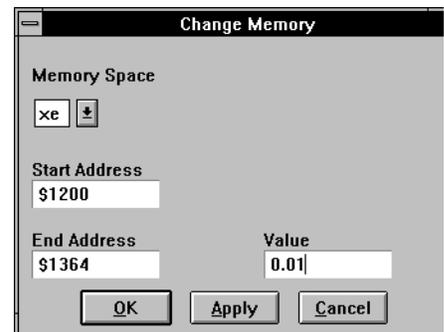
Under Motif, the same effect is achieved in a different way. There is only one button, [Apply], which performs the appropriate operation, and then dismisses it. The dialog box can be made (semi) permanent by clicking on the pin in the top left corner, so it will not be dismissed after clicking the [Apply] 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 without taking any action, double-click on the pin (i.e., 'pin and release', and the dialog box closes).

4.4.3   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 items to be added.

- 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.

4.5   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.

4.5.1 GUI Structure

The GUI provides an interface to the command line debugger, generating commands from the user actions, and interpreting the responses.
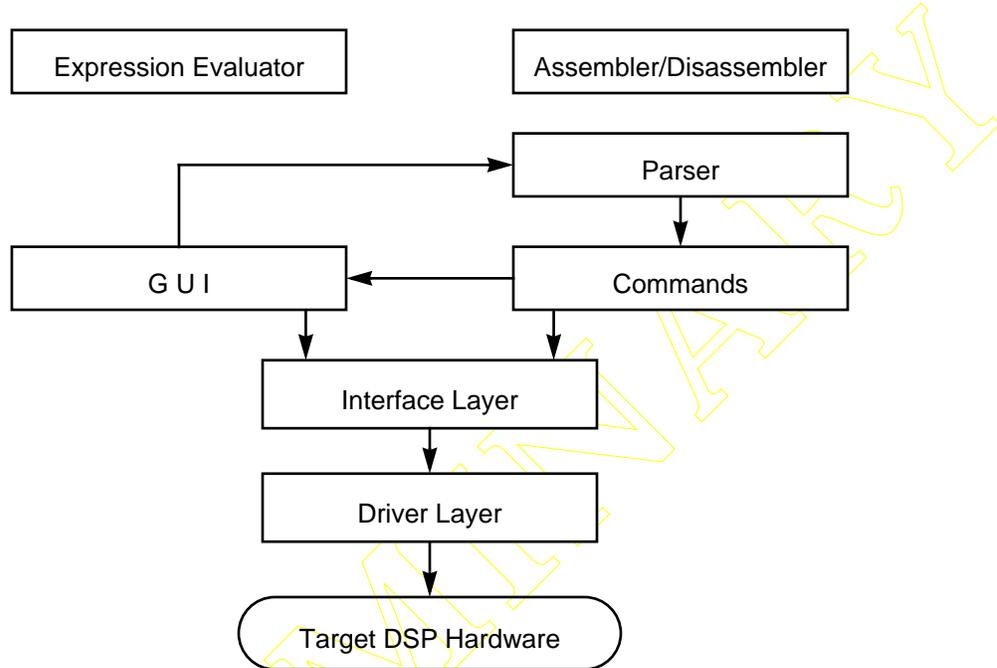
Figure 4-4 GUI Interface to ADS

4.5.2 Starting the ADS

At system start-up, the main window opens. This provides the menu for the system, and the tool bar for convenient access to frequently-used operations.

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.

4.5.3 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.

4.5.4 Loading Object Files

The development system can load object files in COFF and OMF formats into 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 (i.e., 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.

4.5.5 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 Halt 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), and 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.

4.5.6   Examining and Changing Registers

The registers can also be monitored with the Register window (Window menu, Register option). All registers associated with the core or a specified peripheral can be displayed in a window, scroll to view those you want. Multiple windows may be opened for each device to view registers in different peripherals. Registers can also be modified, as with the Memory window; see also Modify menu, Register option.

4.5.7   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 subroutine like the rest of the code, and stops after each instruction in the subroutine. Next treats the subroutine as one instruction, and stops after it is finished.

4.5.8   Device Selection

The debugger can support multiple DSP devices, up to thirty-two 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.

4.5.9 Breakpoints

There are two types of breakpoint used by the ADS:

- A software breakpoint is an instruction placed in the DSP code which when executed (possibly conditionally, for example if carry is clear) places the device in Debug mode. Software on the host then performs further checks, and if satisfied performs the action specified for the breakpoint. A software breakpoint may also be associated with end of file on simulated input.

- A hardware breakpoint involves circuitry built into the DSP which monitors address lines, etc., and when the specified conditions are met, places the device in Debug mode. This monitoring involves no run-time penalty. Software on the host then performs further checks, and if satisfied performs the action specified for the breakpoint.

Both types of breakpoint may have an associated count. If specified, the breakpoint is ignored until the Nth occurrence. This count is reinitialized each time execution is initiated.

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. All breakpoints are listed in the Breakpoint window.

These are HALT breakpoints—the program is halted and control returns to the user. With the Execute menu, breakpoints may have several other actions associated with them. For example, incrementing a counter (four are available) tracks 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 hardware breakpoints in the data, so that when a specific 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 possible to set multiple breakpoints on a single location or event, to specify multiple actions—say increment a counter, display some values, and halt—to be taken at the same time.

4.5.10   Simulated Input and Output

DSP programs do not usually exist in isolation. It is necessary 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 a DEBUG instruction; every time that instruction is executed with the appropriate parameters, the value returned to the program is provided by the data file. Input from a file is directed to a memory location. This simulated input represents a data stream, so that each access gets the next item. Each entry in the file can be read once, and only once, and can not be skipped.

Simulated output is similar. When a specified DEBUG instruction is executed, a record is written to the output file, consisting of the data value specified.

4.5.11   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.

4.5.12   Command and Session Windows

There are two windows which are involved in most GUI operations.

- **Command Window**—Most GUI operations generate commands which are passed to the debugger and stored in a history buffer that 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.

- **Session Window**—Whenever the debugger generates output, it is written to the Session window, the main screen for the current device. 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.

4.5.13   Command and Session Log Files

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 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.

Note: All activity in the Command and Session windows may be recorded to a log file. See Log in the File menu.

4.5.14   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 GUI 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.

4.5.15   Input Conventions

- Numeric values (including addresses) are assumed to be in the default radix unless prefixed by a radix identifier: ` (grave accent) = decimal, $ = hexadecimal, % = binary.

- Many input fields accept expressions—either C expressions, or DSP Assembler expressions. Program symbols may be used in expressions if debug information

has been loaded. C expressions must be enclosed in brackets {}. See **Section 3.4.17** for expression information.

• Program locations are often specified with the mapping in one field and the address in another field. Sometimes, one field is used for input. In this case, the location may be input as a program line number ('@116'—see **Section 3.3**) or an address including the memory type (e.g., p:$4117).

4.6 FILE MENU

The File menu handles all operations associated with file handling. The operations covered are:

File
Path ▶
Load ▶
Save ▶
Input ▶
Output ▶
IO Streams ▶
IO Redirect ▶
Log ▶
Macro...
About...
Preferences...
Exit

• **Path** 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 load object modules into memory, write selected memory areas out into object modules, and save and reload the entire status of the development system.

• **Input** and **Output** provide simulated data for a program, and saves output produced by a program.

• **IO Streams** and **IO Redirect** provide a basic stream IO 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.

• **Log** permits Command and Session windows to 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 saving of window positions.

• **Exit** allows the user to leave the debugger.

4.6.1   File//Path//...

A separate set of file search paths 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 one or more 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 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.

Shows path before
leftmost column. Click to
open pull-down list and
select a directory from
list.

Dialog menus to
select device,
directory list order,
previously-visited
directories.

Use '>' and '<' buttons
to scroll horizontally
through directory
levels.

Displays
currently selected
directory. May
also click and
enter path
directly.

Adjust dialog box
size for number
of columns
shown and length
of list.

Single click and
SELECT sets path
to last selected
directory.

Single click and Open
(or double click) lists
directories available in
last selected directory.

Figure 4-5 File//Path/Set, Add Dialog Box

4.6.2 File//Load//Memory COFF, Memory OMF
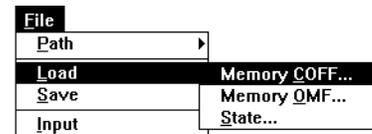
The File//Load//Memory COFF, Memory OMF menu
items read object modules in OMF or COFF format into the DSP memory for the current
device (see Modify//Device//Set Default). Complementary functions
File//Save//Memory COFF and Memory OMF are available to preserve memory
contents in OMF or COFF files, which 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.

Select from loading memory, debug symbols, or both. Usual operation is to load both. Might load symbols only after loading patched memory saved after previous debug session.

Load OMF file opens file chooser dialog box directly.

Initial dialog box is only used with COFF files to select the class of data to be loaded, before starting the file search dialog box.

May enter name of load file manually if desired, [OK] to load.

Click to open file access dialog box.
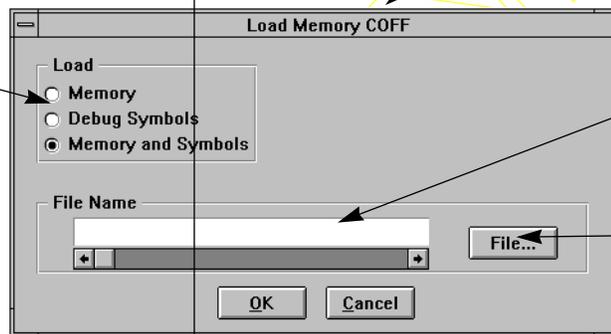
May type file name directly. May include drive and path or use path shown in rest of dialog box.

Select directories from list by double-click to build path.
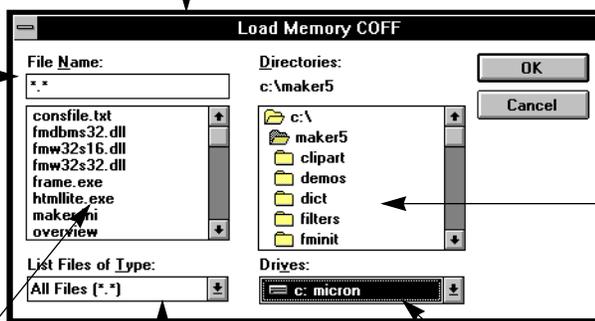
Double-click on required file or Single-click and [OK] to load.

Select desired file type from pulldown list to specify which files are displayed in list.

Select drive from pull-down list.

Figure 4-6 File//Load//Memory COFF, Memory OMF Dialog Box

4.6.3 File//Save//Memory COFF, Memory OMI

The File//Save//Memory COFF, Memory OMI menu items save the contents of a single memory block into DSP COFF or ASCII OMF files that can later be reloaded with the ADS 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.

Both the OMF and COFF options open the SAVE Memory dialog box.

Select the required memory space from the pull-down list.

Tab to (or click on) the address range fields and enter the memory range to be saved. '$' prefix = hexadecimal.

Click here to enter the required file name manually. Device and path may be specified. If omitted, will use working directory or alternate source path.
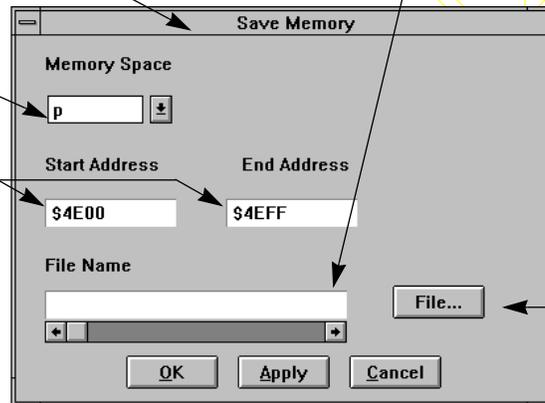
Or click to open file chooser.

Enter file name directly to specify new file name.

Double-click on required file or Single-click and [OK] to save. Another dialog box will open to confirm existing file is to be replaced.

Select desired file type from pulldown list to specify which files are displayed in list.
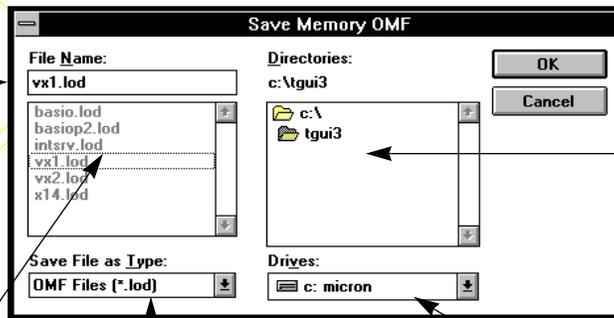
Select directories from list by double-click to build path.

Select drive from pull-down list.

Figure 4-7 File//Save//Memory COFF, Memory OMF Dialog Box

4.6.4 File//Save//State, File//Load//State

The Load and Save State menu items allow the state of the entire ADS system 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.
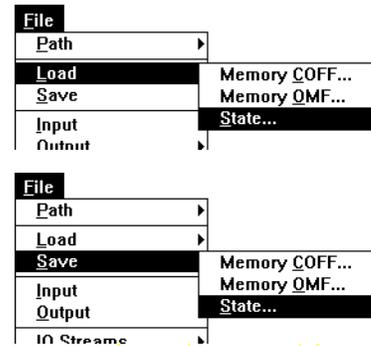
This may be used in several ways. A protracted development session may be saved before a break, and reloaded after the interruption to be continued where it was left off. Alternatively, if a particular part of a program is proving troublesome, the state may be saved just before the problem area, simplifying the setup for repeated attempts to isolate the problem. Or a set of standard routines and data areas may be pre-loaded in a GUI 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.

Enter file name manually if desired. State files use extension '.SIM'.

Select directories from list by double-click to build path.

Double-click on required file or Single-click and [OK] to save. On SAVE, another dialog box will open to confirm existing file is to be replaced.
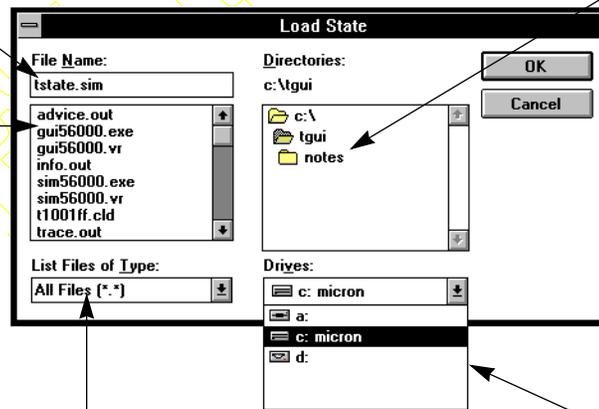
Select desired file type from pulldown list to specify which files are displayed in list.

Select drive from pull-down list.

Figure 4-8 File//Load//State, File//Save//State Dialog Box

4.6.5 Ḟile//Ḻnput//Ọpen

Ḟile//Ḻnput//Ọpen reads data from the terminal or a file to
provide simulated input to a memory location in the default
device. The input file is associated with a DEBUG instruction
in the code. More than one input file may be associated with a single DEBUG instruction.
When DEBUG is executed, registers must be set as shown in **Table 4-1**.
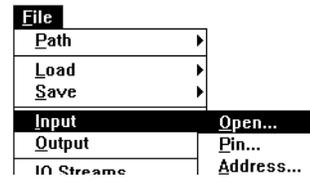
Table 4-**1**   Register Requirements for Simulated Input

Data Item	DSP96xxx	DSP56xxx
Input File Number and byte count	R0	X0
Input Address	R1	R0
Memory map (P=0, X=1, Y=2)	R2	R1

Each time the DEBUG instruction is executed, a data value is obtained from the input file
associated with the file number specified, and stored in the indicated location. The
format of the data file and programming requirements is documented in the command
line Input **Section 3.4.24**.

Specify Input # for this input file. Next available number is offered.

Enter address of DEBUG instruction which requests input from this file.

Select default radix used in data file. Radix specifiers may also be used in file.

Select data from file or entered at terminal.

May specify file name manually. Default file type is '.IO'.

Click to open file chooser.
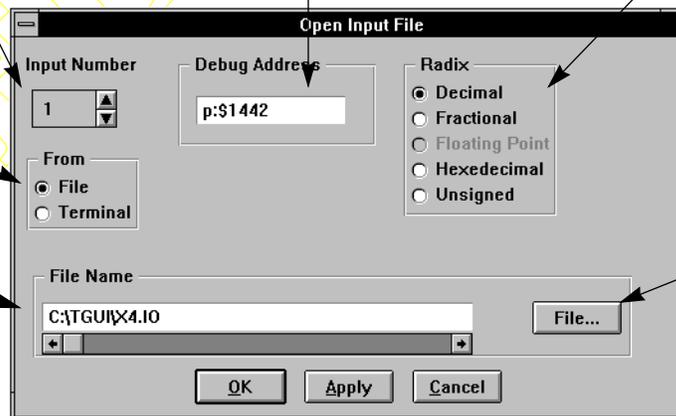
Figure 4-**9** Ḟile//Ḻnput//Ọpen Dialog Box

4.6.6   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>-Clicks, and Click-and-Drag. Then close all selected inputs by clicking [OK].
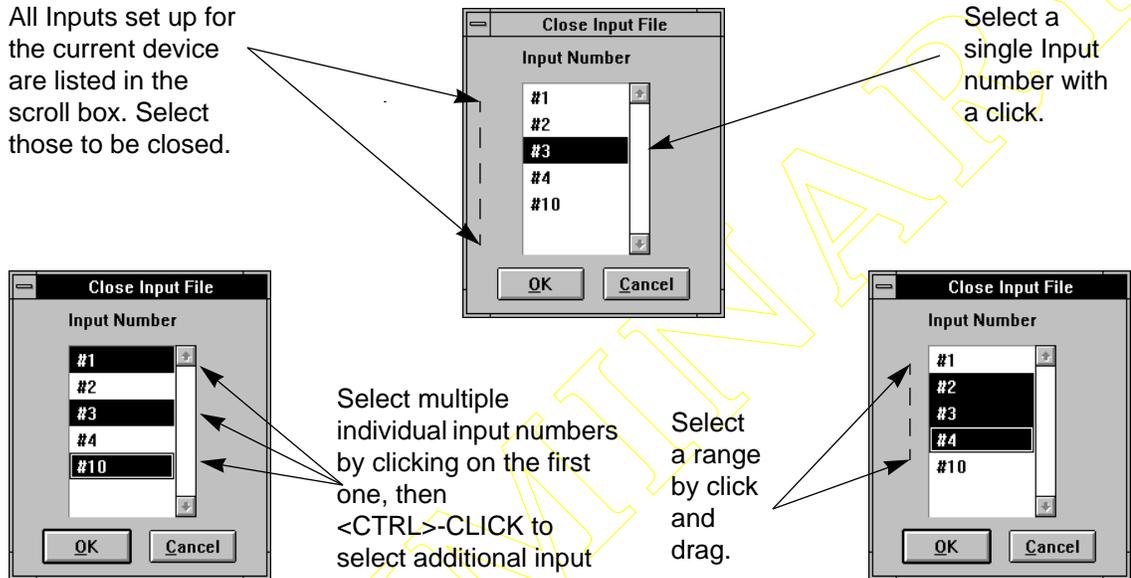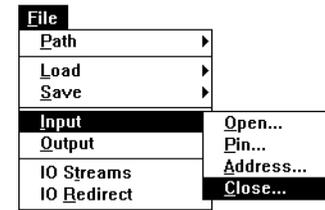
Figure 4-10 File//Input//Close Dialog Box

4.6.7   File//Output/Open

File//Output//Open prepares for writing data from a memory
location or block in the default device to the terminal or a file to
provide simulated output. The output file is associated with a
DEBUG instruction in the DSP code. More than one output file may be associated with a
single DEBUG instruction. When the DEBUG is executed, registers must be set as shown
in **Table 4-2**.
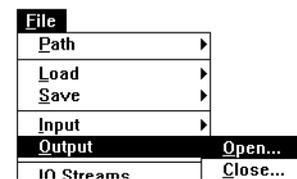
Table 4-2 Register Requirements for Simulated Output

Data Item	DSP96xxx	DSP56xxx
Output File Number and byte count	R0	X0
Output Address	R1	R0
Memory map (P = 0, X = 1, Y = 2)	R2	R1

Each time the DEBUG instruction is executed, the data is obtained from the output address, and written to the output file associated with the file number specified. The format of the data file and programming requirements is documented in the command line Output **Section 3.4.30**.

Set Output File No. First available no. is offered.

Enter address of DEBUG instruction written into code.

Select radix to be used for data output.

Select data to file or session window.

Enter file name for file output. Default extension is '.IO'.

Click to open file chooser.
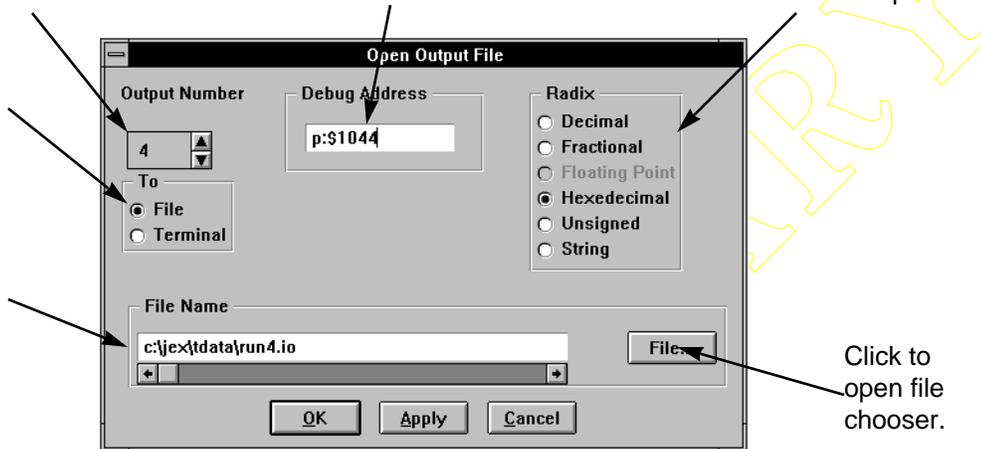
Figure 4-11 File//Output//Open Dialog Box

4.6.8   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>-Clicks, and Click-and-Drag. Then close all selected outputs by clicking [OK]. See File//Input//Close for close dialog box usage illustration.
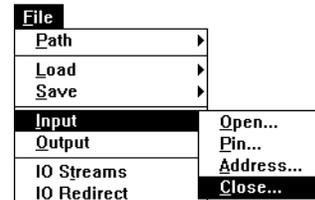
4.6.9   File//IO Streams//...

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.
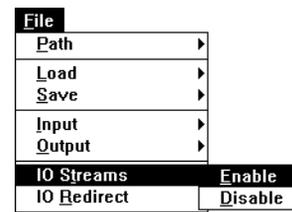
4.6.10　File//IO Redirect//...

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.

Select stream to redirect.

... Click to open File Chooser.

Enter file name manually or...

Select stream(s) to close. Only redirected streams are offered. Then click [OK] to close.
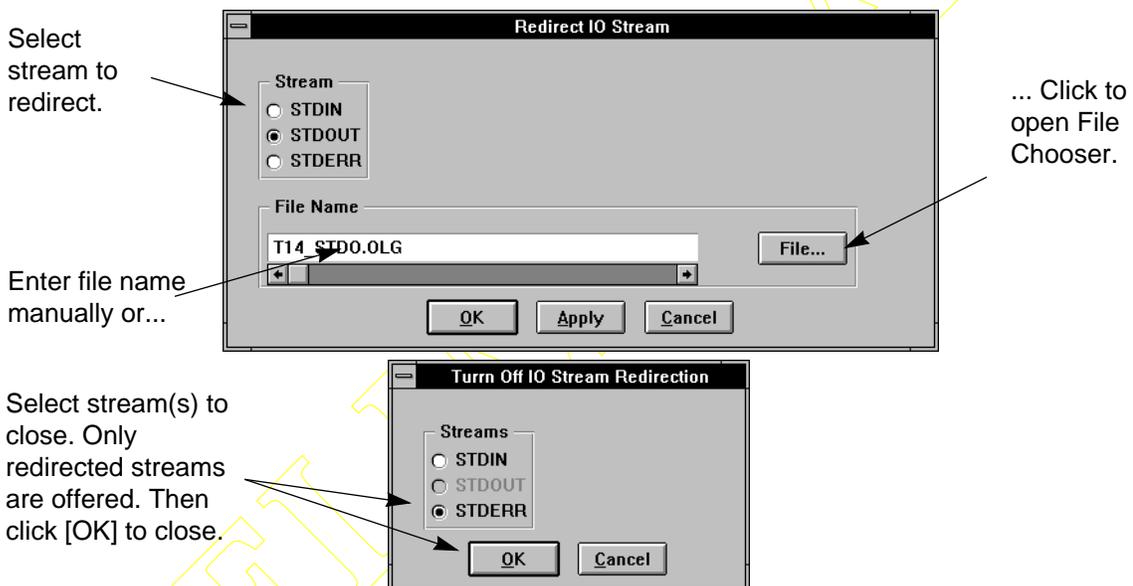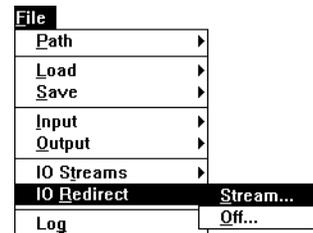
Figure 4-12　File//IO Redirect//... Dialog Boxes

4.6.11　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
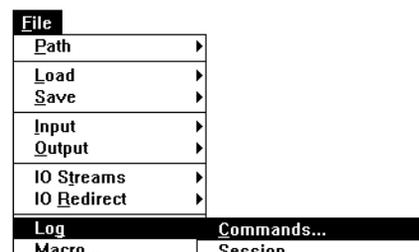
used in the GUI as a macro file (see File//Macro), when all the commands recorded in the log file will be executed. This file is a standard ASCII text file, and may be modified with any text editor as desired.

Note: 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.

Enter file name manually if desired. Command log files use extension '.SIM'. Use wildcards to specify which files are shown in file list.

Select directories from list by double-click to build path.

Double-click on required file or Single-click and [OK] to open log. Another dialog box will open to confirm if existing file is to be replaced.

Select drive from pull-down list.

Select desired file type from pulldown list to specify which files are displayed in list.
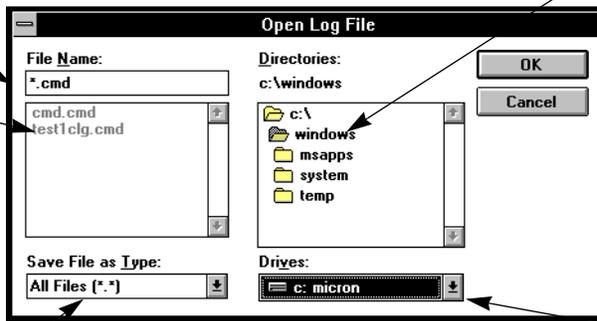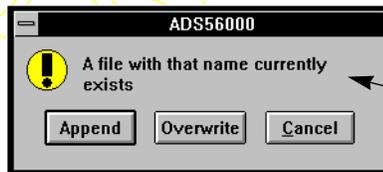
Specify action to be taken if file selected or entered already exists.

Figure 4-13 File//Log//Commands Dialog Box

4.6.12   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.

Enter file name manually if desired. Session log files use extension '.Log'. Use wildcards to specify which files are shown in file list.

Double-click on required file or Single-click and [OK] to open log. Another dialog box will open to confirm action if file already exists.

Select desired file type from pulldown list to specify which files are displayed in the list.

Select directories from the list by double-clicking to build a path.

Select drive from pull-down list.

Specify action to be taken if file selected or entered already exists.
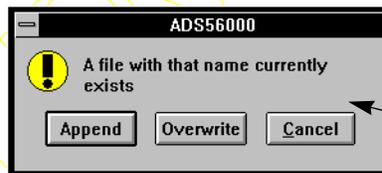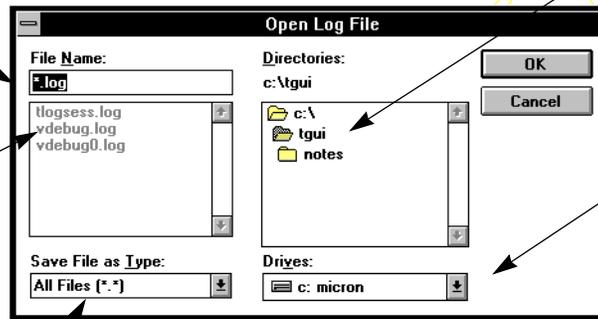
Figure 4-14 File//Log//Commands Dialog Box

4.6.13 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.

Check box is
drawn shaded if
log is not active.

Note that only log
files for the current
device will be
closed.

Click check boxes
to select log activity
to be closed.

Log activity not
checked will
remain active.
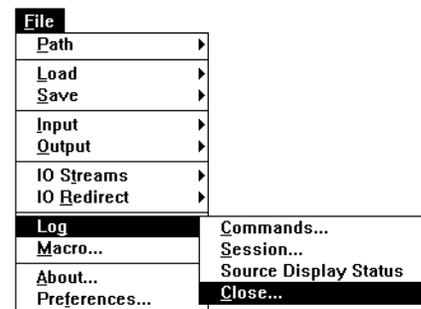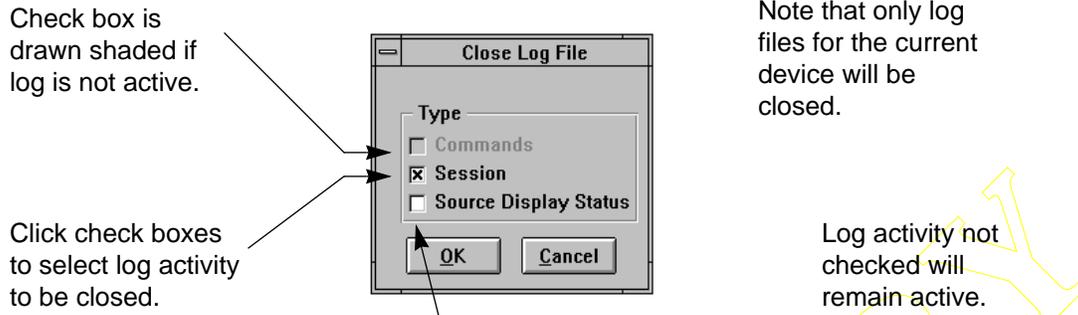
Figure 4-**15** File//Log//Close Dialog Box

4.6.14   File//Macro

File//Macro reads and executes a file containing commands for the
GUI. These commands are documented in the User 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.
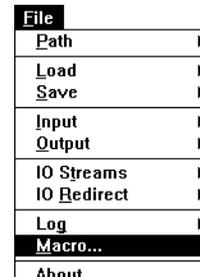
As the commands are read from the Macro file, they are displayed in the Command
window, executed, and echoed in the Session window, along with any output generated.
Commands which affect an individual device will execute on the current device, unless
the command specifies a particular device. Thus, a 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.

4.6.15   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.

Click to dismiss.
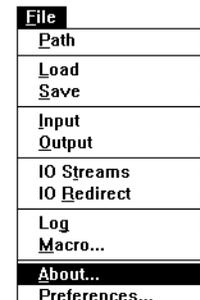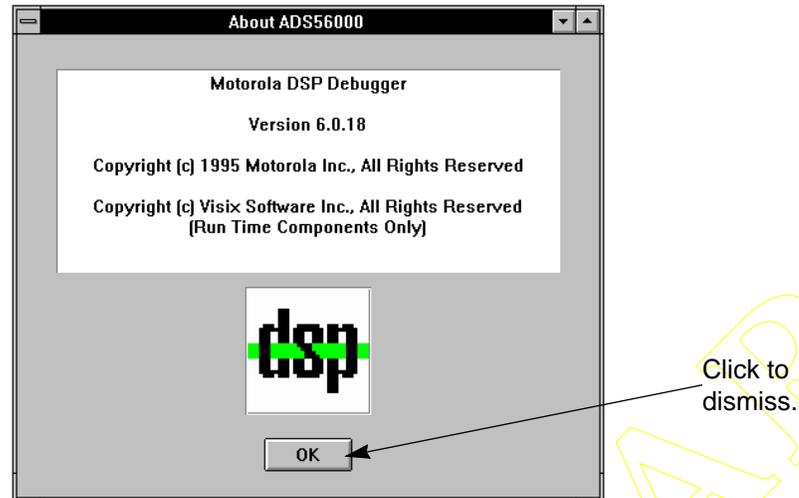
Figure 4-**16** File//About Dialog Box

4.6.16   File//Preferences

The Preferences dialog box provides the option to save the window
positions on exit. Thus when restarting the GUI, 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.
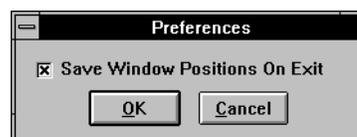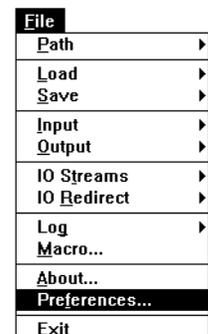
Figure 4-**17** File//Preferences Dialog Box

4.6.17 File//Exit

This option exits 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.
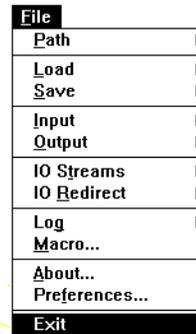
Figure 4-18 File//Exit Dialog Box

4.7   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: 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 include:

- Display selected registers & variables
- View memory as 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 / 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

```
Display
Display                        ▶
Disassemble                    ▶
List...
Evaluate...
Call Stack...
Radix
Device
Path
Input Files
Output Files
Redirected IO Streams
IO Streams Status
Log Files
Breakpoints
Watch                          ▶
Type..

More                           ▶
View                           ▶
```
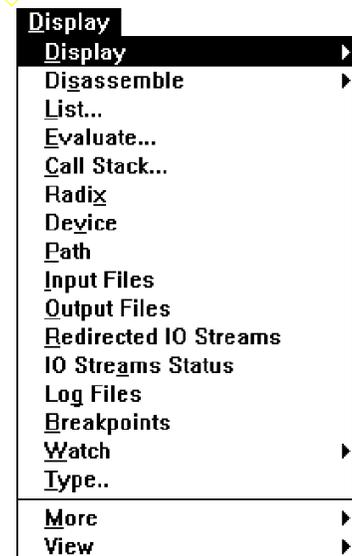
4.7.1 Display//Display//Active
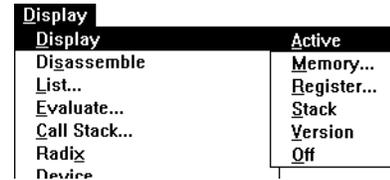
Selecting this window writes the enabled registers and memory locations to the Session window. The initial setting is all registers and no memory displayed. This is the same display as presented in the Session window whenever program execution stops. See Display//Display//Memory, Registers, and Watch.

Note: 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.

Command to enable memory display

Register display. Modify//Radix//Display sets output radix

Memory display requested in

Next instruction to execute
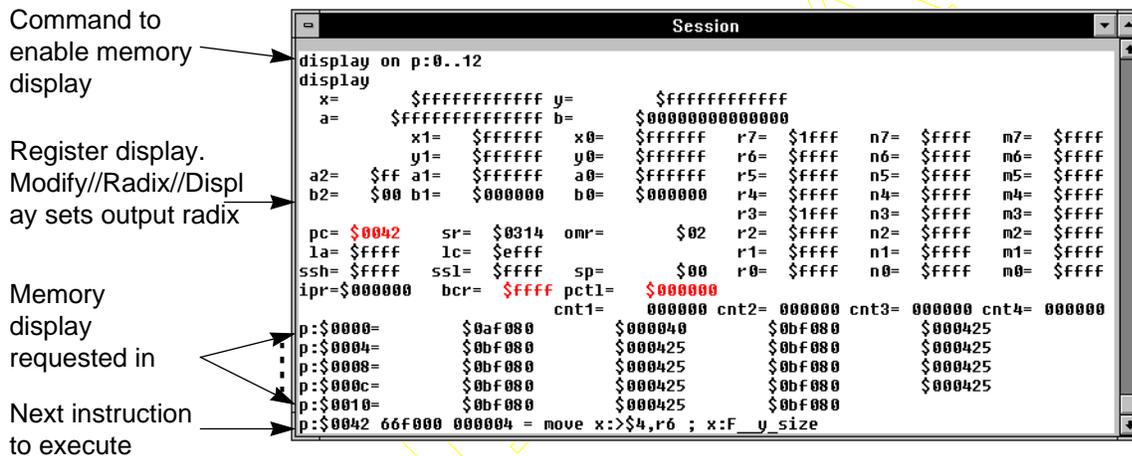
Figure 4-19 Display//Display//Active Output

4.7.2 Display//Display//Memory
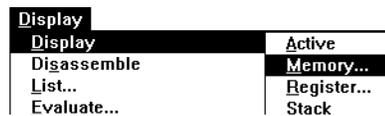
This menu selection controls the display of memory areas either immediately or as part of the post-execution display. Post-execution display may be unconditional or conditional on the way in which memory has been accessed during execution.

Select display mode:

On—Always display after execution.

Off—Do not display.

Immediate—Display now.
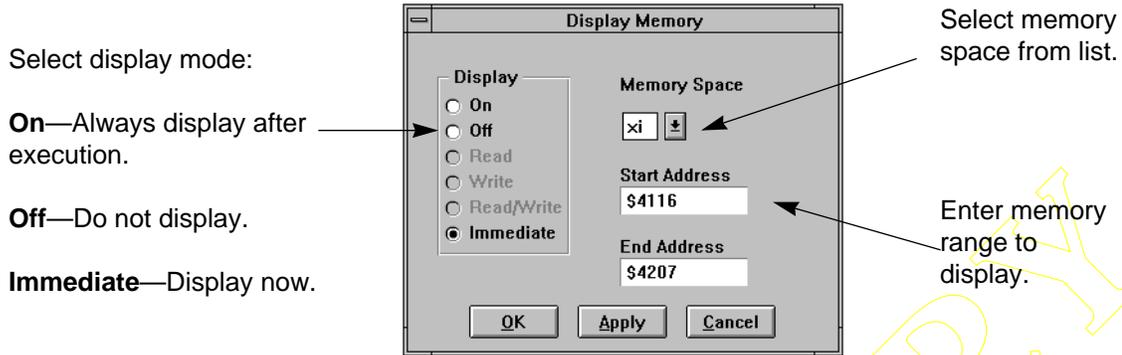
Figure 4-**20** Display//Display//Memory Dialog Box

4.7.3   Display//Display//Registers

The menu selection controls the display of registers either immediately or as part of the post-execution display in Session window.

Select display mode:

On—Always display after execution.

Off—Do not display.

Immediate—Display now.

Select registers with CLICK, CLICK/DRAG, CTRL-CLICK.
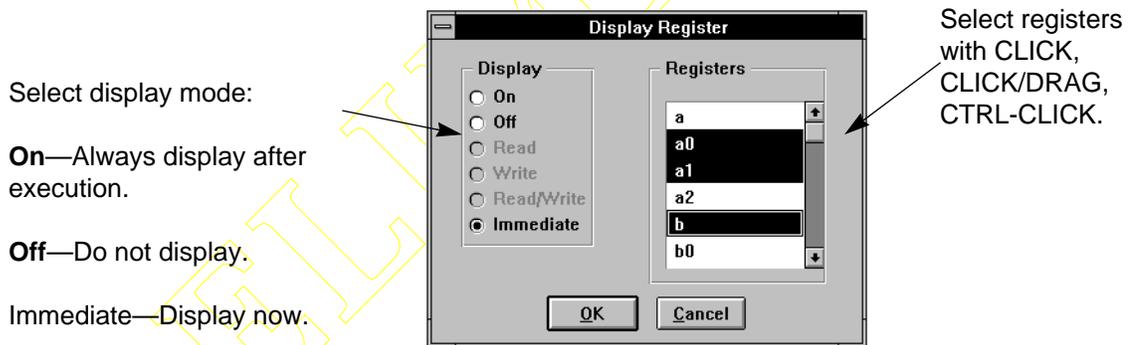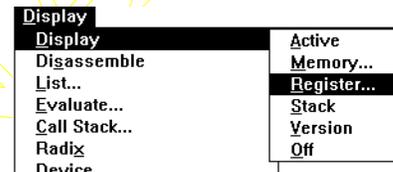
Figure 4-**21** Display//Display//Registers Dialog Box

4.7.4   Display//Display//Stack

This menu selection sends the stack contents to the Session window. The entire stack is displayed 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.
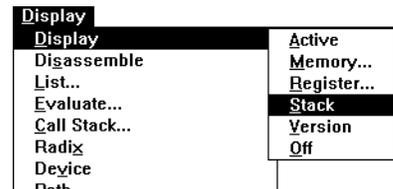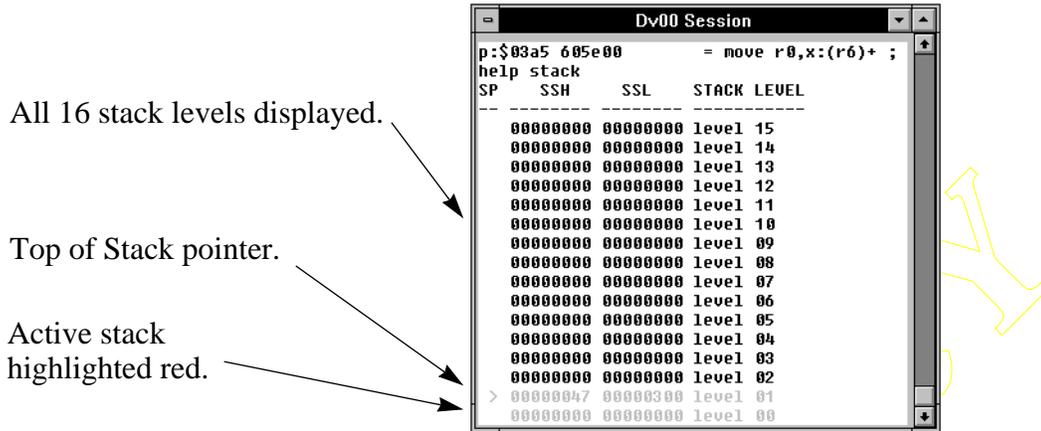
Figure 4-22 Display//Display//Stack Output

4.7.5   Display//Display//Version

This menu selection displays the command converter revision number and the ADS version number and production date.
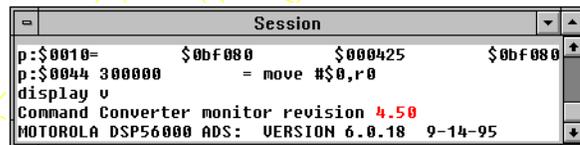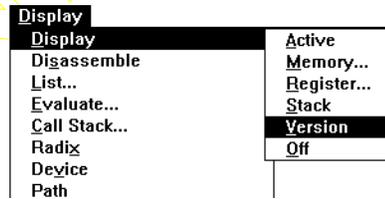
Figure 4-23 Display//Display//Version Output

4.7.6   Display//Display//Off

This selection cancels all memory and register display at the end of execution.
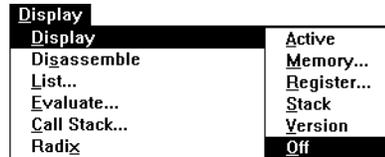
Note: STEP executed, but no register or memory display.
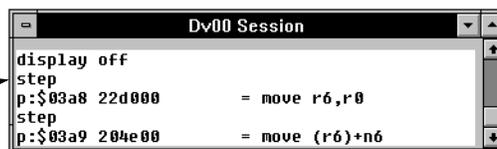
Figure 4-24 Display//Display//Off Output

4.7.7 Display//Disassemble//From PC, 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.
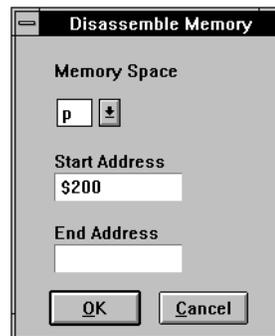
Figure 4-25 Display//Disassemble//Memory Dialog Box

- 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 4-26 Display//Disassemble//... Output

4.7.8   Display//List

This selection 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.
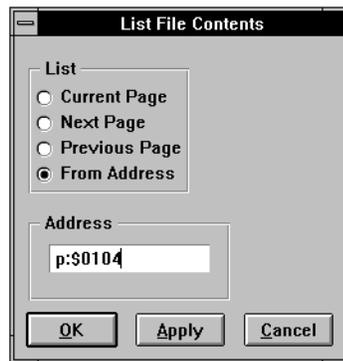
Figure 4-27 Display//List File Dialog Box
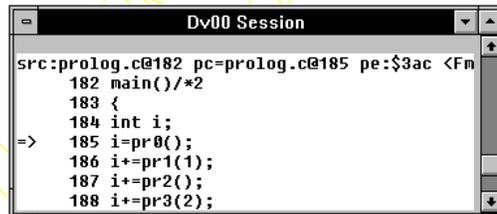
Figure 4-28 Display//List File Output

4.7.9 Display//Evaluate

This selection evaluates DSP Assembler and C expressions and writes 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.

Enter expression to evaluate. Enclose C expression in {}.

Select radix for display.
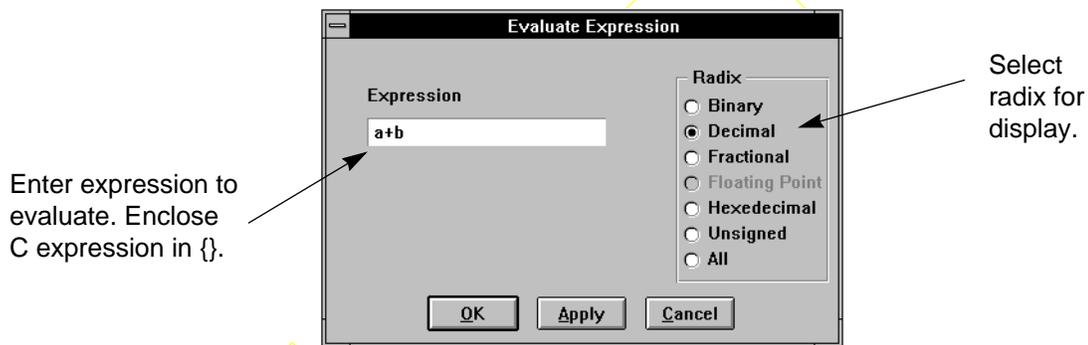
Figure 4-29 Display//Evaluate Dialog Box

Expression is echoed, evaluated, and the result displayed.

C expressions are 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.
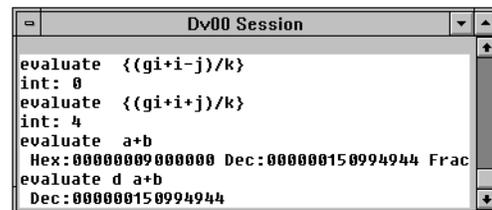
Figure 4-30 Display//Evaluate Output

4.7.10   Display//Call Stack

This selection 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.

Innermost—Start at current procedure and work back toward main().

Outermost—Start at main() and work toward current procedure.

Dialog opens with no. of call frames on stack. Reduce if desired. Increasing gives error message.
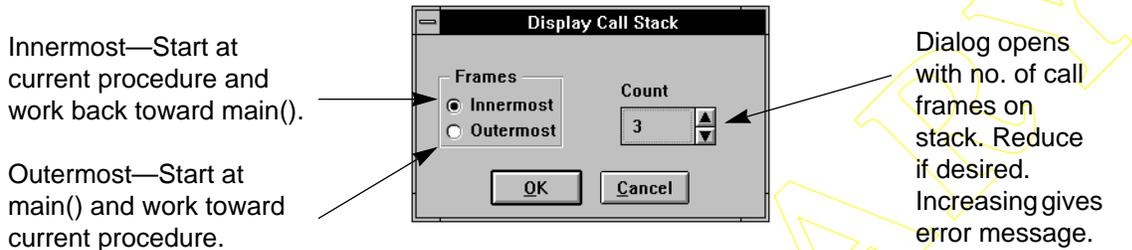
Figure 4-31 Display//Call Stack Dialog Box

Frames are listed in order selected—from inner or outer end (here, inner).

```
p:$00c5 45f000 00001e = move x:>$1e,x1 ; x:Fgi
where +3
==> #0  p:0xc5 in pr5 (k=4)
    #1  p:0x122 in pr7 (k=4)
    #2  p:0x3d2 in main ()
```

Address of next instruction to execute.....   .....In procedure.....   .....which was called with these parameters.
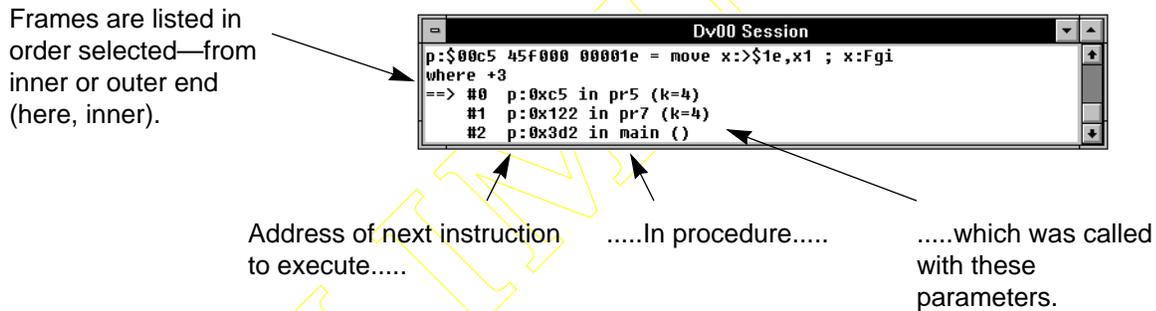
Figure 4-32 Display//Call Stack Output

4.7.11   Display//Radix

This selection 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 Hexadecimal.

```
p:$0122 205600      = move (r6)-
radix
The current default radix is decimal.
```
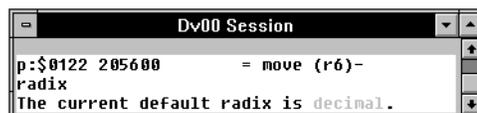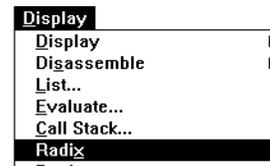
Figure 4-33 Display//Radix Output

4.7.12 Display//Device

This selection displays the status of each possible DSP device and lists the device types supported. The configuration of each device is set with Modify//Device//Configure.

```
 ┌──────────────────────────────────────┐
 │ ⊟            Session            ▼ │▲│
 │DEVICE STATUS TYPE     DEVICE STATUS TYPE │▲│
 │────── ────── ──────── ────── ────── ──────── │
 │                                      │
 │dv00   on     56002    dv04   unused  │
 │dv01   unused          dv05   unused  │
 │dv02   unused          dv06   unused  │
 │dv03   unused          dv07   unused  │
 │                                      │
 │--Possible device types--             │
 │56002                                 │
 │56004                                 │
 │56004rom                              │
 │56005                                 │
 │56007                                 │
 │56009                              │▼│
 └──────────────────────────────────────┘
```

Status of each possible device is listed.

Supported DSP family members are listed.
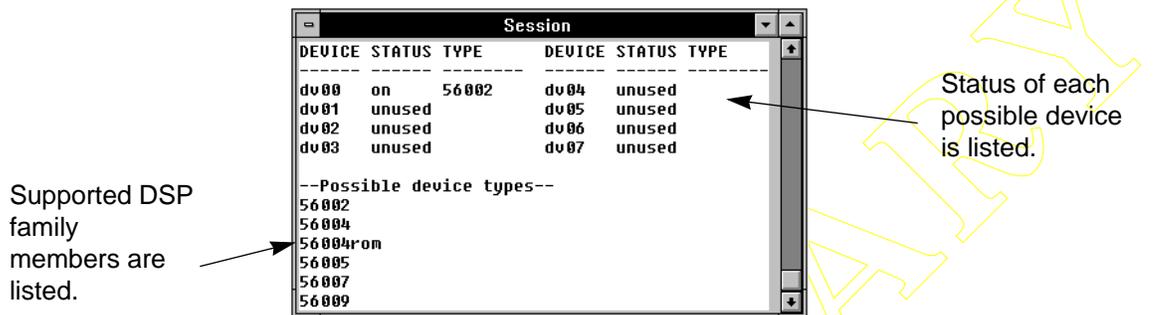
Figure 4-**34** Display//Device Output

4.7.13 Display//Path

This selection 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.

```
 ┌──────────────────────────────────────┐
 │ ⊟         Dv00 Session        ▼ │▲│
 │path                                  │▲│
 │Device Working Directory: C:\TGUI3    │
 │Alternate Source Paths:               │
 │C:\DSPCODE                            │
 │C:\PROJECTX                        │▼│
 └──────────────────────────────────────┘
```

Figure 4-**35** Display//Path Output

4.7.14 Display//Input Files and Display//Output Files

This selection displays the file assignments for simulated input and output for the current device. See File//Input//... and File//Output//... for assignment procedures.

```
┌──────────────────── Session ──────────────────────┐
│ input #5 off                                        │
│ winput                                              │
│ input                                               │
│ #1. Input file: C:\TGUI3\term0005.io at address p:$4117 │
│ #2. Input file: C:\TGUI3\TRAGS.IO at address p:$1046 │
│ #3. Input file: C:\TGUI3\T4R7.IO at address p:$2446 │
│ #4. Input file: C:\DSPCODE\OBJECT\TERM0000.IO at address p:$3e8 │
└─────────────────────────────────────────────────────┘
```
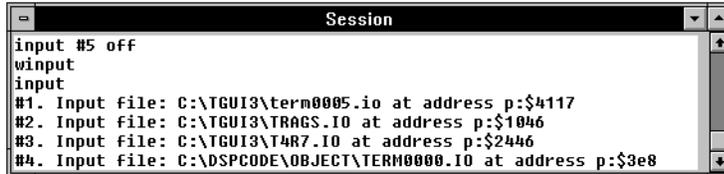
Figure 4-**36** Display//Input Files Output

4.7.15 Display//Redirected IO Streams, IO Streams Status

IO stream redirection supports stream IO for C programs running on a DSP. 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.

Use of
File//IO Redirect to
redirect STDOUT

Stream Status

Redirected IO Streams

```
┌──────────────── Dv00 Session ──────────────────┐
│ redirect stdout d0p1r4.so                        │
│ streams                                          │
│ streams enabled                                  │
│ redirect                                         │
│ stdin off                                        │
│ stdout to C:\TGUI3\d0p1r4.so                     │
│ stderr off                                       │
└──────────────────────────────────────────────────┘
```
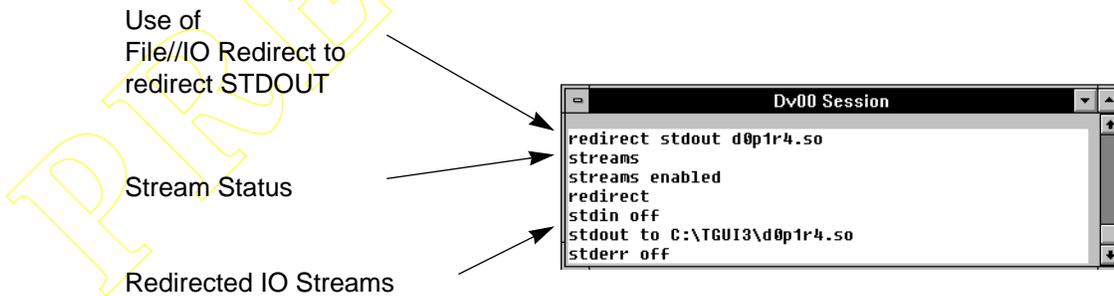
Figure 4-**37** Display//IO Streams Output

4.7.16   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.
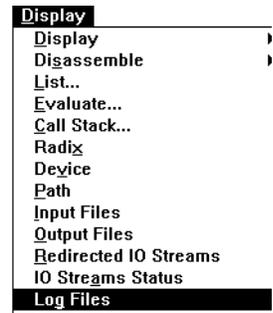
From "File//Log//..." to open the log files

From "Display//Log" to show open log files.
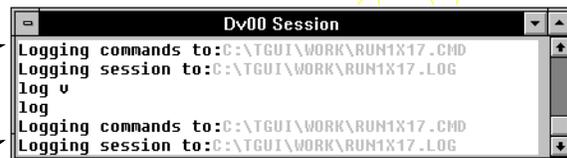
Note: Any log file not listed is closed.

Figure 4-**38** Display//Log Files Output

4.7.17   Display//Breakpoints

This selection 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.
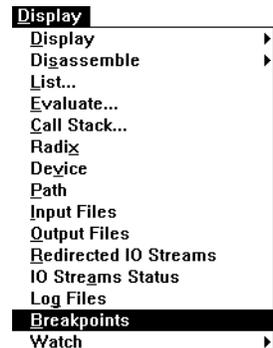
Breakpoints are set by clicking source and Assembler windows (SW).

Note at address$3b6 if carry set (SW).

Break on RW access to addresses specified.
Do action only if x0 is greater than 6 (HW).
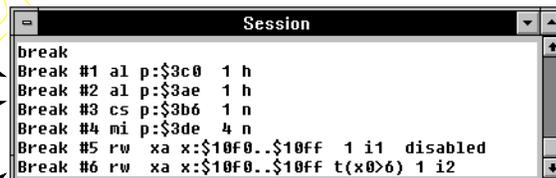
Figure 4-**39** Display//Breakpoints Output

4.7.18   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.
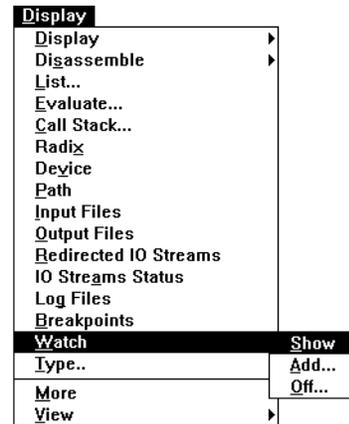
List expressions with reference numbers.

Value of expression is output. Note when 'i' goes out of scope (masked by another 'i'), the expression cannot be evaluated.

Figure 4-**40** Display//Watch//Show Output

4.7.19   Display//Watch//Add

Display//Watch//Add adds expressions to the watch list. Symbolic references are interpreted as Assembler mnemonics and register names, unless the expression is 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

Enter expression for watch list.
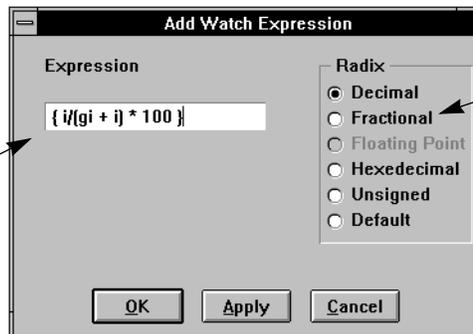
Select radix for display.

Figure 4-**41** Display//Watch//Add Dialog Box

4.7.20   Display//Watch//Off

This selection 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.

Select watch expressions to remove. Use Click, Ctrl-Click and Click/Drag.
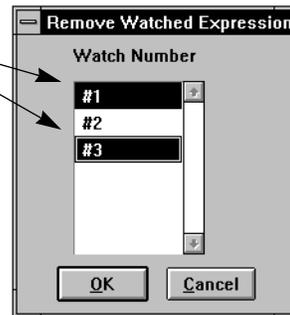
Figure 4-**42** Display//Watch//Off Dialog Box

4.7.21   Display//Type

This selection displays the type of a C variable or expression. Use Modify//Up or //Down to select the evaluation context.
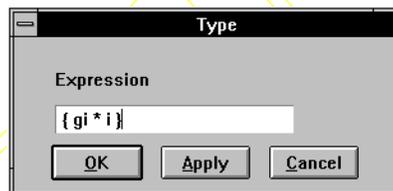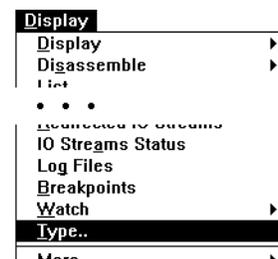
Figure 4-**43** Display//Type Dialog Box
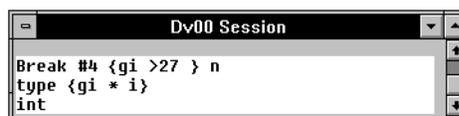
Figure 4-**44** Display//Type Output

4.7.22 Display//More

Display//More freezes the Session window when it is full until the user responds. It is useful when the output from an operation may be longer than the Session buffer.
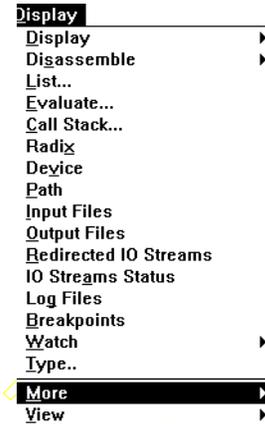
Figure 4-45 Display//More Dialog Box

4.7.23 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.
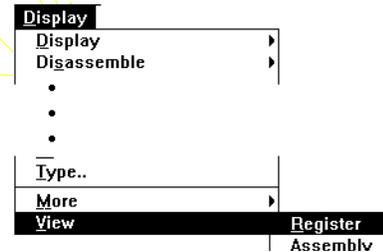
Register View shows commands entered for device and output.

At break, all enabled registers are output. No memory enabled here. Changed values are in red.

Break instruction is displayed.

```
 la= $0000    lc=  $0000              r1=  $0000    n1=  $0000    m1=  $FFFF
ssh= $0000   ssl= $0000    sp=    $00  r0=  $00c0    n0=  $0000    m0=  $FFFF
ipr= $0000   bcr=  $FFFF
cyc=000175   ictr= 000013 cnt1=    000000 cnt2= 000000 cnt3= 000000 cnt4= 000000
p:$03ac 0bf080 000050 = jsr >$50 ; p:Fpr0
path C:\DSPCODE\OBJECT
path + C:\DSPCODE\SOURCE
load  prolog
Loading file:C:\DSPCODE\OBJECT\prolog.cld
wasm
break p:$00cd
go
Break #1 p:$00cd h ;dev:0 pc:00cd cyc:1091
   x=      $000003000000 y=      $0000000000000
   a=      $00000004000000 b=    $00000004000000
           x1=  $000003    x0=  $000000    r7=  $0000    n7=  $0000    m7=  $FFFF
           y1=  $000000    y0=  $000000    r6=  $00ca    n6=  $0001    m6=  $FFFF
  a2=  $00 a1=  $000004    a0=  $000000    r5=  $0000    n5=  $0000    m5=  $FFFF
  b2=  $00 b1=  $000004    b0=  $000000    r4=  $0000    n4=  $0000    m4=  $FFFF
                                          r3=  $0000    n3=  $0000    m3=  $FFFF
  pc= $00ce    sr=  $0310    omr=    $02   r2=  $0000    n2=  $0000    m2=  $FFFF
  la= $0000    lc=  $0000                 r1=  $0000    n1=  $0000    m1=  $FFFF
ssh= $0000   ssl= $0000    sp=    $00    r0=  $00c5    n0=  $FFFd    m0=  $FFFF
ipr= $0000   bcr=  $FFFF
cyc=001091   ictr= 000186 cnt1=    000000 cnt2= 000000 cnt3= 000000 cnt4= 000000
p:$00cd 44f400 000002 = move #>$2,x0
```

Figure 4-46 Session Window—Register View

4.7.24 Display//View//Assembly, 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.
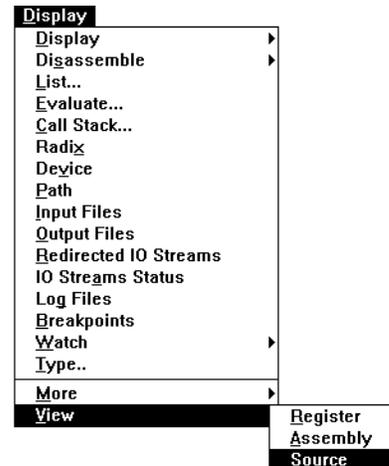
```
─                          Dv00 Session                        ▼ ▲
asm:pi:$9b pc=prolog.c@29 pi:$9e <Fpr4+4> section:prolog_c       ▲
   009b       Fpr4+0001 move ssh,x:(r6)+ ; x:(F__c_sig_handlers+136)
   009c       Fpr4+0002 move #$1,n6
   009d       Fpr4+0003 move r6,r0
=>009e       Fpr4+0004 move (r6)+n6
   009f       Fpr4+0005 move b0,x:(r6)+ ; x:(F__c_sig_handlers+136)
   00a0       Fpr4+0006 move b1,x:(r6)+ ; x:(F__c_sig_handlers+136)
   00a1       Fpr4+0007 move x0,x:(r6)+ ; x:(F__c_sig_handlers+136) ▼
```

Figure 4-47 Session Window, Assembly View

4.8 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 hexadecimal. 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 56002).

- Select a stack frame from the C call stack as the context for C expression evaluation.
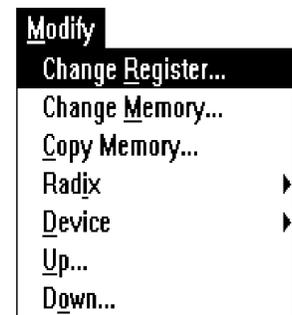
4.8.1 Modify//Change Register

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.
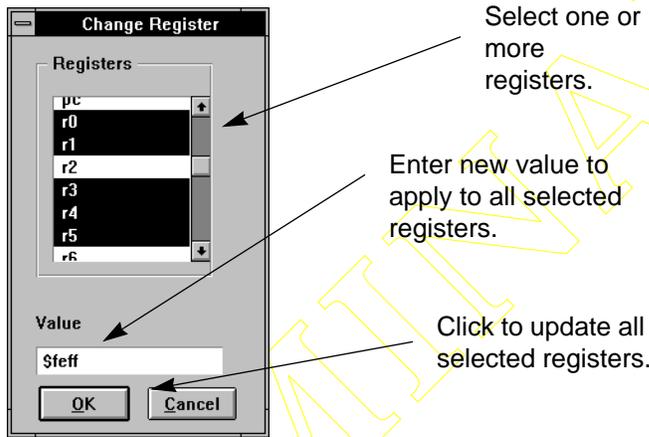
Figure 4-48 Modify//Change Register Dialog Box

4.8.2 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: Addresses are frequently specified in hexadecimal. Use the '$' radix specifier
for hexadecimal, or set the default radix to hexadecimal
(Modify//Radix//Set Default).

Select memory
space from
pull-down list.

Enter new value
for entire
address range
specified.

Enter start and
end of address
range.
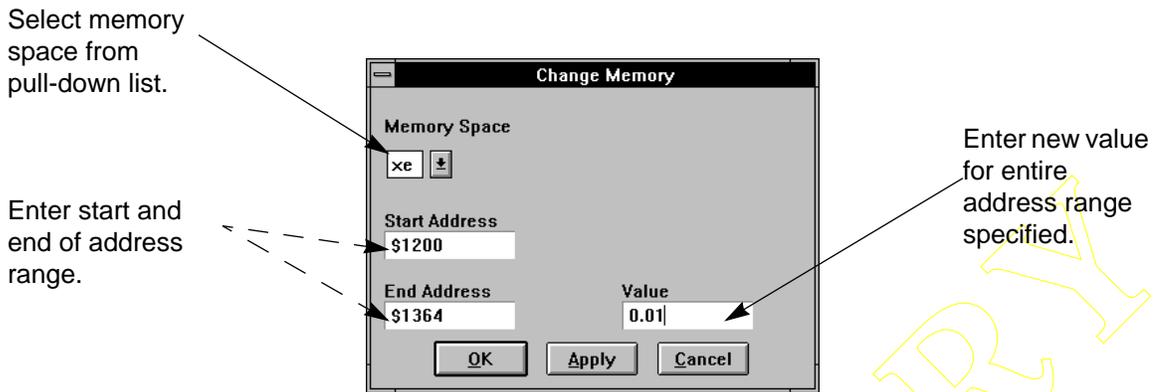
Figure 4-**49** Modify∕∕Change Memory Dialog Box

4.8.3   **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.

Select source
memory space
from pull-down
list.

Select
destination
address space
and enter starting
address.

Enter source
start and end
addresses.
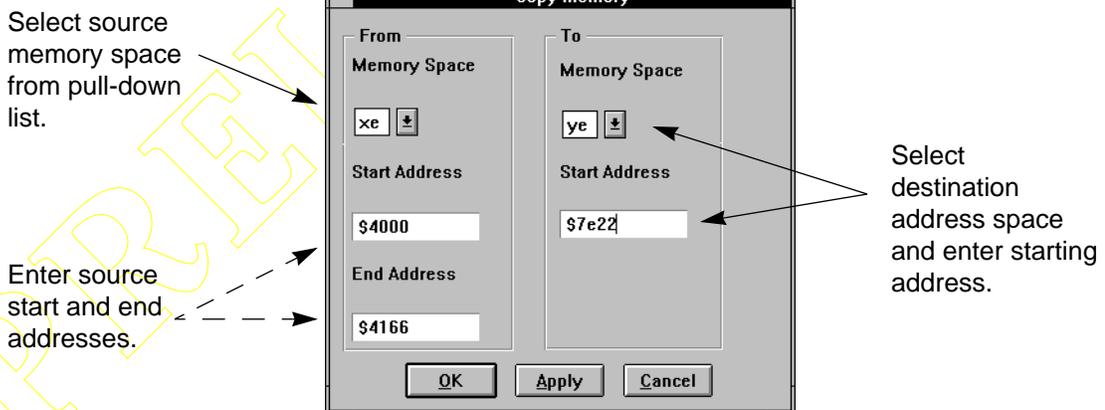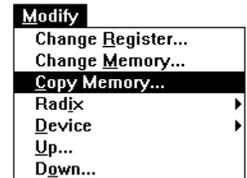
Figure 4-**50** Modify∕∕Copy Memory Dialog Box

4.8.4 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. The initial default radix is Hexadecimal.
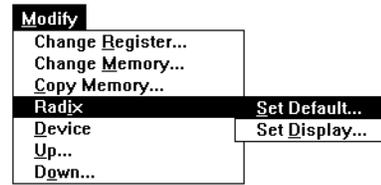
Note: The Radix Operator is used as a prefix to the input value.

Click on radix to be used for all input values—numeric and address.
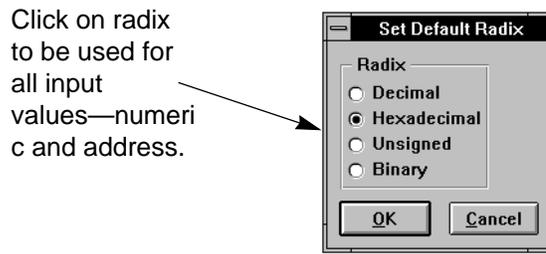
Figure 4-51 Modify//Radix//Set Default Dialog Box

4.8.5 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.

Click on the radix to be applied to all selected locations.

Select one or more registers from scrolling list if required.

Select memory space from pull-down list.

Enter start address to set radix for one word.

Enter end address to apply radix to address range.

Note that the radix may be applied to a selection of registers or a block of memory or both at once.
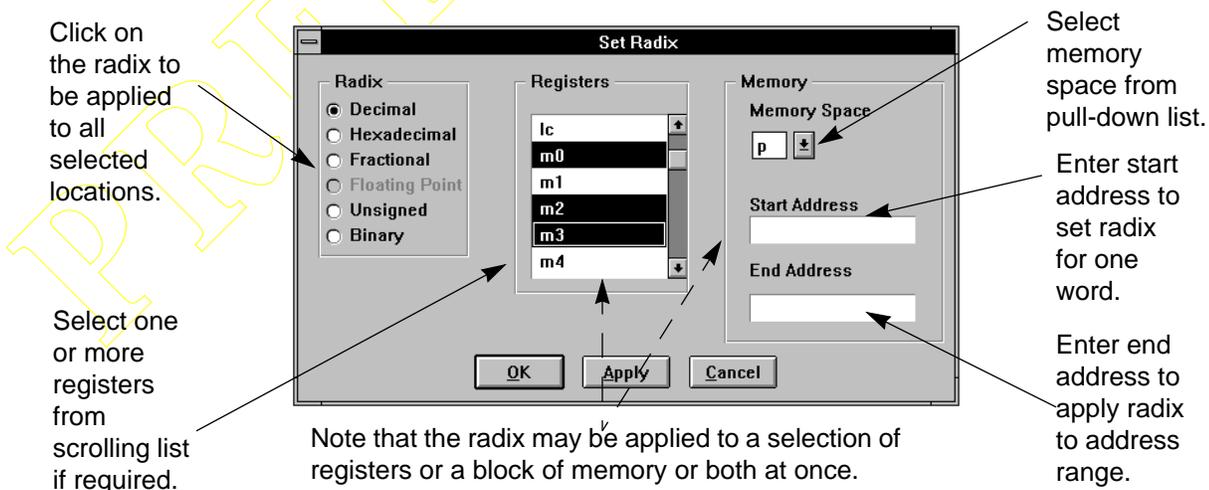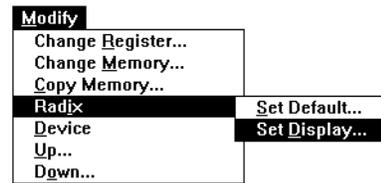
Figure 4-52 Modify//Radix//Set Display Dialog Box

4.8.6   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.
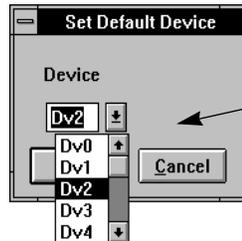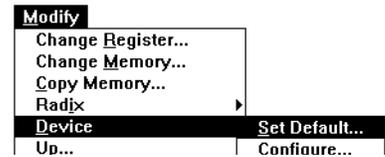
Select a device
from the
pull-down list and
click [OK].

Figure 4-**53** Modify//Device//Set Default Dialog Box

4.8.7   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:
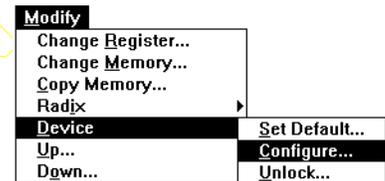
- **Type**—This selection specifies which particular member of the DSP family is in
  use. Type automatically adds a device to the system, initially Device 0.

- **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.

Select device
to configure.
Default is
current default
device.

Select type of
configuration to
be performed.

Select DSP
type from list
when Type
selected above.
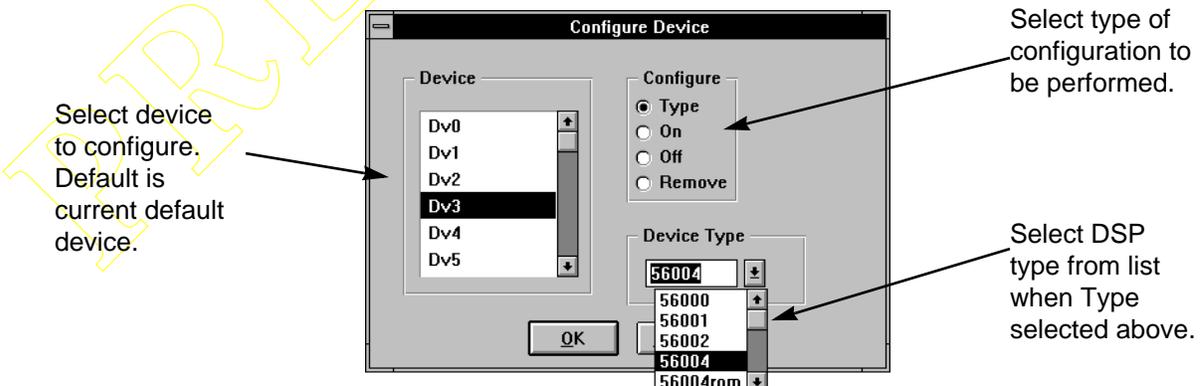
Figure 4-**54** Modify//Device//Configure Dialog Box

4.8.8 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.

Enter password, click [OK]. If valid, the device type appears in selection lists.

Enter device type to unlock.
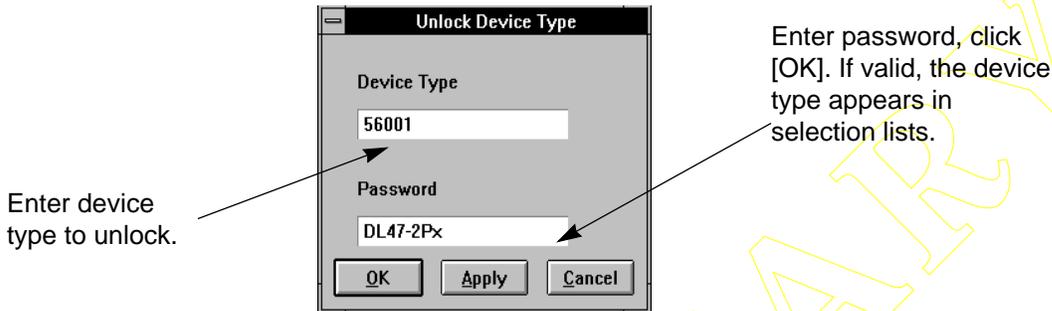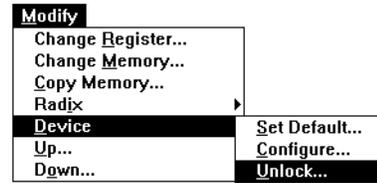
Figure 4-55 Modify//Device//Unlock Dialog Box

4.8.9 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, '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 7. Display//Evaluate returns the value which would be returned at the current execution point. If the expression refers to a variable in a calling function, which is masked by an identical variable in the current function, the required variable is inaccessible. To evaluate the expression, we need to be able to select the required stack frame as 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 can not be evaluated because it is 'out of scope', select the original context to evaluate the expression again.

Down decreases the call frame number towards the current function (frame 0).

Up increases the call frame number towards the main program.

Figure 4-56 Modify//Up Dialog Box

4.9   EXECUTE MENU

The Execute menu controls the execution of programs on the target device:

Execute
Go...
Step...
Trace...
Next...
Finish
Until...

Breakpoints   ►

Wait...
Stop
Reset   ►

*   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 subroutine call is executed, Step follows the execution through the subroutine.

*   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 subroutine calls.

*   Finish executes to the end of the current subroutine, 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 ADS state.
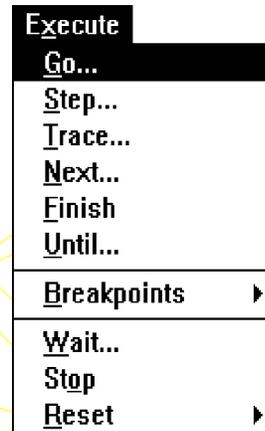
4.9.1 Execute//Go

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), 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).
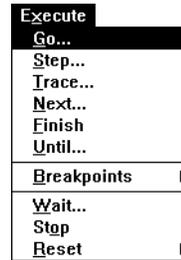
Select from:
1) Proceed from next address OR specified address.
2) Reset device before proceeding.

Select which device(s) execute.

If the address is selected above, may enter start address here. If blank, proceed from next address.
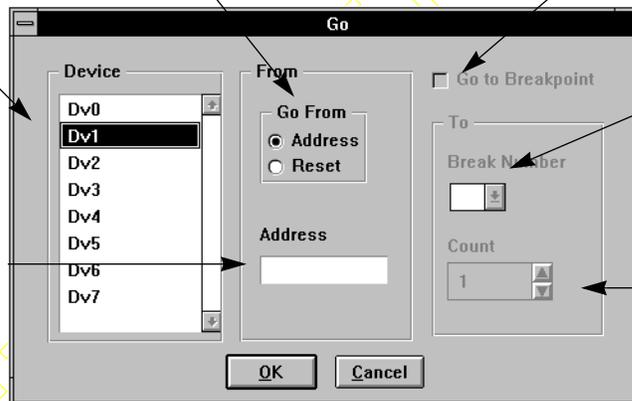
IF breakpoints have been established, may select a target breakpoint. If selected, all other STOP breakpoints will be ignored.

Select target breakpoint from pull-down list.

Specify how many times to encounter breakpoint before stopping (i.e. stop on 4th time breakpoint is encountered).

Figure 4-**57** Execute//Go Dialog Box

4.9.2 Execute//Step, Next, Trace

Execute//Step executes a specified number of instructions or lines of code. If a function is called, it is treated like the rest of the code. Execute//Next treats a function call as a single steps. Execute//Trace outputs all enabled registers and memory locations after each instruction 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 4-**58** Execute//STEP Dialog Box

4.9.3 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.
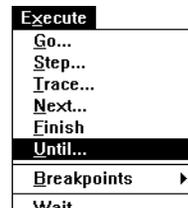
Line numbers and labels may only be used if debug information has been loaded from a COFF file (see File//Load//Memory COFF).
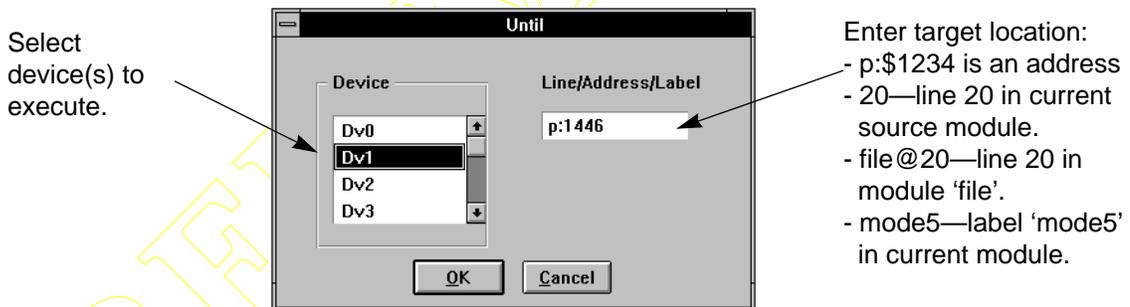
Figure 4-**59** Execute//Until Dialog Box

4.9.4 Execute//Finish

Program executes until the end of the current subroutine. 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.
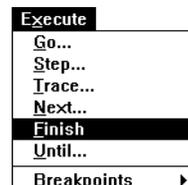
4.9.5   Execute//Breakpoints//Set Software

Set software 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 details on software breakpoint
types, see BREAK command, **Section 3.4.2**.

Figure 4-**60** Execute//Breakpoint//Set Software Dialog Box

4.9.6   Software Break Processing

Before the target is placed in User mode for program execution, the instruction at the
breakpoint address is replaced with a DEBUG opcode. If a conditional breakpoint is
specified, a conditional instruction is used, such as DEBUGLE. When the DEBUGcc

instruction is executed, the target enters Debug mode. The ADS program, which has been polling the command converter, detects the return of the target to Debug mode. The expression is evaluated; if false the target DSP is restarted, if true count is checked, and if the breakpoint has been encountered the correct number of times (ignoring occasions when the expression was false), the specified action is taken. The count is then re-initialized.

4.9.7   Execute//Breakpoints//Set Hardware

Set hardware 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. For details on hardware breakpoint types, see BREAK command, **Section 3.4.2**.

Specify which time break is to be acknowledged.

Set breakpoint number. Initially set to first free number.

Specify break type. OnCE registers are set to detect specified access.

Specify action to be taken when breakpoint is met. Options are:
- **Halt** execution.
- **Note**: Display breakpoint.

- **Show**: Display enabled registers & memory.
- **Increment**: specified counter.
- **Command**: Execute specified command on break.

Expression evaluated when break detected. If false, continue program, if true, continue break processing.
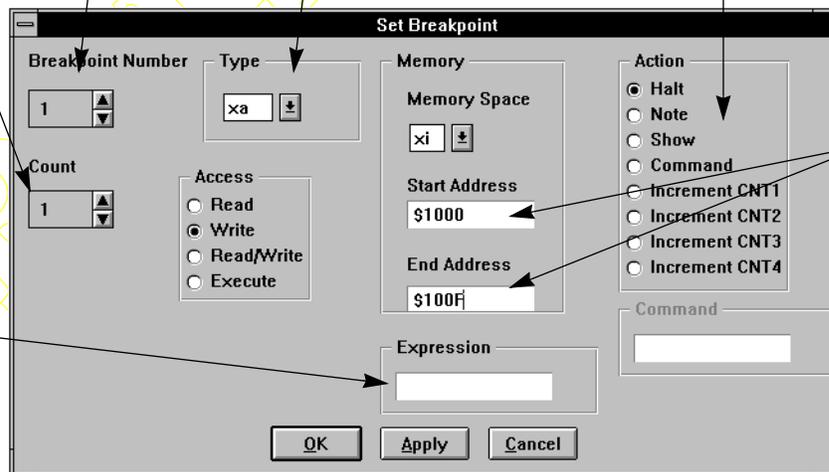
Specify address range for breakpoint.

Figure 4-61 Execute//Breakpoint//Set Hardware Dialog Box

4.9.8   DSP56300 and DSP56600 Breakpoint Logic

The DSP56300 and DSP56600 families feature breakpoint logic with twin comparators. These comparators each provide a True/False indication which may be combined in four ways to fully specify a breakpoint condition. These combinations are:

- AND—both conditions true causes breakpoint

- OR—either condition true causes breakpoint

- THEN—first condition true, and then second condition true causes breakpoint

- ONLY—first condition true causes breakpoint; second condition is not considered
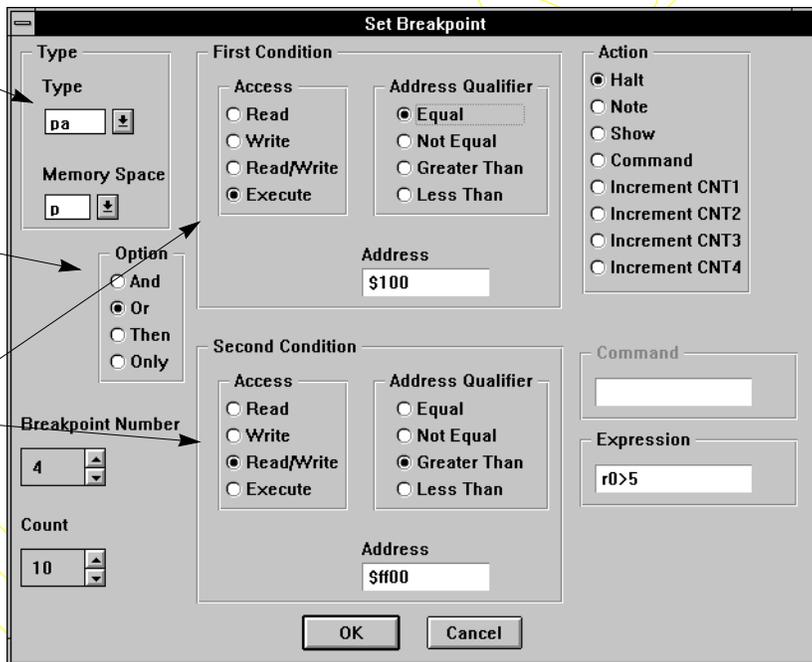
Figure 4-62 Execute//Breakpoint//Set Hardware Dialog Box (DSP56300, DSP56600)

4.9.9　Hardware Break Processing

During program execution, the OnCE logic constantly monitors address lines for specified values (e.g., program fetch from address P:$...). When the condition is true, the counter is decremented; when it reaches zero, the DSP enters Debug mode. The entry to Debug mode is detected by the Command Converter, which sends a service request to the development host. The ADS software then evaluates the expression (if specified), fetching register and memory values from the target DSP as necessary. If the expression is false, the DSP is returned to User mode and the program resumes execution. The user is not notified of this activity.

If the expression evaluates to true, or no expression was specified, the breakpoint has occurred. The specified action is then taken, such as returning control to the user (Halt action), or incrementing the specified counter (Increment Cnt1). If the action is Halt, the user has full control over subsequent activity. For all other actions, the target device is automatically returned to USER mode to continue program execution.

4.9.10　Execute//Breakpoints//Clear

This selection 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...
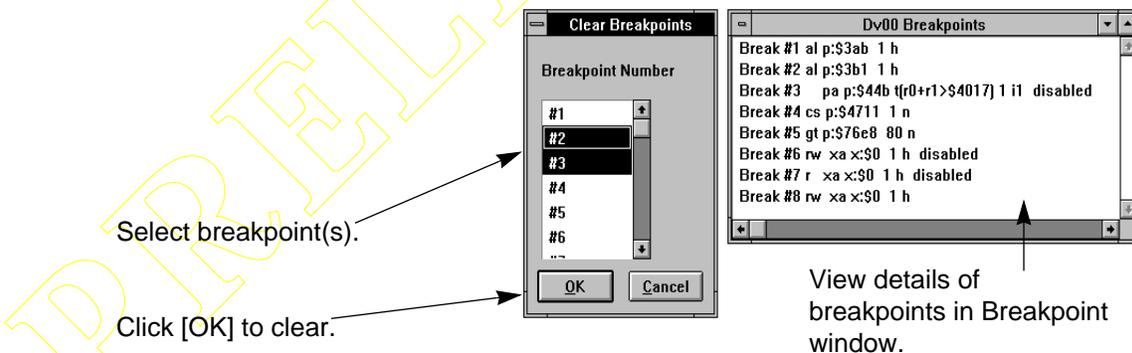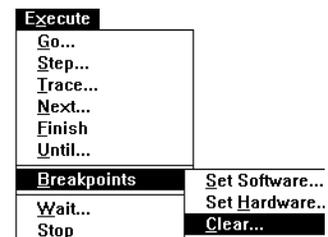
Figure 4-63 Execute//Breakpoint//Clear Dialog Box

4.9.11   Execute//Breakpoints//Enable, Disable

Breakpoints may be disabled and enabled from this menu.
Disable temporarily deactivates the selected breakpoints,
Enable reinstates them. While disabled, breakpoints have no
effect on DSP program execution, and do not cause any of the
actions associated with the breakpoint.

Only disabled
Breakpoints
are listed.
Select those to
enable.

Breakpoint
window also
shows
disabled
status of
breakpoints.

NOTE: Disable Breakpoint dialog box is identical
apart from the title bar, and is not shown.
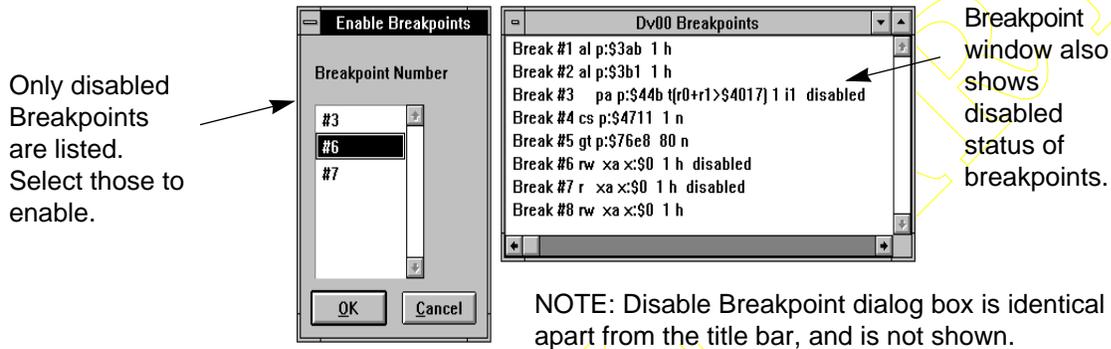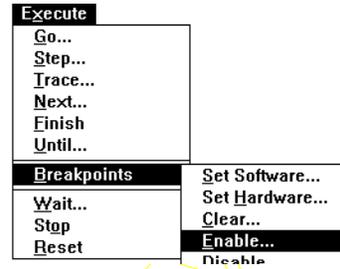
Figure 4-**64** Execute//Breakpoints//Enable Dialog Box

4.9.12   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.

Enter wait time
in seconds or
check 'Forever'.

Click to terminate wait.
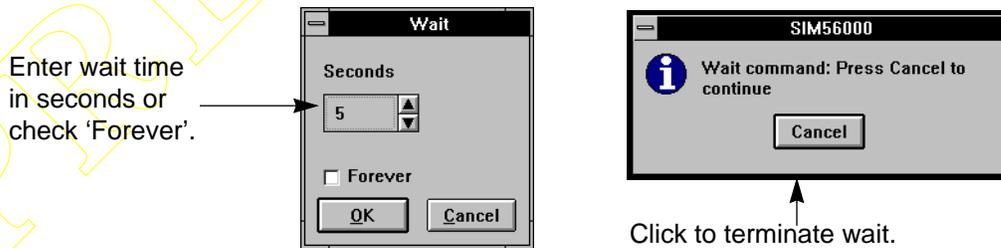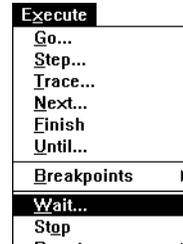
Figure 4-**65** Execute//Wait Dialog Box

4.9.13   Execute//Stop

Execute//Stop interrupts execution of the DSP program or macro execution. Control is returned to the user interface.

 Any temporary breakpoint set by Execute//Until is cleared.
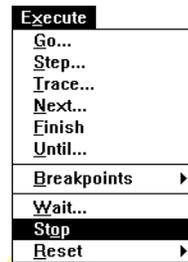
4.9.14   Execute//Reset...

Execute//Reset//Device performs hardware reset on the current device, asserting debug request, to leave the device in Debug mode. Some registers are initialized by a reset.

Execute//Reset//Command Converter resets the command converter.

Execute//Reset//System resets the command converter, then resets the device.
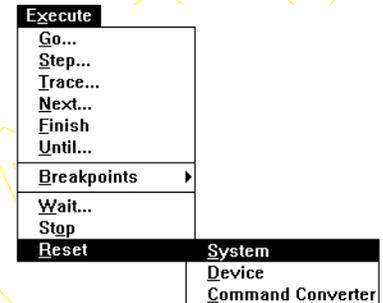
4.10   WINDOWS MENU

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 for each device to show different memory areas at the same time.
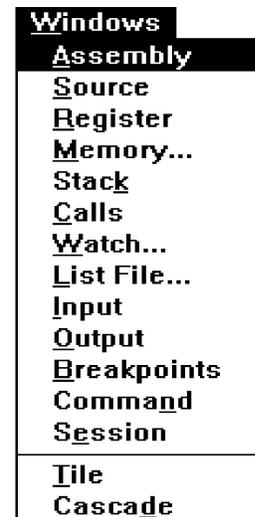
Table 4-3 Summary of Window Functions

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. There can be a maximum fifteen entries.	One per Device
Calls	Display **C** procedure call stack.	One per Device
Watch	Display expressions selected for Watching. Erase with double-click.	Multiple
List File	Examine any text file.	Multiple
Input	Display simulated input assignments.	One
Output	Display simulated output assignments.	One
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

4.10.1 Windows//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.

Adjust width of columns by dragging the gap between labels.

Binary is disassembled and listed. Illegal opcodes are listed as numeric constants.

Operands are decoded and interpreted as symbolic references when appropriate.

Next instruction is highlighted in red.

Double click on address or label field to set or clear breakpoints. Enabled breakpoints display blue.
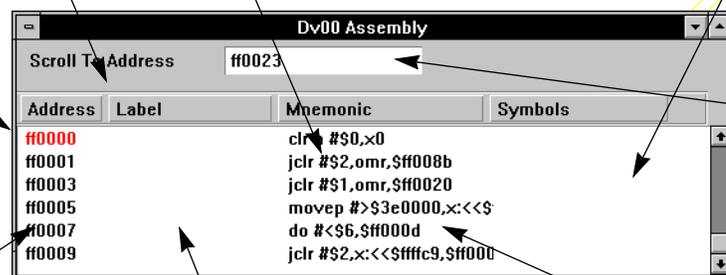
Enter start of memory to display. May use filenames, line numbers, labels and addresses.

If debug information loaded, nearest label is shown with offset.

Click on a mnemonic field, type in new instruction, <CR> to store and step to next instruction.

Figure 4-66 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.

4.10.2 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.
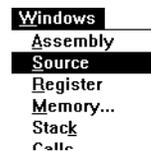
A halt execution breakpoint may be set with the Source window. Double-click on a statement to set or clear the breakpoint. The breakpoint is added to the breakpoint list, displayed in the breakpoint window and highlighted blue in the Assembly window. The presence of the breakpoint is not indicated in 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 4-67 Source Window (no source)

Single line display summarizes status.

Current instruction highlighted in red.

Scroll through whole program. Display reverts to current line after execution.

Double-click on a statement to set a breakpoint.
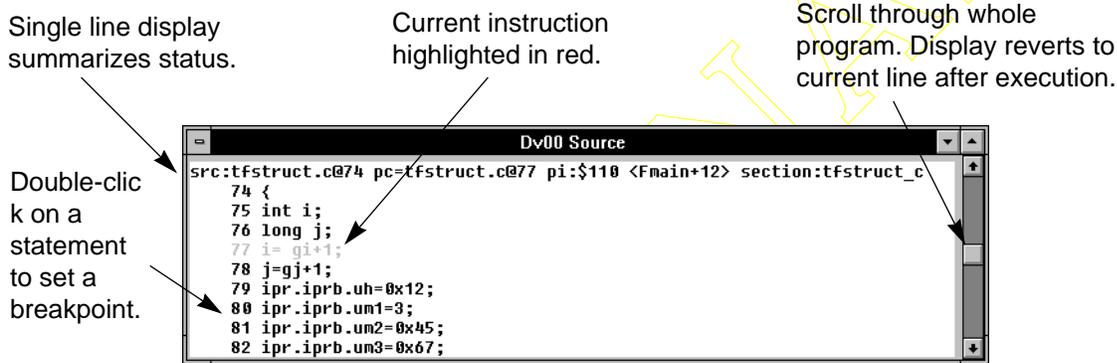
Figure 4-68 Source Window (source file present)

4.10.3 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.
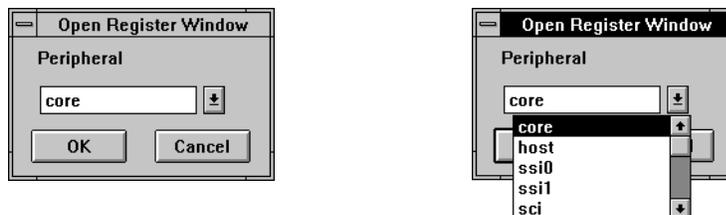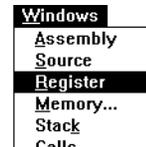
Figure 4-69 Register Window Peripheral Group Selection

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 view 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.

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.

Displays registers for core or peripheral for Current Device.

Register values displayed in hexadecimal or radix set as display radix. Enter values in specific radix or default radix. (see Modify//Radix//...)
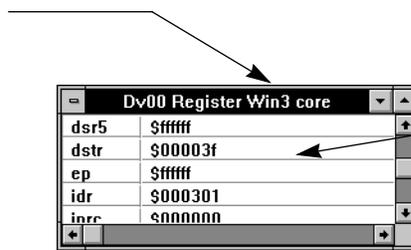
Single click on a value to select. Type new value and <CR> to change. Highlights red and selects next value to change.

Scroll to view desired registers.

Figure 4-**70** Register Window

4.10.4 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 a pull-down list in the dialog box.
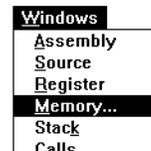
Figure 4-**71** Windows//Memory Dialog Box

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.

Memory window displays one memory space for a device.

Values shown in Display Radix.

Enter an address for start of display area. Use filenames, labels, line numbers and memory addresses

Open multiple windows for other devices, address spaces or discontiguous memory ranges.

Enter new values in Default Radix or use explicit radix specifier.

Click to select a location. Type new value and <CR> to save and select next

Figure 4-**72** Memory Window

4.10.5   Windows//Stack

The Stack window displays the hardware stack. It may be resized and scrolled to view as much or as little as required.
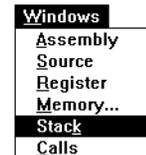
Drag gaps to adjust column width

Figure 4-**73** Stack Window

The hardware stack is used by the subroutine call instructions, interrupt handling and by some other instructions. In C procedures, 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.

4.10.6   Windows//Calls

The Calls window tracks C procedure calls. Each procedure call adds another stack frame, each return removes one. Entry 0 is the most nested procedure, that is, the top entry on the stack; the highest number is the main() procedure.
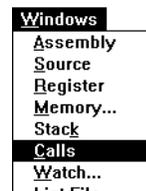
Each entry has a nesting level number, the PC return address (i.e., the address after the procedure call), and the name of the procedure. The top level represents the entry to the debug monitor, and so indicates the next instruction to be executed.

The call stack also indicates the context to use for evaluating C expressions. As each procedure 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.

Shows one entry for each call to reach current location.

P-space address is address of next instruction to execute in calling procedure.

#0 indicates next instruction to execute in current procedure

See also Modify//Up and Modify//Down.

Double-click on a stack frame to select as Evaluate context.

Note PC address format—0xHHHH is hexadecimal.

```
 —         Dv00 Calls         ▼ ▲
 #0  p:0×6f in ne3 ()
 #1  p:0×7b in ne2 ()
 #2  p:0×87 in ne1 ()
 #3  p:0×93 in main ()
```

Figure 4-**74** Calls Window

4.10.7   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.

```
Windows
 Assembly
 Source
 Register
 Memory...
 Stack
 Calls
 Watch...
 List File...
 Input
```

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 procedure call or return from a procedure), 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.
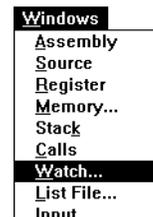
An expression which has gone out of scope because of procedure a call may be evaluated and displayed by selecting the stack frame for the evaluation context. See Modify//Up and Modify//Down. The stack frame assignment remains in effect only until the next instruction is executed. An expression out of scope because of procedure exit can not be evaluated until the procedure is next invoked, as its variables no longer exist.

Select window number. Multiple Watch windows may be opened for each device.

Enter expression. Enclose C expressions in brackets {}.
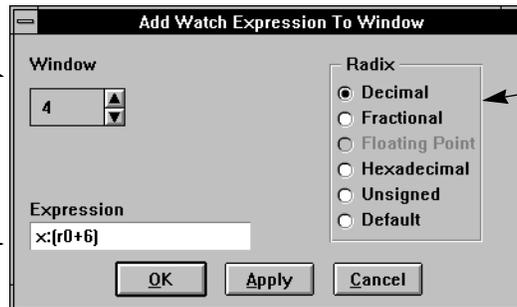
Select display radix for expression. C expressions default to type of expression.

Figure 4-**75** Windows//Watch Dialog Box

4.10.8   Windows//List File

Views an 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.
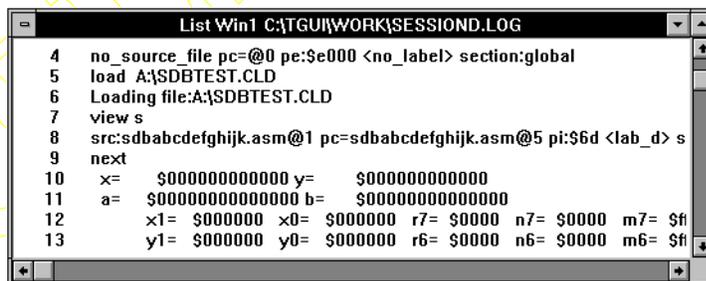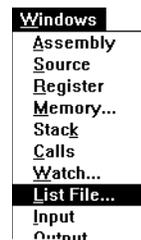
Figure 4-**76** List File Window

Note: 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.

4.10.9   Windows//Input

Displays all simulated input which has been assigned for the current device. An Input window may be opened for each device.
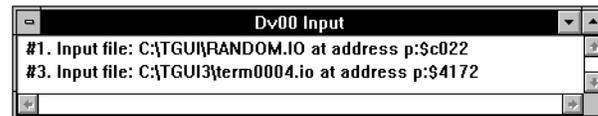
Figure 4-77 Input Window

4.10.10   Windows//Output

Displays all simulated output which has been assigned for the current device. An Output window may be opened for each device.
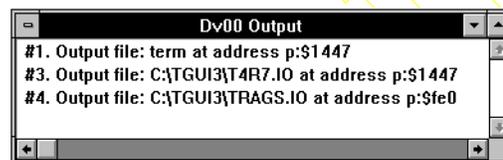
Figure 4-78 Output Window

4.10.11   Windows//Breakpoints

Displays, 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 that are set by clicking on the source window are identified by the line number. Breakpoints that are set by clicking on the Assembly window have the address listed. Disabled breakpoints are marked with the word 'disabled'; all other listed breakpoints are enabled.

Breakpoints listed for current device only. Those not **disabled** are **enabled**.

Required action indicated by letter:
h—Halt.
i—Increment counter.
s—Show registers & memory in Session.
n—Note breakpoint.
x—Execute macro file.

Breakpoint address displayed as hexadecimal.

First three are SW breakpoints. Last is HW breakpoint.

Double-click to enable or disable breakpoint.
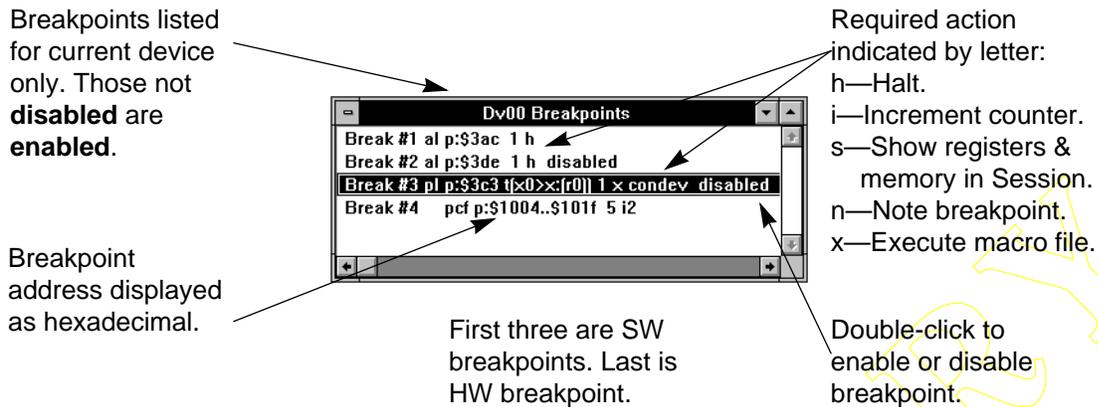
Figure 4-**79** Breakpoint Window

4.10.12   Windows//Command

The Command window provides the main interface between the user interface and the rest of the system:

- The 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.

One command window handles all commands for the system.

Click on history line to select, <CR> to execute. May edit first.

If history is edited, original command is always restored. When executed, the new command is added to history.

Use scroll bar to view command history.

Click and type commands directly into command window.

Summary help lists commands. Space bar cycles through commands. Type portion in red and a space, command is completed and help gives syntax for that

```
Command
radix d b2
radix h p:$4000..$401e
radix h a
radix h a0
wasm
device Dv2
change b1 $ffffe
change b2 $ffffe
step
2> frame #2

FRAME [#fn]
```

Prompt is current device No.

Command abbreviations shown in red

Figure 4-80 Command Window

4.10.13 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.

```
Windows
 Assembly
 Source
 Register
 Memory...
 Stack
 Calls
 Watch...
 List File...
 Input
 Output
 Breakpoints
 Command
 Session
 Tile
 Cascade
```

- 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.

Commands echoed in Session window

Window title shows current device.

Scroll bar to review output buffer.

Enlarge or maximize window to display more of the device buffer.

```
 ─                              Dv00 Session                          ▼ ▲
                                                                       ↕
break p:$04a6
list .
view r
display on x:$0001..$0004
go
Break #1 p:$0051 h ;dev:0 pc:0051 cyc:198
  x=       $000000000000 y=       $000000000000
  a=      $00000000000000 b=      $00000000000000
             x1=  $000000    x0=  $000000   r7=  $0000  n7=  $0000  m7=  $ffff
             y1=  $000000    y0=  $000000   r6=  $00c3  n6=  $0000  m6=  $ffff
  a2=  $00 a1=  $000000    a0=  $000000   r5=  $0000  n5=  $0000  m5=  $ffff
  b2=  $00 b1=  $000000    b0=  $000000   r4=  $0000  n4=  $0000  m4=  $ffff
                                           r3=  $0000  n3=  $0000  m3=  $ffff
  pc=  $0052    sr=  $0300  omr=      $02   r2=  $0000  n2=  $0000  m2=  $ffff
  la=  $0000    lc=  $0000                  r1=  $0000  n1=  $0000  m1=  $ffff
ssh=  $03ae   ssl=  $0300   sp=      $01   r0=  $00c0  n0=  $0000  m0=  $ffff
ipr=  $0000   bcr=  $ffff
cyc=000198   ictr= 000015  cnt1=     000000 cnt2= 000000 cnt3= 000000 cnt4= 000000
x:$0001=       $000400       $00ffbe       $00ffbe
x:$0004=       $0000be
p:$0051 055e3c          = move ssh,x:(r6)+ ; x:(F__c_sig_handlers+129)          ↓
```

Values changed since last output displayed in red. Error messages also in red.

Memory locations and registers output at break selected by Display//Display menu.

Initial setting: Display all registers and no memory.

Figure 4-81 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.

4.10.14   Windows//Tile, Cascade (Microsoft 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 or scrolled off the edge of the main window.
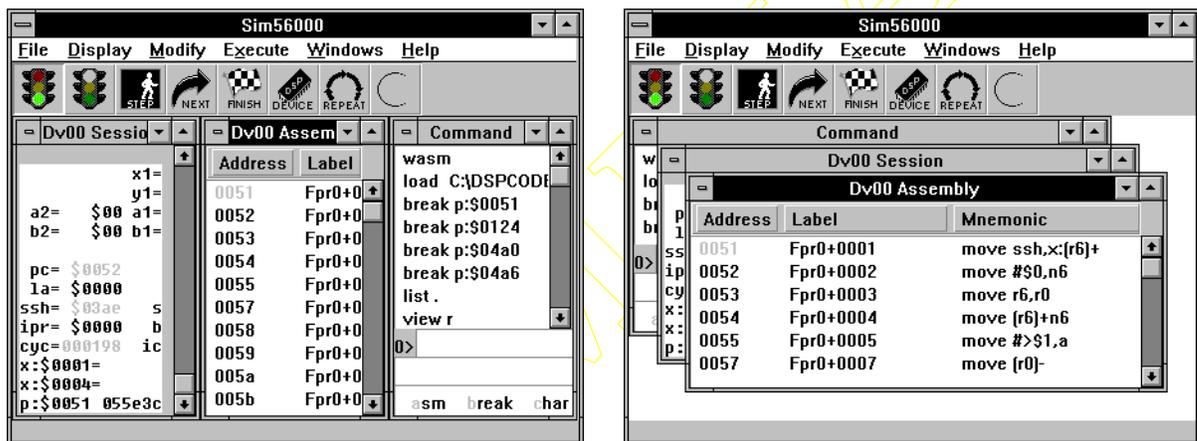
Tiled window   Cascaded window

Figure 4-82 Tiled and Cascaded Windows

4.11 HELP MENU

The <u>H</u>elp menu item calls up the command line help for the ADS, which is displayed in the Session window. This lists all the commands available, with a syntax summary for each. Acceptable abbreviations are shown in red.
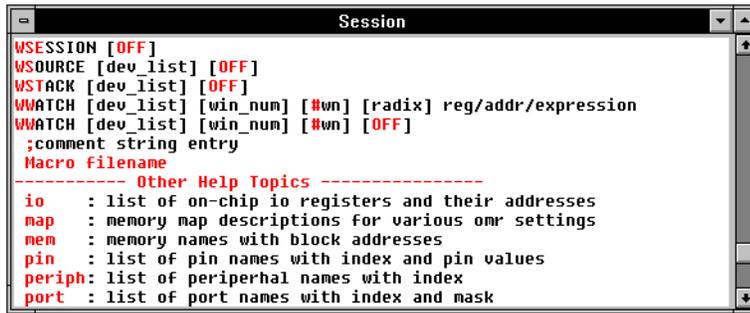
Figure 4-83

Further command details are available. Using the Command window, type Help (H <space> is enough) followed by the help topic (the command name (or abbreviation) or port name, etc.) and further details are output to the Session window.

Note: The Help//Help output is nearly 100 lines long, and fills the output buffer. Any previous output in the Session window will be lost.
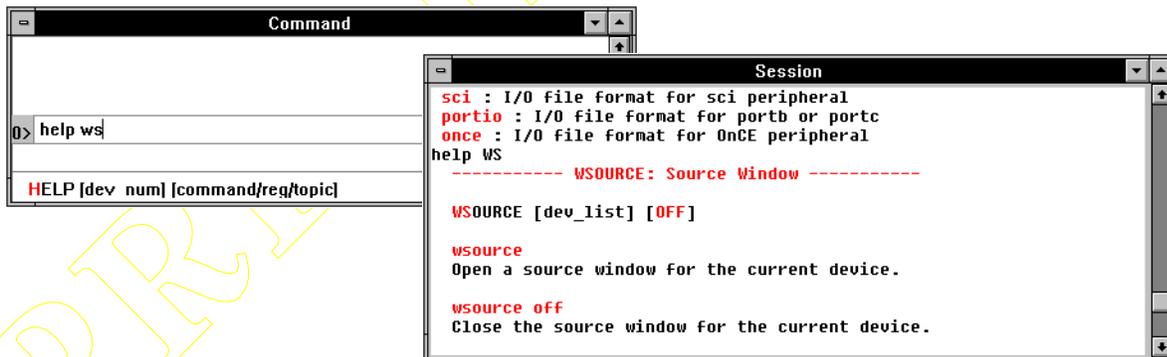
Figure 4-84 Help on a Specific Topic

4.12   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.

4.12.1   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.

4.12.2   Stop Button

The Stop button interrupts DSP program execution and returns control to the
user. The command 'force b' appears in the Session window. If a macro
command file is executing, it is aborted by the Stop button. This button is equivalent to
Execute//Stop.

4.12.3   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.

4.12.4   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.

4.12.5   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.

4.12.6   DEVICE Button

The DEVICE button opens the 'Set Default Device' dialog box. This selects which is 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.
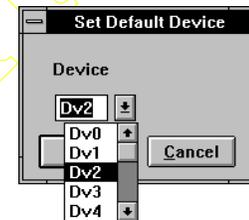
4.12.7   REPEAT Button

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>.

4.12.8   RESET Button

The RESET Button generates a reset command for the current device. It is equivalent to Execute//Reset, with the device mode unchanged.

SECTION 5

FUNCTIONAL DESCRIPTION

5.1 INTRODUCTION

The Application Development System (ADS) user interacts with the target DSP through two subsystem components, the host computer interface and the Command Converter controller. The host computer interface consists of a program written in the C language which interacts with a host computer bus interface card. The Command Converter consists of a program written in DSP56002 assembly language which interacts with the host computer bus interface card and the target OnCE port.

It should be noted that older versions of Motorola DSP products use the OnCE port protocol, while newer versions of Motorola DSP products use the IEEE JTAG 4-wire protocol to interact with their OnCE port. New versions of the Command Converter support both OnCE and JTAG protocols.

Commands entered from the host computer's keyboard are parsed and a series of low level command packets are sent to the Command Converter. The Command Converter translates these low level command packets into serial sequences that are transferred to the target DSP via its OnCE port. The OnCE port provides the necessary control to the target so programs may be loaded or saved, registers read or modified, and hardware breakpoints set or cleared.

The host computer interface is designed to communicate with as many as eight targets. This requires a special software protocol to avoid data collisions between one target and another. The purpose of this section is to describe the subsystem components of the ADS to give a better understanding of the communication link between the user and the target DSP.

5.2 HOST COMPUTER HARDWARE

The host computer hardware interface provides the communications link between the user and the Command Converter. The ADS user interface program uses a software handshake when communicating with the Command Converter. There are signals defined on the host computer bus interface card which are used for requesting and acknowledging information transfer. Since the handshake is software driven the transfer rate will be dependent upon the host computer bus speed and its operating system. This section describes the host computer interface hardware and software components.

5.2.1   Host Computer Bus Interface Card

Figure 5-1 shows a block diagram of the host computer interface card for all supported computer platforms. The interface consists of three fixed addresses in the host computer I/O memory map. Host computer interface card address zero is used as a control port for selecting, resetting, or interrupting one or more Command Converters. Address one reads and writes eight bit data bytes to one or more Command Converters. Address two acknowledges Command Converter service requests and selects group members for multiple Command Converter commands. All data is passed high order byte first. For example, in a 32-bit transfer, bits 31–24 are transferred first, followed by bits 23–16, bits 15–8, and then bits 7–0.
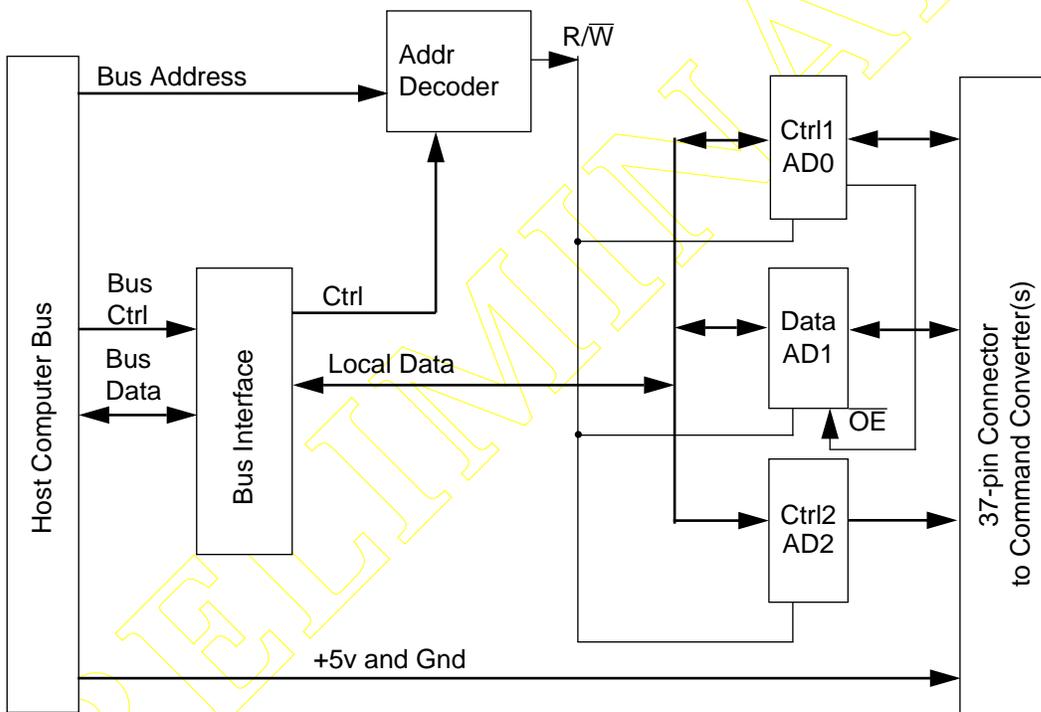
Figure 5-1 Host Computer Bus Interface Card

Command converters always act as slaves to the host computer interface card to avoid transfer collisions. The ADS user interface program allows as many as eight Command Converters to be addressed using one host computer interface card. Command converters may be addressed in groups or individually depending upon the command and command arguments entered by the user.

Host computer interface card address zero has eight output control lines. These output control lines are asserted using positive logic (VOH = TRUE). Three address zero signals (ADM_SEL0, ADM_SEL1, ADM_SEL2) select a Command Converter before sending a

command. Further information on the method of Command Converter selection is discussed in subsequent sections.

Two handshake signals originate at the host computer and are used to pass data to and from a Command Converter. HOST_REQ initiates a data byte transfer to a Command Converter while HOST_ACK acknowledges receipt of a data byte from a Command Converter.

Two control signals (ADM_BRK,ADM_RESET) allow the user to assert an interrupt or a reset exception on a single Command Converter or a group of Command Converters. The ADM_BRK signal is used to put the Command Converter back into Command Entry mode. while the ADM_RESET signal is used to reset the Command Converter.

The Command Converter informs the host computer of target DSP entries to the Debug mode of operation by asserting the HOST_BRK signal. The ADS user interface program on the host computer periodically polls the HOST_BRK signal from the keyboard polling routine. If the HOST_BRK signal is asserted the host computer will determine which Command Converter is requesting service by reading the ADM_INT signal. The INT_ACK signal is asserted by the host computer when a service request has been recognized. Further details on the functions of each signal will be given in a subsequent section. **Figure 5-2** on page 5-6 illustrates the 37-pin cable and the direction of the signal groups.

5.2.2   Host Computer Interface Cable

The host computer interface card interacts with the Command Converters via a 37-pin ribbon cable assembly. Each end of the ribbon cable has a 37-pin DIN receptacle connector. The cable assembly is approximately 4 feet in length and is designed so that additional Command Converters may be easily attached to the existing cable by crimping a new DIN connector.

Normally the ADS is shipped with an Application Development Module (ADM) and power for both the Command Converter and ADM are supplied by the host computer interface card via the cable. The ribbon cable is not designed to draw more than 2 amps current at 5 volts. Since each Command Converter draws approximately 250 milliamperes, it is safe to power all 8 units via the cable, but the target systems must be powered by a different source to insure correct operation.
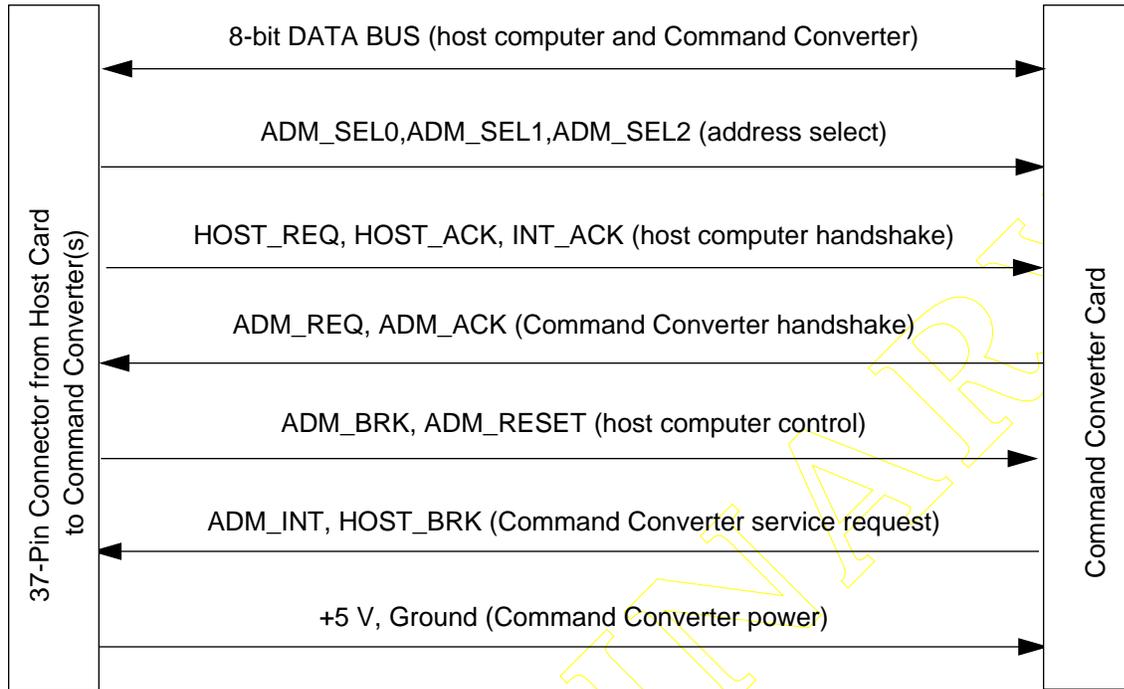
Figure 5-2 37-Pin Host Computer Interface Cable

5.3   COMMAND CONVERTER CARD

The Command Converter is based on a DSP56002, which uses its on-chip resources to minimize and simplify the interface to the target OnCE debug port. Communication with the host computer is via the DSP56002 Port B or the SCI port, while the serial interface to the target DSP is via the DSP56002 SSI port.

Each Command Converter has a unique address ranging from 0 to 7. This allows the user to debug multiprocessor systems where as many as eight Command Converters are physically in the target system. JG2 of the Command Converter card defines the address selected for that card.

When a user wishes to communicate with a Command Converter, its address must first appear on the interface cable output control lines before the Command Converter can communicate with the host computer. The host computer must then hold that address on the control bus until communication has ended.
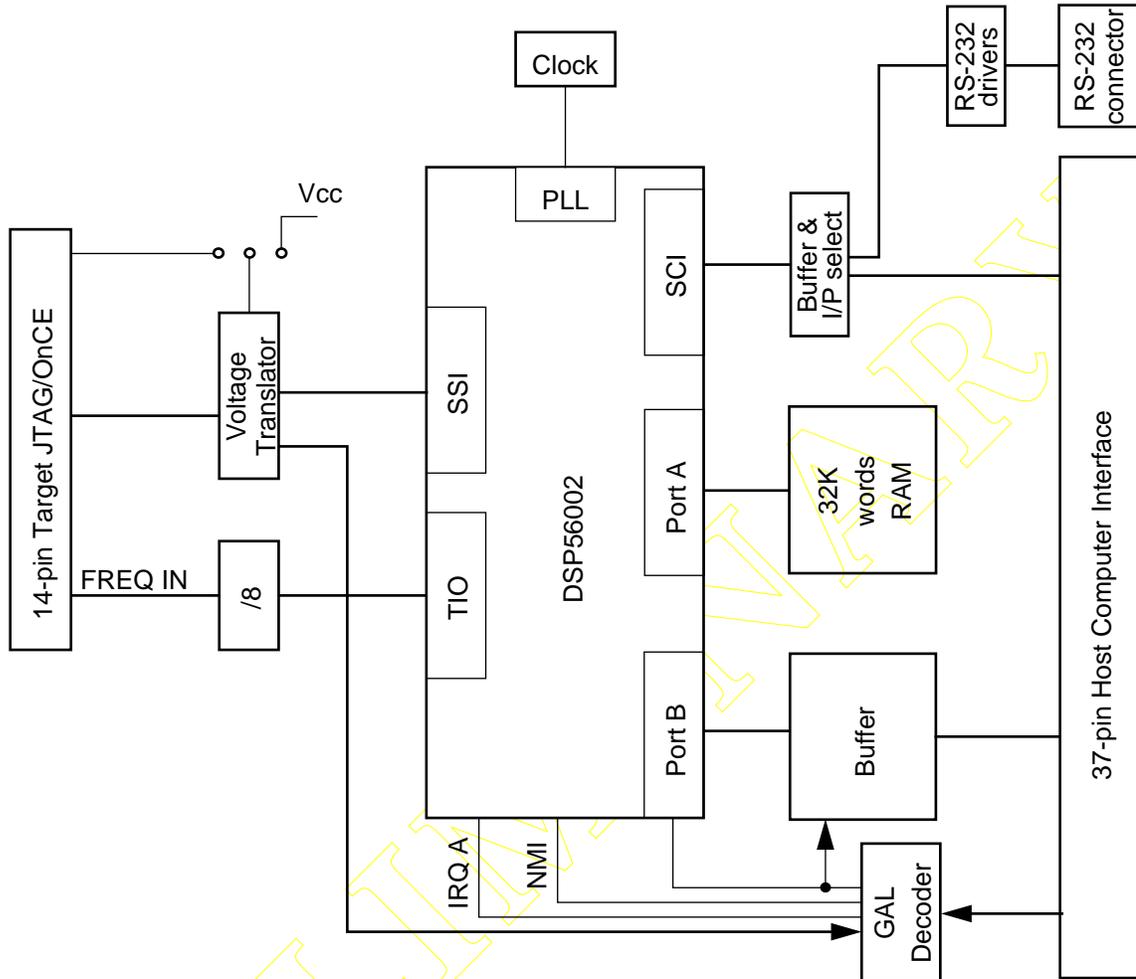
Figure 5-3 Command Converter Block Diagram

5.3.1 Command Converter Handshake Signals

The DSP56002 on the Command Converter card is configured such that bits 0–7 of its Port B are used for 8-bit data transfers and bits 8–14 are used for data transfer control.

There are three output control bits in the middle order byte of the DSP56002 Port B data word. These bits are the ADM_INT, ADM_REQ, and ADM_ACK signals. ADM_REQ and ADM_ACK, act as handshake lines for reading and writing data. ADM_INT acts as a flag to indicate whether the ADM is requesting host computer service. These three control bits are part of the host parallel control bus. They are enabled when the host computer selects the ADM.

There are three input control bits in the middle order byte of Port B data word that represent the HOST_ACK, HOST_REQ, and INT_ACK. These signals are sent from the host computer for reading and writing data. INT_ACK informs the monitor that the host computer has received its service request and is ready to communicate.

$\overline{\text{HOST\_BRK}}$ is a wired-or control line. $\overline{\text{HOST\_BRK}}$ is used by the Command Converter to inform the host computer whenever the target DSP has entered the Debug mode of operation. Since more than one Command Converter may be started for a user debug session more than one may hold $\overline{\text{HOST\_BRK}}$ active low at one time. Once this signal is asserted it may only be deasserted by the host computer or by a Command Converter reset.

5.3.2 Command Converter Interface Connector

The target application must have a 14-pin connector to interface to the Command Converter controller. This interface comprises nine signals and three ground connections on 7 row $\times$ 2 column male pins which are 1/10 inch center to center as illustrated in **Figure 5-4** on page 5-9.

Since the target system will have a resident reset circuit, it is recommended to have an AND gate in series with the $\overline{\text{CC\_RESET}}$ signal. This will insure that the DSP will be reset with a valid VOL level from either the target reset circuit or from the Command Converter. The pull-down resistors are to insure that no false signals are propagated to the JTAG/OnCE circuit when the Test Data Input/Debug Serial Input (TDI/DSI) and Test Data Clock/Debug Serial Clock (TCK/DSCK lines are active. The Test Data Out/Debug Serial Output (TDO/DSO) pullup is to insure that the Debug Acknowledge signal from the OnCE circuit is deasserted. The debug request ($\overline{\text{DR}}$) pullup is to insure that the Command Converter controls when the target DSP is put in the Debug mode.

When a command has been received by the Command Converter from the host computer, a series of serial command packets are sent to the target OnCE debug port. These serial command packets consist of an 8-bit command followed by a 16-,24-, or 32-bit serial read or write of data. The serial bit width is dependent on the DSP architecture.

The Command Converter will always act as the clock master of the serial transfers. In order to transfer a command, the target DSP must be in the Debug mode of operation. This may be accomplished by a force b or force r command from the user interface program.

TOP VIEW
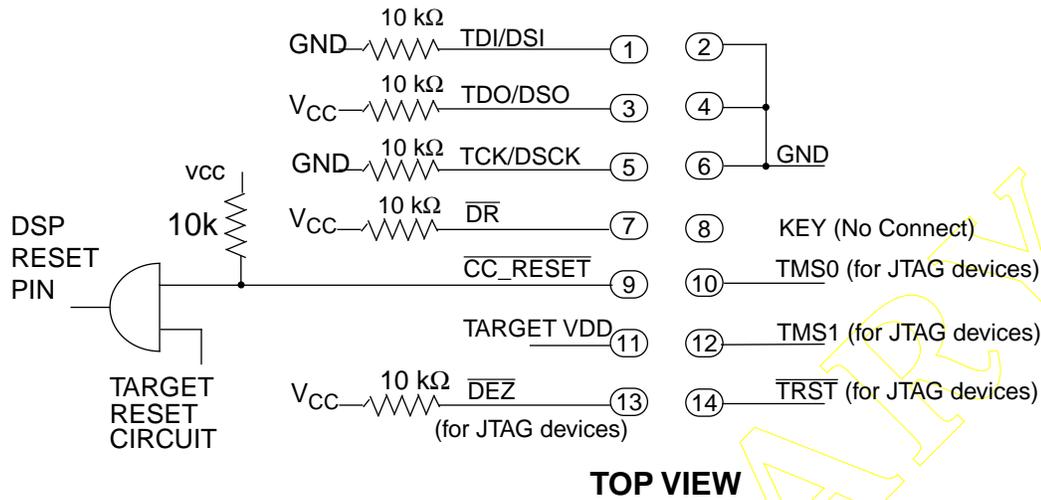
Figure 5-4 Target System OnCE Interface Connector

The force b command will cause an assertion of the $\overline{DR}$ pin of the target DSP until an acknowledge is received on the TDO/DSO pin. The force r command will cause an assertion of the $\overline{DR}$ pin while also asserting the reset of the target DSP. When the target reset pin is deasserted the $\overline{DR}$ pin will remain asserted until an acknowledge is received on the TDO/DSO pin.

The host computer user interface program, as well as the Command Converter monitor program, each have a flag which is set when the force b command is issued. This flag tells these programs that the target system is in the Debug mode of operation and is ready to receive commands. For more information refer to the **CF**ORCE command in **Section 3**.

5.3.3   Multiple Target Connections

The basic JTAG connection comprises 5 pins as illustrated in **Figure 5-5**.
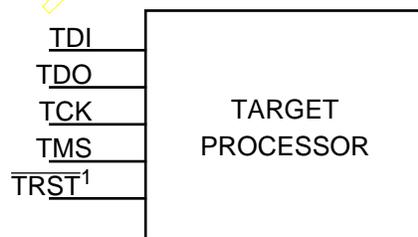
[1] Optional connection

Figure 5-5 JTAG Connections

Multiple target devices may be connected in series, allowing a single Command Converter JTAG/OnCE connector to control multiple devices, as in **Figure 5-6**. Data flows from the JTAG host, into each JTAG implementation through TDI, out through TDO and into TDI in the next chip, eventually returning to the JTAG host.
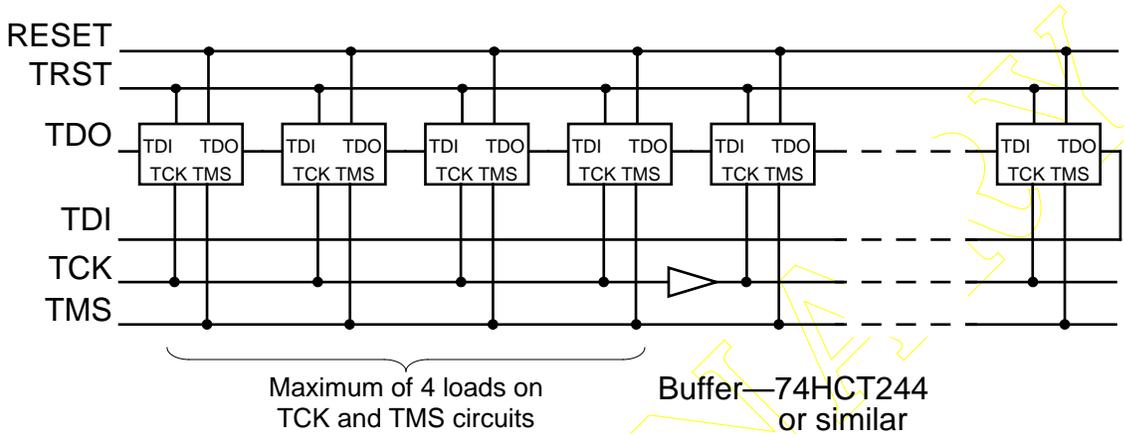
Figure 5-**6** Multiple JTAG Target Connections (1)

5.3.4   TCK Drive and Timing Considerations

The signals from the Command Converter are TDO, TCK and TMS, and $\overline{\text{TRST}}$. Signal TCK requires fast rise and fall times dictated by the TCK pin timing specification, and consequently attention must be given to the drive capabilities of the circuits driving this signals. There is no problem with TDO, as each TDO output is connected to only one TDI input. TMS need only be valid at the rising edge of TCK, similarly there is no problem with $\overline{\text{TRST}}$ as the reset signal is not subject to the timing constraints of TCK.

There is a potential problem with driving the TCK circuit with a large number of target devices. The problem is related to the rise and fall times of TCK, caused by excessive capacitance, which can cause communication problems with a single circuit connecting multiple TCK input pins.

Acceptable transition times may be achieved for TCK by driving no more than four JTAG inputs from each buffered output. This may be achieved with two configurations.

Figure 5-6 above shows one method. Here (in effect) one track connects each of the TCK inputs. A buffer is placed in circuit after (at most) each fourth input to restore the signal quality for subsequent inputs. The propagation delay of the buffer is not significant.

Figure 5-7 shows another possible configuration which also enables signal quality to meet the requirements. In this configuration the signal is split and buffered into a number of parallel TCKn signals. Each of these signals may drive up to 4 TCK inputs.
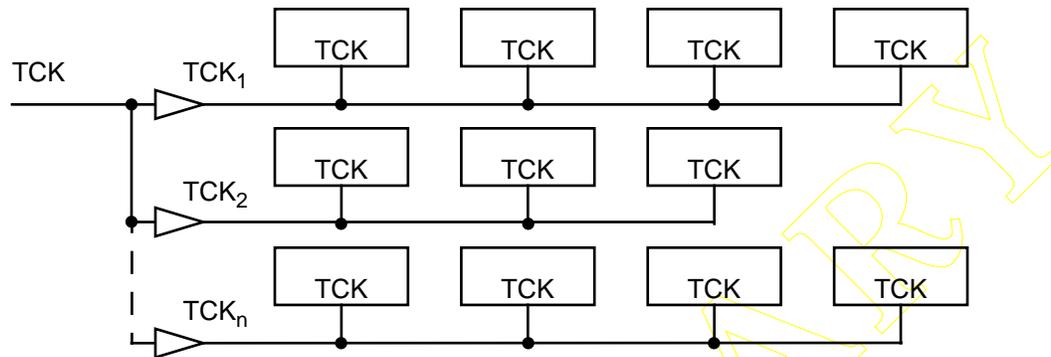
Figure 5-7 Fan Out of TCK at Source

Either configuration above is equally valid. The choice will depend on practical considerations related to each project, or a combination could be used.

JTAG signal TMS may also need some consideration. Although not subject to the strict requirements for TCK, it is still important that TMS has settled to a valid level at the rising edge of TCK.

5.3.5 Resetting Target DSP Devices.

The RESET signal from the Command Converter is typically connected to all target devices on a JTAG chain. $\overline{RESET}$ is asserted by the ADS command **F**ORCE **R**. All devices on the JTAG chain handling the specified device are reset. Execution control is established immediately on exit from reset, before any instructions are executed. The sequence of events is illustrated in **Figure 5-8** on page 5-12. Since all targets on the JTAG chain are connected to the same $\overline{RESET}$ signal, all devices enter reset. The JTAG controller is still active during reset, and while $\overline{RESET}$ is held low, the JTAG instruction DEBUG_REQ is clocked in. When $\overline{RESET}$ is deasserted, the device is immediately in Debug mode, with no instructions executed since releasing $\overline{RESET}$.
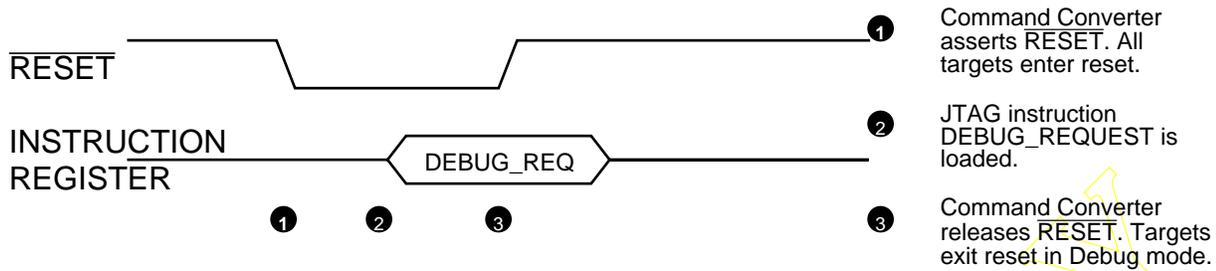
Figure 5-8 Reset JTAG device with $\overline{\text{RESET}}$ Signal

5.4   OnCE PORT ARCHITECTURE

This section covers a global view of the OnCE architecture. To get specific details on the OnCE port registers and addresses for the 16-, 24-, or 32- bit DSPs it is best to refer to the pertinent DSP user manual. To control the target DSP with minimal die area penalty the OnCE port was designed so users could interact directly with the program controller. This port eliminates the need for a special debug monitor resident in the user program memory map. **Figure 5-9** illustrates the OnCE port architecture.

5.4.1   OnCE Controller

The OnCE port controller acts as a serial slave interface which is controlled by the Command Converter. To communicate with the OnCE controller the DSP must be put in the Debug mode of operation. The Debug mode may be entered from a hardware or software breakpoint, single stepping through opcodes, or from an assertion of the Debug Request ($\overline{\text{DR}}$) pin.

A state machine decodes 8-bit commands and controls interaction with the OnCE registers. The OnCE controller block also contains a clock counter and circuits for synchronizing external clocks with the system clock. Exiting the Debug mode can only be achieved by a OnCE command or from the Reset state without the $\overline{\text{DR}}$ pin asserted. The OnCE Status/Control Register (OSCR) is used to enable or disable hardware breakpoints or the single step mode. It also provides status information on how the Debug mode was entered.

5.4.2 Program Controller Pipeline Information

When the Debug mode of operation is entered, the current state of the DSP pipeline is saved in three registers. The Program Instruction Latch Register (OPILR) holds the next opcode to be executed when returning to the User mode of operation. The host computer should immediately save this register when the Debug mode of operation is entered so the exact state of the program controller pipeline may be restored when returning to User mode.
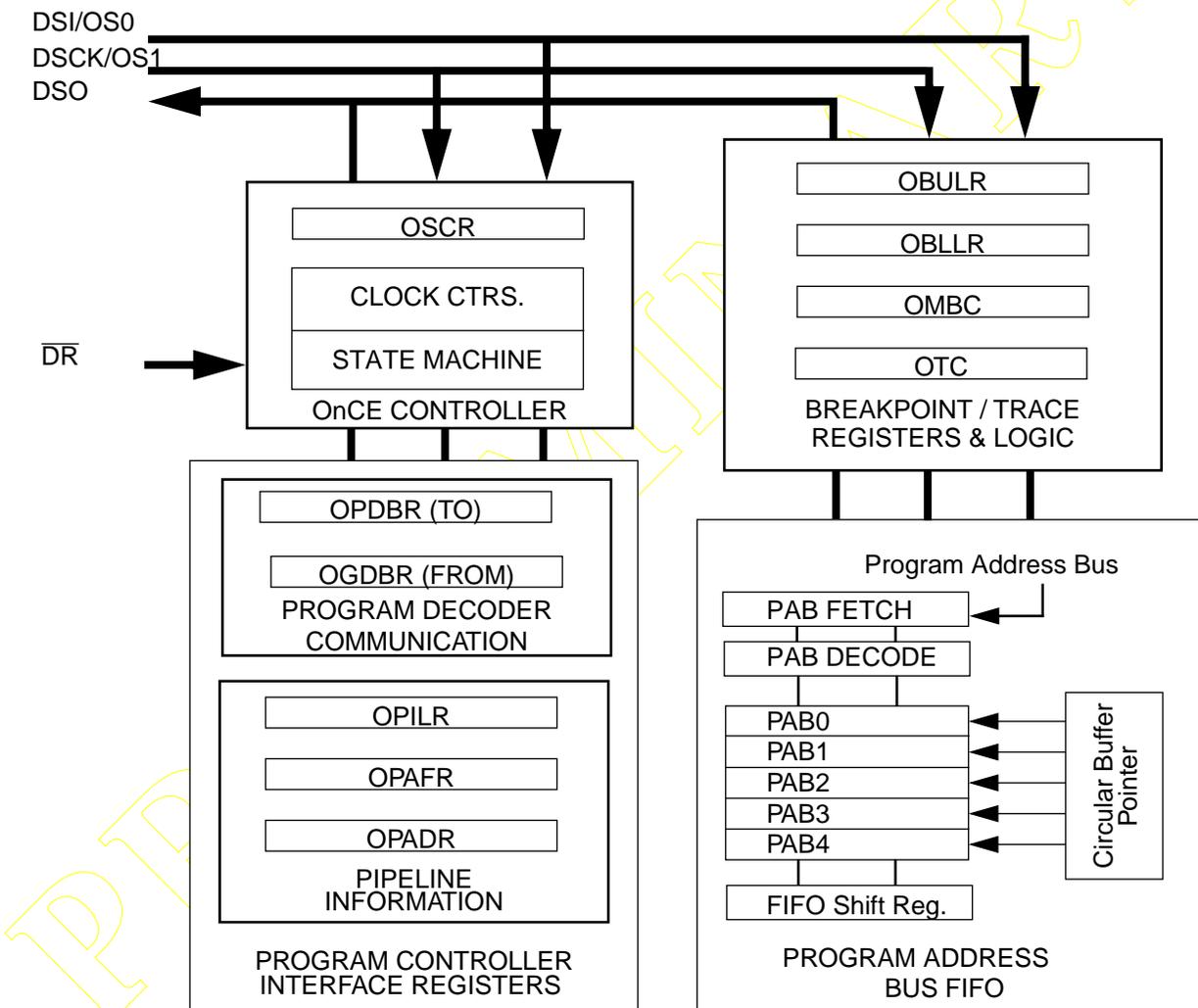
Figure 5-9 OnCE Port Architecture

Since the DSP is a pipelined machine there is no program counter in the DSP programming model. Program flow is dictated by a program address generator so special registers are made available to give insight on where the program currently is. The Program Address Fetch Register (OPAFR) holds the address of the opcode that was fetched for decoding and the Program Address Decode Register (OPADR) holds the address of the opcode that resides in the instruction latch. These registers are read only and are not affected when in the Debug mode.

5.4.3 Program Address Bus FIFO

When programs are executing out of on-chip Program RAM there is no external bus activity. This is important for accessing multiple internal buses simultaneously during a single instruction cycle. To provide better visibility into the user's program flow, the last five addresses of DSP opcodes executed may be read from a FIFO circular buffer. This FIFO is frozen when in the Debug mode and will not change until re-entering the User mode of operation. The 8-bit command to read the FIFO automatically increments the FIFO pointer to the next location. Successive reads will always give the oldest address first and the newest address last. All five registers should be read when addressing the FIFO so the pointer will always be deterministic. The Program Address Bus FIFO is very useful for evaluating interrupt service routines, program flow changes or code which is executed from internal RAM. The FIFO is read only and is not affected when in the Debug mode.

5.4.4 Program Decoder Communication

In order to communicate with the DSP user registers the OnCE port interacts with the DSP program controller unit via a Program Data Bus Register (OPDBR). DSP opcodes may be fed serially into the DSP program decoder unit bypassing the pipeline instruction fetch stage. Opcodes are always written first followed by operands when executing two word instructions. OnCE port registers are not accessible from the DSP programming model. DSP programming model and peripheral registers are accessible only through a Global Data Bus Register (OGDBR). This read only register is located in the user X data memory peripheral address space. The OGDBR is the transfer register for passing information back to the OnCE port. For example, to read the R0 register value, the OPDBR is loaded with the opcode "move R0,X:OGDBR". An 8-bit command is then sent to read the OGDBR to retrieve the contents of the register value transferred.

5.4.5 Hardware/Software Breakpoints

Users may halt program execution and enter the Debug mode via hardware and/or software breakpoints. Hardware breakpoints may be set on program opcode fetches, program memory moves or data memory accesses. A Memory Breakpoint Counter (OMBC) must be loaded to cause a halt of program flow on the nth occurrence of the breakpoint. The Debug mode of operation will be entered after the opcode at the breakpoint address has been executed. The OBC should be loaded with $n-1$ times of breakpoint occurrences. The OnCE port is scalable so that single chip DSPs of different sizes can use the OnCE controller concept. The die area of the DSP96002 permits extra features that are not feasible on less powerful devices; therefore, hardware breakpoint registers vary in size and features. For example, the DSP96002 has separate data and program memory breakpoint comparators, while the DSP56002 only has one set of breakpoint comparators. The DSP96002 and the DSP56002 have a Program Upper Limit Register (OPULR) and a Program Lower Limit Register (OPLLR). These registers allow breakpoint address ranges to be defined. The DSP56L811 uses only a single breakpoint address (OPBR). Conditional or unconditional software breakpoint opcodes (DEBUGCC) may be set in Program RAM. The DSP Status Register may be evaluated in real-time to determine whether to halt program execution depending on whether particular bits are set or clear. For example, setting a DEBUGEQ opcode in program flow will only cause the program to halt if the status register Z bit is set. Software breakpoints are set on opcode addresses only.

Software breakpoints are useful for halting program flow only after a particular condition is true. Users can select when to stop program execution based on predetermined conditions in a program's behavior. When the Debug mode of operation is entered after executing a software breakpoint, a special bit in the OnCE Status/Control Register (OSCR) will be set to tell the user that the Debug mode was entered from a software breakpoint. The same holds when the Debug mode is entered from a hardware breakpoint. There is also a special bit in the OSCR which flags the user that a hardware breakpoint has been accomplished.

5.4.6 Program Single-Stepping

To evaluate programs one opcode at a time, the OnCE controller provides a single step capability. Single stepping requires the chip be put into the User mode of operation so the pipeline registers are updated after the execution of the opcode. This is accomplished by loading a Trace Counter register (OTC) with the $n-1$ opcodes to execute. The OTC allows users to multiple step opcodes in real-time so routines may be quickly executed. Entering the Debug mode from a single step will cause a special bit in the OSCR to be set to flag the user how the DSP exited the User mode of operation.

5.5   HOST COMPUTER SOFTWARE

The host computer user interface program comprises of two levels of support. There is a
turn-key program which provides immediate access to the target and there is also a set
of libraries and C language support modules for customized interfaces. The C modules
of source are provided to allow default values to be changed when entering the ADS and
the libraries are provided to facilitate assembly and disassembly of target DSP
instructions. Also, screen and keyboard characteristics may be changed and access to
help information may be changed as well, as illustrated in **Figure 5-10**.
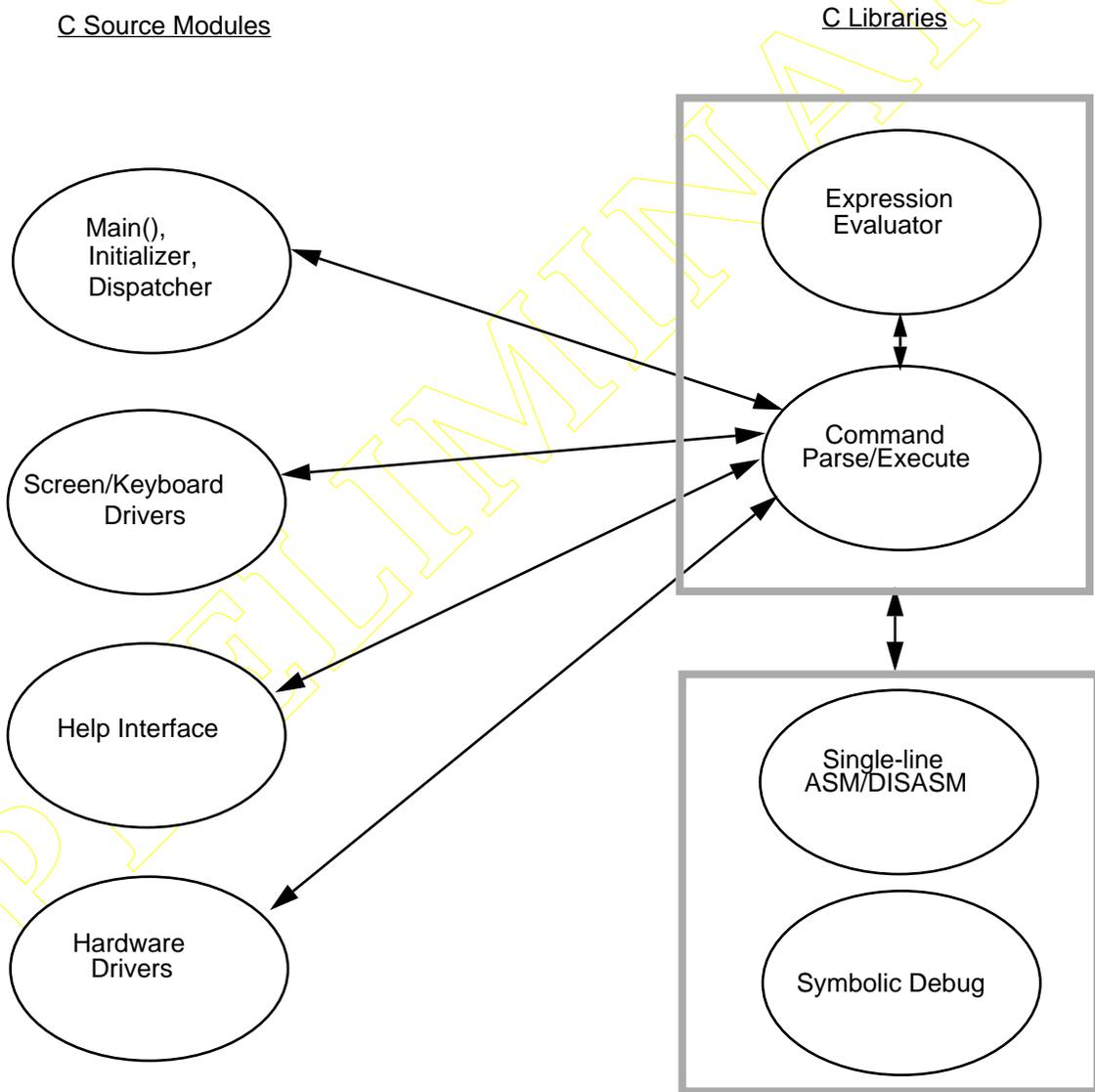
Figure 5-10 Host Computer User Interface Program

If a user wishes to interact with the hardware directly, a set of functions exist which may be called by a separate program. These functions provide a means of communicating with the target DSP without having to completely understand the communications protocol. The information contained in this section is made available so that users may write their own programs to communicate with the hardware. The distribution diskettes which contain the software programs for the Application Development System will contain these C language modules.

5.6   COMMAND CONVERTER SOFTWARE

The Command Converter exits reset in the Bootstrap mode of operation and boot loads its monitor program from the host interface or an RS-232 serial link. The ROM loader loads a secondary loader via the SCI port. This loader program loads the monitor software via the host interface.

The X data RAM is used as a scratch pad area for storage of temporary information and also for information retrieved when performing OnCE port read commands. X data memory locations $00–$16 are used as a storage area and should not generally be changed by the user. The Y data RAM is used as an overlay area for storing the commands and data sent to the OnCE port by the DSP56002 Synchronous Serial Interface. Y data RAM locations $00–$0F are used for storing OnCE command sequences. **Figure 5-11** illustrates the memory usage of the DSP56002 RAM.

32 K Word Program SRAM
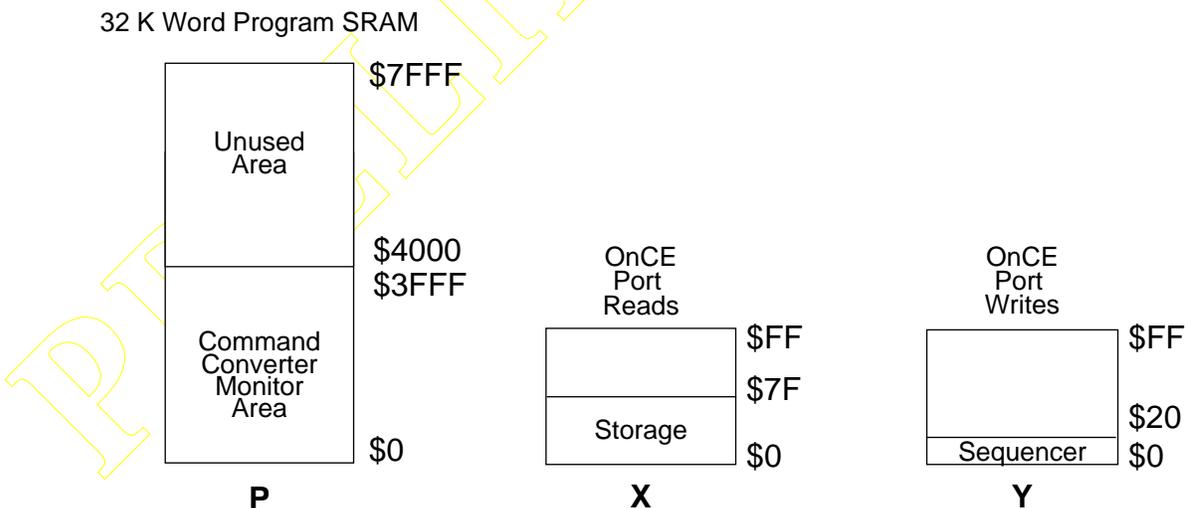
Figure 5-11 Command Converter Monitor Memory

5.7   JTAG/ONCE COMMUNICATIONS PERFORMANCE

The performance of the JTAG/OnCE link, and therefore some aspects of the ADS system performance, is determined by the clock speed of the SSI port on the Command Converter DSP used for the JTAG/OnCE link. Loading, saving and examining memory and other data transfer operations proceed faster with a faster SSI clock. However, the maximum permissible SSI clock speed is determined by the clock speed used on the target DSP processor, and must not exceed target DSP clock / 8. When multiple DSP processors are connected in a JTAG chain, the slowest target DSP clock must be considered.
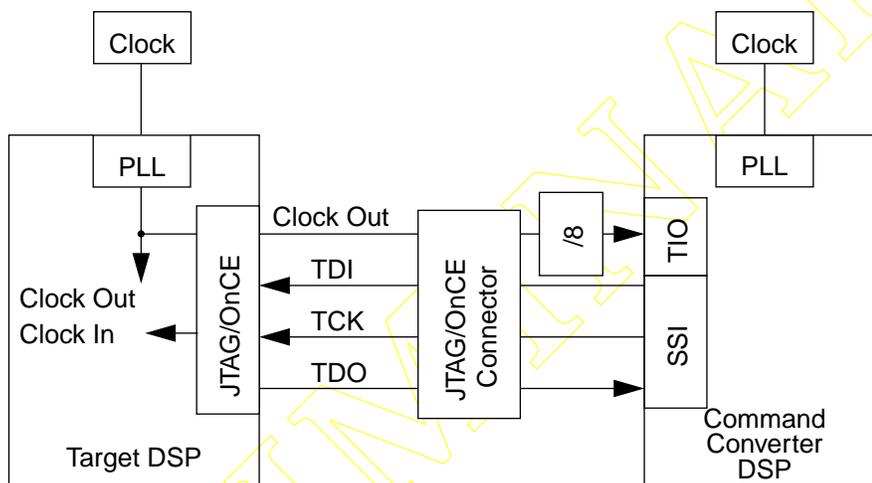
Command Converter, SSI and Target clocks must be set so that

Clock Out $\geq$ Clock In $\times$ 8

Figure 5-12 Command Converter / Target DSP Clock Constraints

The SSI port in the Command Converter DSP56002 used for the JTAG/OnCE serial link, is controlled by the value stored in Command Converter memory location X:$6, SSICLK. This value is used to initialize the SSI control register CRA on the Command Converter, and may be altered to control the SSI clock speed.

See the 56002 User Manual for full SSI details; briefly, CRA comprises several fields:

- Bits 0–7 Prescale Modulus (PM0–PM7)—0 = divide by 1 up to 255 = divide by 256.

- Bits 8–12 (DC0–DC4)—frame rate divider

- Bits 13–14 Word Length control (WL0–WL1)

- Bit 15 Prescale Range (PSR)—0 = divide by 1,1 = divide by 8

Only PM0–PM7 and PSR should be changed, all other fields should be left unchanged.

The value for PM0–PM7 that will give the fastest link with a target DSP is given by:

$$PM = (2 \times Fcc - 1) / Ftgt$$

where Fcc and Ftgt are the clock speeds of the Command Converter and target DSPs; use integer arithmetic; fractional results are to be truncated.
If PM exceeds 256, set PSR to 1 and divide PM by 256.

The PLL oscillator in the Command Converter DSP56002 (rev 6) is controlled by the value stored in Command Converter memory location X:$16, PLLVAR. This value is used to initialize the PLL control register PCTL and may be altered to select the operating speed of the Command Converter.

These locations may be changed with the change command, either manually or as part of a command file. See **Section 3** for more information.

For example, the following values could be used for PLLVAR and SSICLK:

- PLLVAR: $40003   20 MHz external clock $\times$ 4 = 80 MHz system clock

- SSICTL: $1   80 MHZ system clock $/$ 8 = 10 MHz SSI clock

After establishing communication with a target DSP device, it is possible to lose communication if the target system reduces its clock speed, possibly as a power-saving feature or for some other reason. To re-establish communication, the Command Converter SSI clock rate must be reduced to the range dictated by the clock in the target DSP.

5.8 COMMUNICATING WITH THE TARGET OnCE PORT

All OnCE debug port registers are accessed via an 8-bit command. The 8-bit command contains the register address, whether to read or write to that address, whether to execute the opcode/operand in the pipeline registers and whether to exit or stay in the Debug mode of operation.

There are two basic classes of registers in the OnCE port. Registers associated with port status and breakpoints will always be accessed with the intent to stay in the Debug mode and not to execute the opcode/operand in the pipeline registers. Registers associated with the pipeline are the only registers which are accessed when executing opcodes/operands and exiting the Debug mode of operation.

In order to retrieve or store information to target system memory, the Command Converter must load opcodes and operands into the pipeline registers and execute them while still in the Debug mode. Since the opcode and operand values may be loaded into the program instruction latch and program data bus latch, it is possible to execute one instruction and then re-enter the Debug mode of operation. If the opcode/operand is executed from the Debug mode the pipeline status registers will not be updated but if the opcode/operand is executed from the User mode the pipeline status registers will change.

All 8-bit commands will be followed by an acknowledge from the DSO pin. All write commands will also be followed by an acknowledge after the 32nd bit of the data has been shifted into the DSI/OS0 pin. A write to the PDB register with an exit to User mode will be the only case where an acknowledge will not be followed by a write command.

In order to insure that the target system will re-enter the User mode in the correct state, the host computer interface program will always store the pipeline register values when the target system enters the Debug mode of operation. When the user wishes to re- enter the User mode a go command is issued and the previous state of the machine is restored prior to exiting the Debug mode. If the user enters the Debug mode and wishes to change the program counter to evaluate code at a different address, a long jump instruction is entered into the pipeline registers so that the program controller may reload the pipeline. All data output on the target system is via a Global Data Bus Register. To read a memory location, a register is loaded with the address to read, the data is retrieved and put into another register and then that register value is moved to the global data bus register. The user then reads the value of Global Data Bus.

All registers used to move data should be saved prior to executing the commands and restored immediately after using them. Reading and writing new values to registers are accomplished using two word opcodes which write directly to the register or move the register directly to the Global Data Bus Register.

5.8.1 OnCE Command Format

OnCE 8-bit commands consist of five address select bits and three control bits as described in **Figure 5-13**. OnCE register addresses are slightly different on the 16-, 24-, and 32- bit DSPs. An example command sequence for reading a register value follows using the DSP56002 OnCE register addresses; OPDBR register is at address 9 and the OGDBR register is at address 8.

7	6	5	4	3	2	1	0
R/W	GO	EX	RS4	RS3	RS2	RS1	RS0

R/W—0 = Write Data to Register selected
     1 = Read Data from Register selected

GO—0 = Do not execute opcode in OPDBR
    1 = Execute opcode in OPDBR

EX—0 = Remain in the Debug mode
    1 = Exit the Debug mode to the User mode

RS4–RS0 = OnCE Register Select bits 4–0
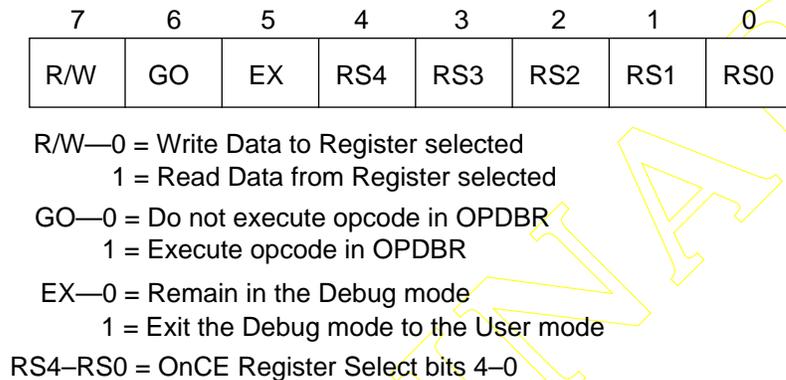
Figure 5-13 OnCE 8-Bit Command Format

5.8.2 OnCE Port Protocol

Two of the four OnCE port pins, Debug Serial Input/OnCE Status 0 (DSI/OS0) and Debug Serial Clock/OnCE Status 1 (DSCK/OS1), are initialized to outputs during hardware reset of the DSP. These pins provide information regarding whether the DSP is in STOP/WAIT mode, waiting for an external access to complete or whether executing normally from internal RAM. These pins should have pull down resistors so when the Debug mode is entered they will be in a deterministic state and not produce false clocks or data.

The host computer must provide all clocks to the DSCK/OS1 when reading or writing commands and data. All serial data is clocked in on the negative edge while data is clocked out on the positive edge of the DSCK clock input. The maximum rate of DSCK is 1/4 of the DSP internal clock rate.

When $\overline{DR}$ is asserted the DSP program controller completes the current instruction it is executing, stores the contents of the pipeline registers and freezes the DSP program controller. All program interrupt service requests are held in a pending state and the DSI/OS0 and DSCK/OS1 pins become inputs so data and commands may be transferred.

5.8.3 OnCE Debug Acknowledge Signal

Whenever the Debug mode is entered an acknowledge signal is transmitted out of the Debug Serial Output (DSO) pin. This signal provides a means for synchronizing the host computer to the OnCE controller. **Figure 5-14** illustrates the timing protocol for the DSP56100 16-bit architecture. The delay between the assertion of the $\overline{DR}$ and the transmission of the acknowledge signal can vary. The DSP must complete the current instruction being executed before entering the Debug mode. Therefore, if an external bus access is in progress and the DSP Bus Control Register is greater than zero, the delay will be dependent upon the number of internally generated wait states.

Also, if a bus arbitration unit currently does not permit the DSP to complete an external access the Debug mode cannot be entered until the arbiter permits the access to complete. This same rule holds for the Transfer Acknowledge ($\overline{TA}$) signal of the DSP. If the TA signal is not asserted the DSP will not enter the Debug mode until the bus access is complete. The debug acknowledge signal is transmitted immediately after each 8-bit command, and after the proper number of clocks required to write to a register. Also, if an execute opcode command is given, the acknowledge signal will be asserted after the DSP program controller completes execution of the opcode in the OPDBR.
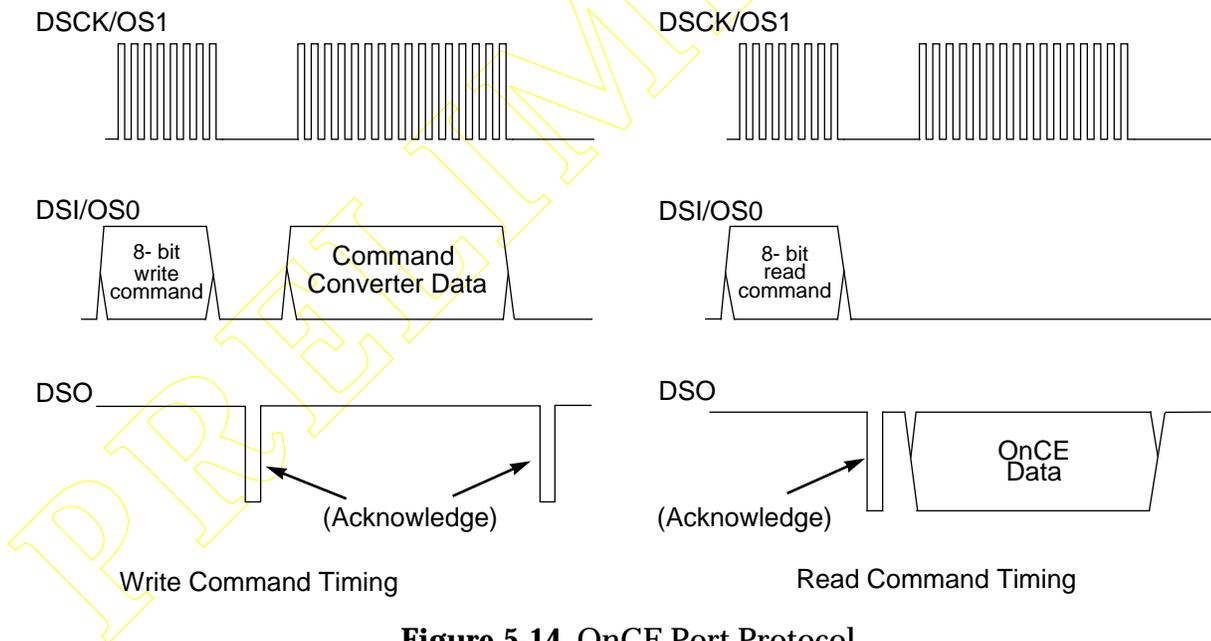
Figure 5-14 OnCE Port Protocol

It should be noted that the DSO normally is in a high impedance state and only drives a signal output when executing an 8-bit write command or when sending the debug acknowledge signal. This signal normally has a pull-up resister.

5.9   WRITING YOUR OWN OnCE COMMAND SEQUENCE

The ADS user interface program has eight special commands that allow the user to develop their own OnCE serial sequences. These commands allow the user to CLOAD, CSAVE, CCHANGE, CDISPLAY, CTRACE, CGO, CFORCE, or CSTEP through OnCE serial sequences. Users have total control over the OnCE port. They may wish to develop their own OnCE command sequences which they use in a system. The Command Converter monitor and host computer user interface allows for development of these sequences.

Sequences consist of OnCE commands and data as described in the DSP user manual. Sequences may read or write values to the OnCE registers in order to extract data from the target. Sequences use a defined format so that the Command Converter monitor may know when data is to be read or written over the OnCE port. Data that is read from the OnCE port is stored in the Command Converter X memory specified by a pointer in the X memory location 4 of the DSP56002 controller. Data values that are read are stored in the upper 16 bits of the X memory 24-bit words.

The 8-bit OnCE commands are stored in the upper byte of the Y memory 24-bit words with the lower byte value giving the appropriate action necessary to complete the 8-bit OnCE command. For example, when a OnCE read command is issued, an acknowledge signal must be received to stay synchronized, then the proper number of clocks must be issued by the controller to receive the data from the target OnCE port. To accomplish this a defined set of actions will occur when a sequence byte is in the lower byte of the command word of the Y memory.

Table 5-1 describes the actions associated with each sequence byte type:

Table 5-1   OnCE Sequence Control Codes

Sequence code	Action taken by Command Converter Monitor
0	No Action
1	End of sequence—return to monitor
2	Send command, wait for ack, clock in data into X memory.
4	Send command, wait for ack, clock out data, wait for ack.
8	Send command, wait for ack
10	Send command, wait for ack, clock out data

An example Command Converter Y memory command sequence follows:

```
800002      ;read OSCR register and store in X memory.
000004      ;write OSCR register
123400      ;value of $1234
090004      ;write PDB no go, no ex.
3A1400      ;move r0,x:OGDB - opcode
490004      ;write PDB go, no ex.
FFFF00      ;$FFFF - operand -
880002      ;read OGDB no go, no ex. and store in X mem.+1
000001      ;end of sequence
```

5.10   COMMUNICATING WITH THE TARGET JTAG PORT

Devices which have an IEEE JTAG 5-wire port communicate with the OnCE port via the IEEE JTAG protocol. A special user defined JTAG command, ENABLE_ONCE, is executed through the JTAG state machine in order to pass serial commands and data to and from the OnCE controller. The OnCE concept of communicating with registers via an 8-bit command continues to hold true. There have been minimal changes to the OnCE controller and logic. The major difference is the communication protocol has been changed to adhere to the JTAG protocol.

5.11   CHANGES TO THE OnCE PORT PINS

OnCE port pins have been converted to JTAG pins on newer devices. The DSI/OS0 i/o pin has become the TDI input pin. This pin adheres to the JTAG standard and therefore no longer provides status information. The DSCK/OS1 pin has become the TCK input pin. Like TDI, this pin will no longer provide status information. The DSO output pin has become the TDO output pin. This pin no longer provides the acknowledge pulse since it must adhere to the JTAG standard. The DRZ input pin has become the TMS input pin. This pin no longer is used as an external debug request mechanism.
Figure 5-15 is a block diagram of the IEEE 1149.1-1990 test logic coupled to the OnCE logic TAP Controller.
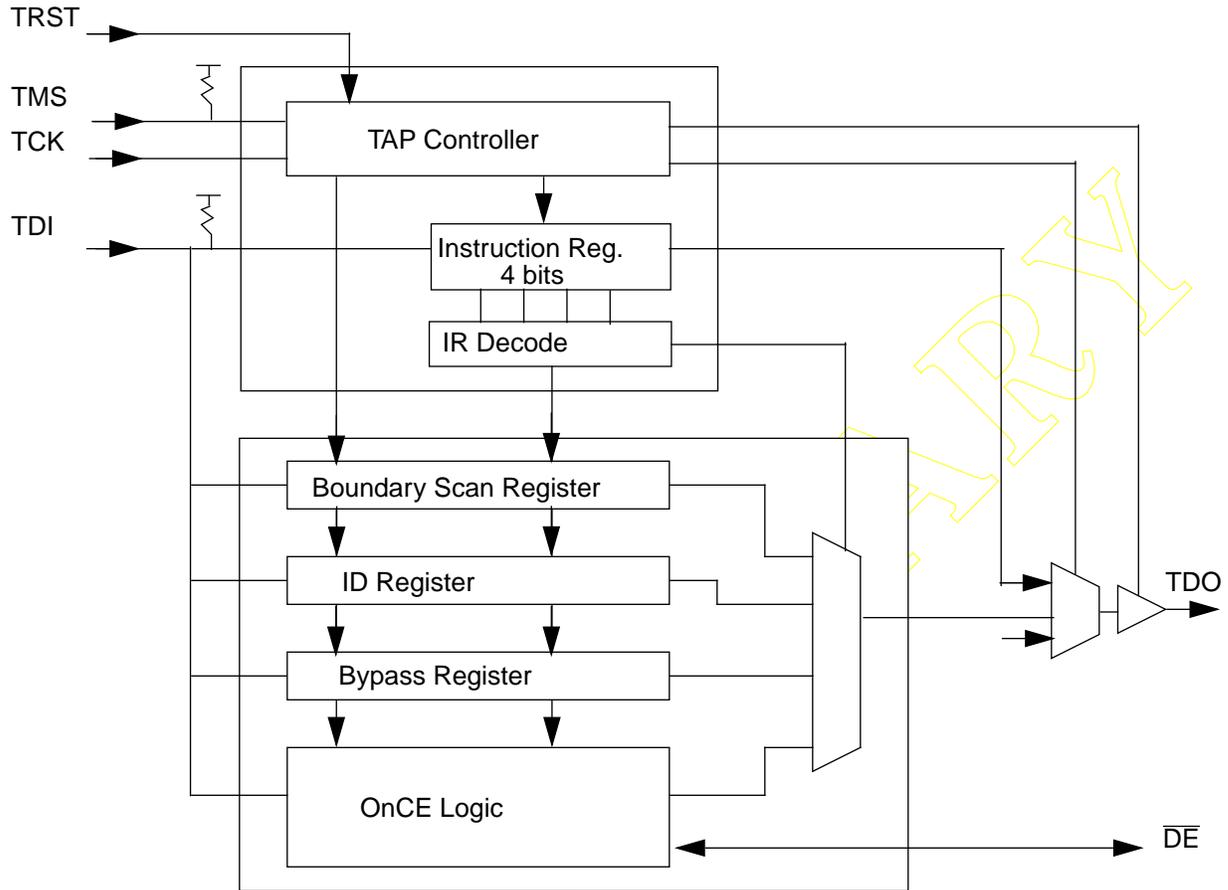
Figure 5-15 JTAG/OnCE Interface

A new bidirectional pin called the Debug Event ($\overline{DE}$) has been added. This pin provides an open drain output signal which indicates an event has occurred in the OnCE debug logic. This event can be an entry to Debug mode, a trace count decrement to zero or a vectored interrupt taken due to one of the above.

The $\overline{DE}$ pin also provides an input function which acts as the debug request signal used to halt the DSP core. The main advantage of this pin is in debugging multiple DSP applications.

The TAP controller is a synchronous finite state machine that contains sixteen states as illustrated in **Figure 5-16** on page 5-26. The TAP controller responds to changes at the TMS and TCK signals. Transitions from one state to another occur on the rising edge of TCK. The value shown adjacent to each state transition in this figure represents the signal present at TMS at the time of a rising edge at TCK.
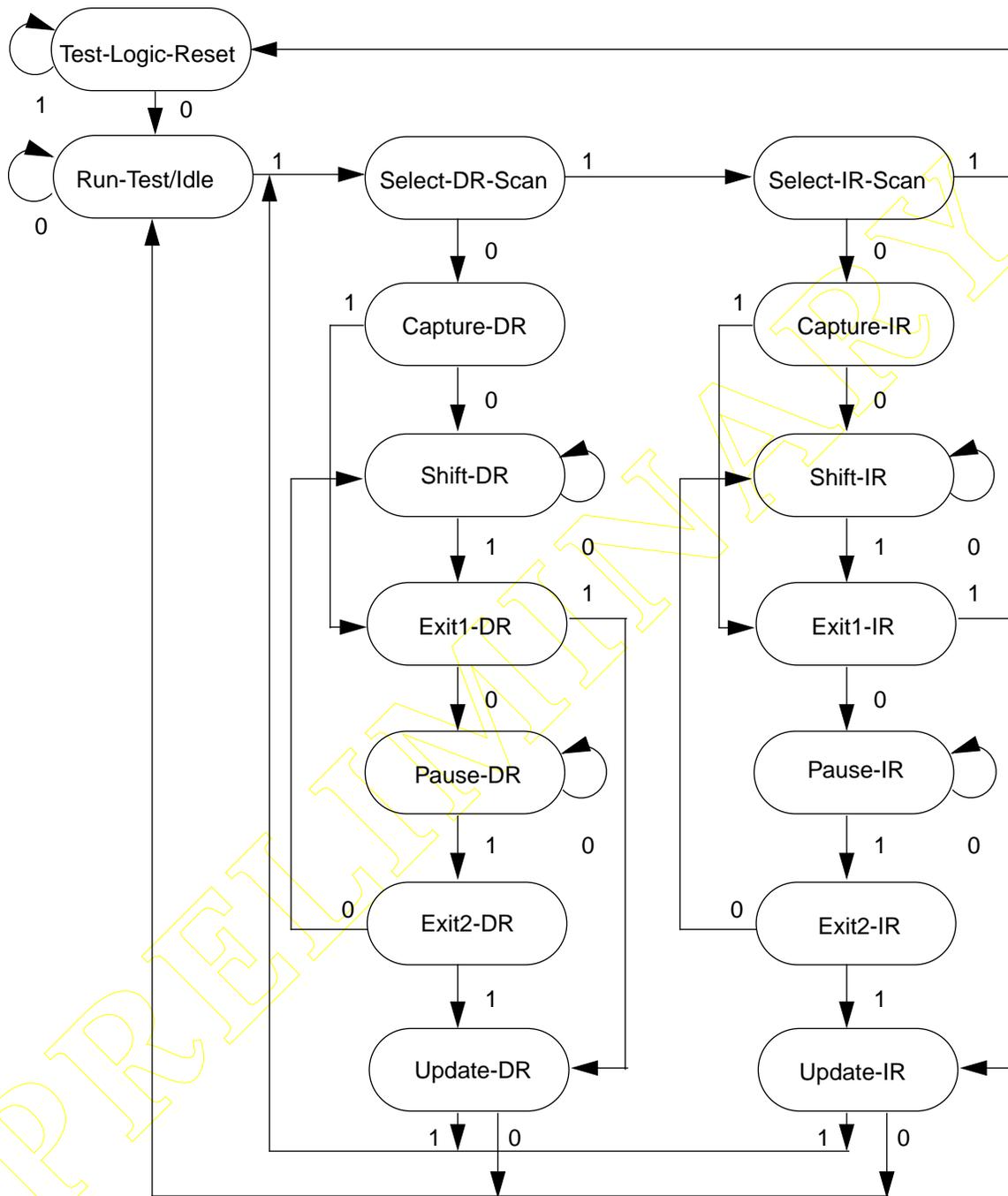
Figure 5-**16** TAP Controller State Diagram

The TDO pin remains in the high impedance state except during the Shift-DR or Shift-IR controller states. In these controller states, TDO will update on the falling edge of TCK. TDI is sampled on the rising edge of TCK.

There are two paths to the 16-state machine. The SHIFT-IR_SCAN path is used to capture and load JTAG instructions into the instruction register. The SHIFT-DR_SCAN path is used to capture and load data into the test data registers. The TAP controller will execute the last instruction decoded until a new instruction is entered at the Update-IR state or until the Test-Logic-Reset state is entered. All communication with the OnCE port is via the Select-DR-Scan path after the TAP controller has decoded an ENABLE_ONCE command.

5.12   JTAG INSTRUCTION REGISTER

The DSP IEEE 1149.1-1990 implementation includes the three mandatory public instructions (BYPASS, SAMPLE/PRELOAD, and EXTEST) and four public instructions (CLAMP, HIGHZ, IDCODE, ENABLE_OnCE). The TAP controller contains a four bit instruction register. The instruction is presented to an instruction decoder during the Update-IR state. **Table 5-2** illustrates the four bits (B3–B0) used to decode sixteen instructions.

Table 5-2   JTAG Instruction Register Encoding

B3	B2	B1	B0	Instruction
0	0	0	0	EXTEST
0	0	0	1	SAMPLE/PRELOAD
0	0	1	0	IDCODE
0	0	1	1	RESERVED
0	1	0	0	HIGHZ
0	1	0	1	CLAMP and BYPASS
0	1	1	0	ENABLE_OnCE
0	1	1	1	DEBUG_REQUEST
1	X	X	X	BYPASS

All other encodings are reserved for future enhancements and will be decoded as BYPASS. The Instruction Register is reset to 0010 in the Test-Logic-Reset controller state. Therefore, the IDCODE instruction is selected on JTAG reset. In the Capture-IR state the two Least Significant Bits of the Instruction Shift Register will be preset to 01 where the 1 is in the Least Significant Bit location as required by the standard. The two Most Significant Bits may either capture status or be set to 0. New instructions are shifted into the Instruction Shift Register stage on Shift-IR state.

5.12.1 ENABLE_OnCE (0110)

The ENABLE_OnCE instruction enables the JTAG port to communicate with the OnCE state machine and registers. It is provided as a Motorola public instruction to allow the user to perform system debug functions. When the ENABLE_OnCE instruction is invoked, the TDI and TDO pins will be connected directly to the OnCE registers. The particular OnCE register connected between TDI and TDO is selected by the OnCE state machine and the OnCE instruction being executed. All communication with the OnCE instruction controller is done through the SELECT-DR-SCAN path of the JTAG state machine.

5.12.2 DEBUG_REQUEST (0111)

The DEBUG_REQUEST instruction asserts a request to halt the core for entry to Debug mode. It is typically used in conjunction with ENABLE_ONCE to perform system debug functions. It is provided as a Motorola public instruction. When the DEBUG_REQUEST instruction is invoked, the TDI and TDO pins will be connected to the BYPASS register.

5.12.3 Polling for Chip Status From the JTAG Port

Two DSP core status bits are accessible by reading the OnCE Status/Control Register or when a JTAG instruction is entered in the SHIFT-IR state. **Table 5-3** describes the two status bits of the DSP core. To insure synchronization of an external JTAG controller and the target DSP, the status bits should be polled after entering commands which do external memory accesses. This will insure that external accesses with wait states or bus arbitration will terminate correctly before trying to enter any new commands associated with executing DSP opcodes thought the OPDBR.

Table 5-3 DSP Core Status Bit Description

Function	OS1	OS0	Comment
Normal	0	0	DSP core executing Instructions
Stop/Wait	0	1	DSP core in Stop or Wait Mode
Busy	1	0	DSP doing external or peripheral access
Debug	1	1	DSP core halted

SECTION 6

HOST COMPUTER CARD/COMMAND CONVERTER SUPPORT INFORMATION

6.1 INTRODUCTION

This chapter provides the connector signal descriptions and parts lists for hardware that is required to run with the ADS software. This list includes the host interface cards, host interface cable and the Command Converter card.

6.2 HOST INTERFACE CARD BUS SIGNAL DESCRIPTION

Each host interface card is designed for a unique host computer bus architecture. This section gives each card edge connector description which is plugged into the host computer expansion bus. This information is for reference only.

Table 6-1 PC Interface Card J2 (ISA16 Bus) Connector

Pin #	Mnemonic	Signal Name and Description
A1	$\overline{\text{I/O CH CK}}$	No Connect
A2–A9	D7–D0	PC Bus data bits 7 to 0.
A10	$\overline{\text{I/O CH RDY}}$	No Connect
A11	AEN	PC BUS Address Enable output
A12–A21	A19–A10	No Connect
A22–A31	A9–A0	PC Bus address bits 9 to 0.
B1	GND	PC ground
B2	RESET	PC reset signal,(positive true).
B3	+5v	PC +5 V
B4-B8		No Connect
B9		PC +12 V
B10	GND	PC ground
B11–B12	$\overline{\text{MEMW}}/\overline{\text{MEMR}}$	No Connect
B13	$\overline{\text{IOW}}$	I/O write command,(negative true).
B14	$\overline{\text{IOR}}$	I/O read command,(negative true).
B15–B20		No Connect
B21	IRQ7	PC Interrupt request 7,(Printer), NOT USED.
B22–B23		No Connect
B24	IRQ4	PC Interrupt request 4,(COM1), NOT USED
B25	IRQ3	PC Interrupt request 3,(COM2),
B26–B31		No Connect

Table 6-**2**   Sun 4 SPARC (SBus) Connector

Pin #	Description	Pin #	Description	Pin #	Description
1	GND	33	PA(06)	65	D(18)
2	$\overline{BR}$	34	PA(08)	66	D(20)
3	$\overline{SEL}$	35	PA(10)	67	D(22)
4	$\overline{INTREQ1}$	36	$\overline{ACK0}$	68	GND
5	D(00)	37	PA(12)	69	D(24)
6	D(02)	38	PA(14)	70	D(26)
7	D(04)	39	PA(16)	71	D(28)
8	$\overline{INTREQ2}$	40	$\overline{ACK1}$	72	+5V
9	D(06)	41	PA(18)	73	D(30)
10	D(08)	42	PA(20)	74	SIZ(1)
11	D(10)	43	PA(22)	75	RD
12	INTREQ(3)	44	$\overline{ACK2}$	76	GND
13	D(12)	45	PA(24)	77	PA(01
14	D(14)	46	PA(26)	78	PA(03)
15	D(16)	47	DTAPAR	79	PA(05)
16	$\overline{INTREQ4}$	48	-12V	80	+5V
17	D(19)	49	CLK	81	PA(07)
18	D(21)	50	$\overline{BG}$	82	PA(09)
19	D(23)	51	$\overline{AS}$	83	PA(11)
20	$\overline{INTREQ5}$	52	GND	84	GND
21	D(25)	53	D(01)	85	PA(13)
22	D(27)	54	D(03)	86	PA(15)
23	D(29)	55	D(05)	87	PA(17)
24	$\overline{INTREQ6}$	56	+5V	88	+5V
25	D(31)	57	D(07)	89	PA(19)
26	SIZ(0)	58	D(09)	90	PA(21)
27	SIZ(2)	59	D(11)	91	PA(23)
28	$\overline{INTREQ7}$	60	GND	92	GND
29	PA(00)	61	D(13)	93	PA(25)
30	PA(02)	62	D(15)	94	PA(27)
31	PA(04)	63	D(17)	95	$\overline{RESET}$
32	$\overline{LERR}$	64	+5V	96	+12V

6.3 HOST COMPUTER INTERFACE CABLE

The 37-pin cable which hooks to the host computer is called the host computer interface. This cable provides the signals and power to the Command Converters.

Table 6-3   Host Computer Interface Cable

Pin #	Mnemonic	Signal Name and Description
1	INT_ACK	Host ack of ADM service request
2	ADM_GROUP	CC group control flag from Host
3	HOST_ACK	Host ack of ADM data transfer request
4	ADM_ALL	Host signal which selects all ADMs
5	ADM_RESET	Host signal which asserts CC(s) reset
6-8	ADM_SEL2,1,0	CC address select signals 2-0 for one of 8 CCs
9	HOST_REQ	Host signal which requests CC data
10	ADM_REQ	CC signal which requests Host data transfer
11	ADM_ACK	CC ack of Host data transfer request
12	ADM_INT	CC service request status signal to Host
13	$\overline{\text{HOST\_BRK}}$	CC service request signal to Host,(low true)
14	ADM_BRK	Host signal to interrupt CC(s)
15	BRACE_SEL	Brace56 Emulator select signal
16-19	PD1,3,5,7	HOST/CC data bus bits1, 3, 5, 7
20-25	GND	HOST/CC ground lines
26	+12v	+12 volts from the HOST
27-29	+5v	+5 volts from the HOST
30	HOST_ENABLE	HOST signal which enables CC address logic
31-33	GND	HOST/CC ground lines
34-37	PD0,2,4,6	HOST/CC data bus bits 0, 2, 4, 6

6.4   JTAG/OnCE INTERFACE CABLE

The Command Converter 14-pin connector is hooked to the target system through the JTAG/OnCE interface cable. These signals provide the control signals to as many as twenty-four target DSP or other JTAG devices.

Table 6-4   JTAG/OnCE Connector J3

Pin #	Signal	Signal Description
1	TDI/DSI	Target JTAG/OnCE Serial Input
2	GND	Ground
3	TDO/DSO	Target JTAG/OnCE Serial Output
4	GND	Ground
5	TCK/DSCK	Target JTAG/OnCE Serial Clock
6	GND	Ground
7	$\overline{\text{DR}}$	Target OnCE Debug Request Input
8	No Connect	Used as Key
9	$\overline{\text{CC\_RESET}}$	Target DSP Reset Input
10	TMS0	Target JTAG Test Mode Select 0 input
11	Vdd	Target Vdd—Supplies OnCE Buffer (HC367)
12	TMS1	Optional Target Test Mode Select 1 (not required)
13	$\overline{\text{DEZ}}$	Target JTAG/OnCE Debug Event (input/output)
14	$\overline{\text{TRST}}$	Target JTAG Reset Input
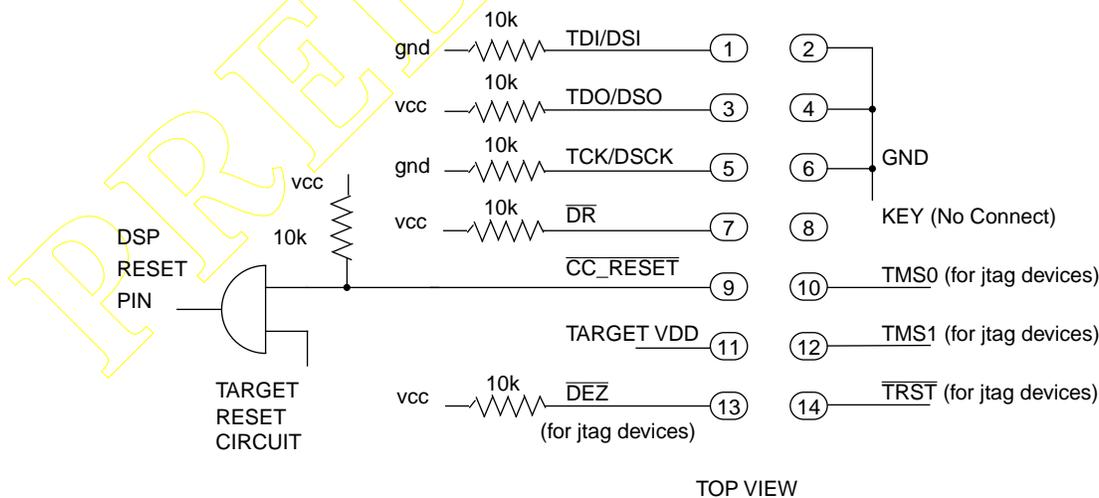
Figure 6-1 Command Converter Interface

Note: This is a plug connector that has all odd numbers on the left side and all even numbers on the right side when viewing from top. Pin 1 is on upper left side. Spacing between pins is 1/10th of an inch. Refer to Command Converter J3 connector as an example.

6.5   HOST COMPUTER CARD BILLS OF MATERIALS

This section contains the bill of materials for each of the host computer interface printed circuit board assemblies in the Application Development System.

Table 6-5   ADS PC-Compatible Interface Electrical Parts List Rev 2.01—06/06/96

Qty	Description	Ref. Designator	Vendor Part #
	Integrated Circuits		
1	MC74LS04D	U1	Motorola
2	MC74F521DW	U2,U3	Motorola
1	MC74F138D	U4	Motorola
2	MC74F32D	U5,U6	Motorola
1	MC74LS00D	U7	Motorola
1	MC74F245DW	U8	Motorola
2	SN74ALS373DW	U9,U11	Texas Instrument
2	SN74ALS575DW	U10,U13	Texas Instrument
1	SN74ALS374DW	U12	Texas Instrument
1	MC74F125D	U14	Motorola
	Resistors		
1	100 KΩ	R1	Bourns CR1206-1003-FVCA
1	10 KΩ	R2	Bourns CR1206-1002-FVCA
	Resistor Networks		
1	10 KΩ—8 resistor pack-common pullup	RN1	Bourns 4610X-101-103
1	22 Ω—8 series resistors	RN2	Brady 4816-P-001-220
1	22 Ω—10 series resistors	RN3	Brady 4820-P-001-220
	Capacitors		
15	0.1 μF	C1–C15	Murata GRM42-6COG104K050BL
2	100 μF	C16,17	Sprague 501D107M6R3LL
Note: Resistors are 5% 1/4 w carbon (unless otherwise specified).			

Table 6-6 ADS PC-compatible Interface Hardware Parts List Rev 2.01—06/06/96

Qty.	Location	Description	Vendor Part #
		Jumpers.	
1	JG1	5 row × 2 berg stick	R.N. NSH-10DB-S2-TG30
1	JG2	8 row × 2 berg stick	R.N. NSH-16DB-S2-TG30
		Connectors	
1	J2	AMP 37-pin SUB-D Connector	AMP 745097-1
		Miscellaneous	
1		PC Bracket	Olsen 9007001
2		4-40 3/8 Screws	
2		4-40 3/8 Nuts	
1		AMP Female Screwlocks	AMP 205817-3
2		Molex jumpers	Molex 15-29-1024

Table 6-7 37-Conductor Cable Assembly List Rev 2.0 - 11/01/95

Qty.	Description	Vendor Part #
2	AMP Mating Connector	AMP# 747319-1
1	37-Conductor Ribbon Cable 4 ft.	T&B/Ansley
	28AWG,stranded, .050 inch pitch	#171-37

Table 6-8 Sun-4 SBus Parts List Rev. 01 May 27,1992

Qty	Description	Ref. Designator	Vendor
2	74ACT138	U10,12	Motorola
4	74ACT244	U4,7,8,9	Motorola
1	74ACT245	U11	Motorola
1	74ACT02	U6	Motorola
1	74ACT08	U5	Motorola
1	74ACT32	U2	Motorola
1	74ACT04	U1	Motorola
1	74F374	U17	Motorola
1	74F273	U18	Motorola
2	74ACT273	U13,16	Motorola
1	74F373	U19	Motorola
2	74ACT373	U14,20	Motorola

Table 6-8 Sun-4 SBus Parts List Rev. 01 May 27,1992 (Continued)

Qty	Description	Ref. Designator	Vendor
1	PALCE22V10H-15PC/4	U3	AMD
1	WS57C291B-35T	U15	WSI
2	R pack 8×2 22 Ω	RN1,2	Bourns 4116R-001-RC
1	100 k Ω	R1	Newark 10F305
20	0.01 μf Ceramic	C22	Kemet C322C104M5R5CA
1	10 μf Electrolythic Axial	C22	Sprague 501D106M063LL
1	100 μf Electrolythic Axial	C21	Sprague 501D107M010LM
2	24 Pin IC socket	U3,15	R.N. ICE-243-S-TG30
1	SBUS Male connector	J1	Fujitsu FCN-234P096-GO
1	37-pin D connector	J2	Amphenol 617-C037P-AJ221

6.6   COMMAND CONVERTER BILL OF MATERIALS

Table 6-9 ADS Command Converter Electrical Parts List Rev. 2.01—06/06/96

Qty	Description	Ref. Designator	Vendor Part #
Integrated Circuits			
1	DSP56002PV	U1	Motorola
1	MC1455D	U2	Motorola
1	A53-20MHz	U3	Connor Winfield
3	MC74HC244DW	U4,6,13	Motorola
1	MC74F245DW	U5	Motorola
1	MC74F373DW	U7	Motorola
3	MCM6206DP12	U8,9,10	Motorola
1	PALCE20V8Q15PC	U11	AMD
1	MAX232CWE	U12	Maxim
1	MC74ACT161D	U14	Motorola
Resistors			
1	3 KΩ	R1	Bourns CR12060302JVCA
28	10 KΩ	R2,3,4,6,9,11,12,13,14,15,16, 18,19,22,23,25,29,30,31,32,3 3,34,35,36,39,41,42,43	Bourns CR12061002JVCA
2	1.8 KΩ	R5,26	Bourns CR12060182JVCA
4	1 KΩ	R7,8,24,27	Bourns CR12060102JVCA
1	1 MΩ	R10	Bourns CR12061004JVCA
4	330 Ω	R17,20,28,38	Bourns CR12060330JVCA
2	100 Ω	R21,37	Bourns CR12060100JVCA
1	500 Ω	R40	Bourns CR12060500JVCA
Resistor Networks			
3	22 Ω	RN1–3	Bourns 4610-102-220
1	54 Ω	RN4	Bourns 4610-102-540
Capacitors			
17	0.1 μF	C1-6, 8, 15, 18, 20, 21–23, 25, 26	Kemet C315C104M5U5CA
23	.1 μF	C1–6, 10, 15–19, 21, 23, 25, 27, 29, 31, 33, 36, 37, 38, 39	Murata Erie GRM42-6X7R104K025BB
1	.47 μF	C7	Panasonic ECS-F1HE-474

Table 6-9   ADS Command Converter Electrical Parts List Rev. 2.01—06/06/96

Qty	Description	Ref. Designator	Vendor Part #
1	.1 µF	C8	Panasonic ECS-F1VE-104
1	390 pF	C9	Murata Erie GRM42-6X7R301K050BB
2	10 µF	C11,12	Panasonic ECS-F0JE-106K
2	4.7 µF	C13,14	Panasonic ECS-F0JE-475K
7	.01 µF	C20, 22, 24, 26, 28, 30, 32	Murata Erie GRM42-6X7R103K025BB
1	1.0 µF	C34	Panasonic ECS-F1CE-105K
1	10 µF	C35	Sprague 501D107M010LM
Diodes			
1	1N5711	D1	Hewlett Packard
Crystal			
1	27 MHz	X1	International Crystal #436161-27.00., Abricon #AB-27.00MHZ-10 Fundamental Frequency At - Cut Crystal
Transistors			
3	2N3904	Q1–3	Motorola
Fuse			
1	3.0A POLY	F1	Raychem RUE300
Light Emitting Diode			
1	Green SMT	LED1	Hewlett Packard HSMG-T400

Note: 1.   X1 must be parallel resonant, 10 pF load, fundamental frequency.
2.   Mylar spacer goes under crystal to insulate from PCB.

Command Converter Bill of Materials

Table 6-10 ADS Command Converter Hardware Parts List Rev. 2.01—06/06/96

Qty	Description	Ref. Designator	Vendor Part #
Jumpers			
2	1 × 2 Bergstik	JG1,4	R.N. NSH-02SB-S2-TG30
1	2 × 3 Bergstik	JG2	R.N. NSH-06DB-S2-TG30
1	1 × 3 Bergstik	JG3	R.N. NSH-03SB-S2-TG30
Connectors			
1	1 × 2 Terminal Block	P1	Augat NC6-P102-02
1	37-Pin Connector	J1	Amphenol 617C037PAJ221
1	1 × 4 Bergstik	J2	R.N. NSH-04SB-S2-TG30
1	2 × 7 Bergstik	J3	R.N. NSH-14DB-S2-TG30
1	2 × 5 Bergstik	J4	R.N. NSH-10DB-S2-TG30
Sockets			
1	24-PIN DIP Socket	U11	R.N. ICE-283-S-TG
Switches			
1	SPST MOM	SW 1	S1
Miscellaneous			
4	Ruber Feet		Amatom #5186
4	3/ 4" Nylon Standoffs		HH Smith
4	4-40 x 1/4" Nylon Screws		Waldon

Table 6-11 JTAG/OnCE 14-Pin Cable Assembly

Qty	Description	Vendor Part #
2	IDC Receptacle Connector	Dupont #66432-014
1	12" Flat Ribbon 14 Pin Cable	Dupont #76825-014
1	Polarization Plug	Dupont #65762-001

6.7   HOST INTERFACE CARD SCHEMATICS

ADM INTERFACE CABLE

Ribbon Cable
24 AWG Stranded
0.1" Conductor Spacing.

37 pin mass terminating
receptacle connector with
strain reliefs (2 places).

4 ft.

MOTOROLA
INC.

FABRICATION DETAIL.

ADM INTERFACE CABLE 1.1

PAGE 1 OF 1

REV. 1.0
1/15/93

NOTES:
1. Pin 1 orientation from connector to connector must be
consistent.
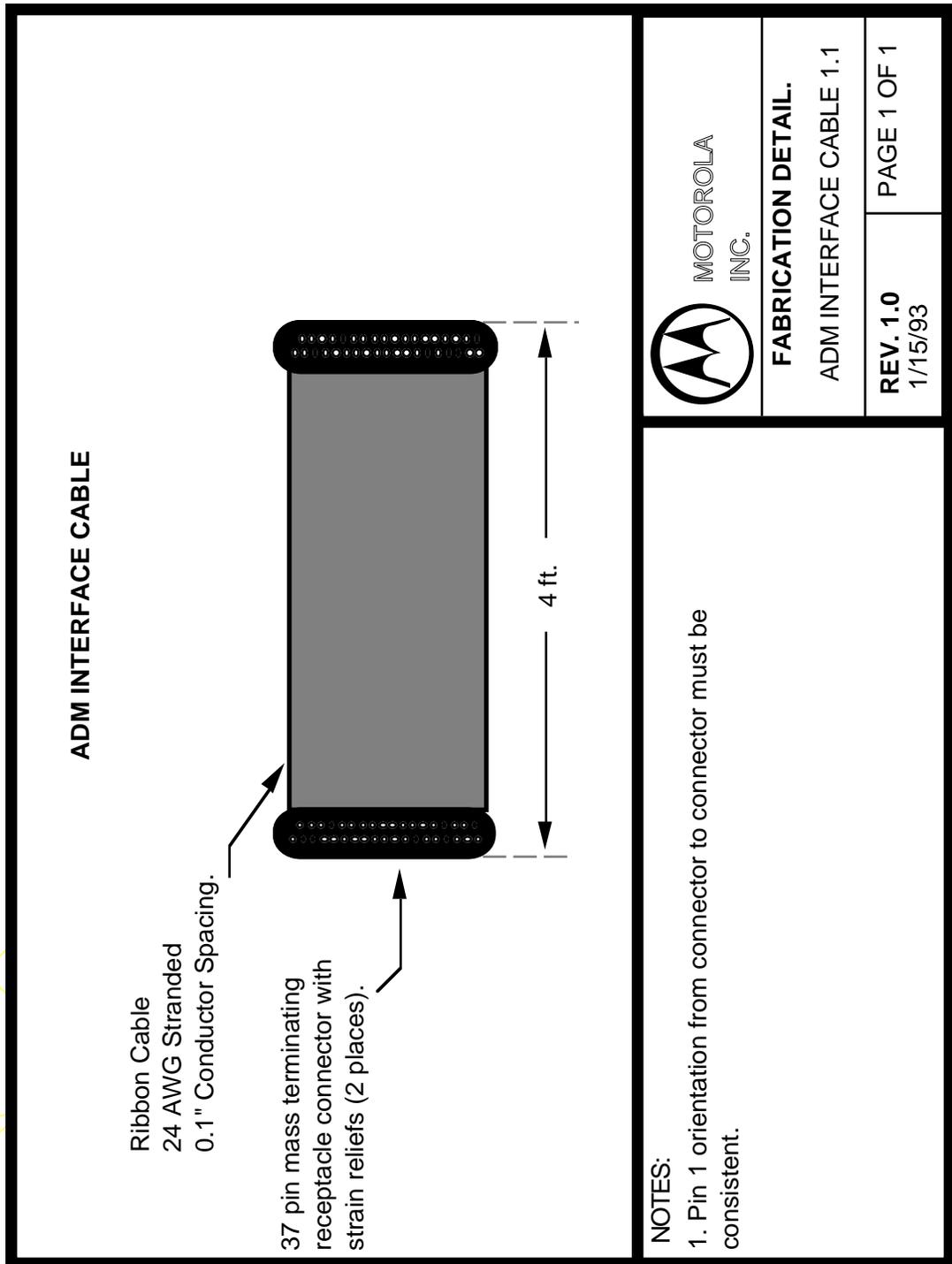
Figure 6-2 37-Pin Host Interface Cable
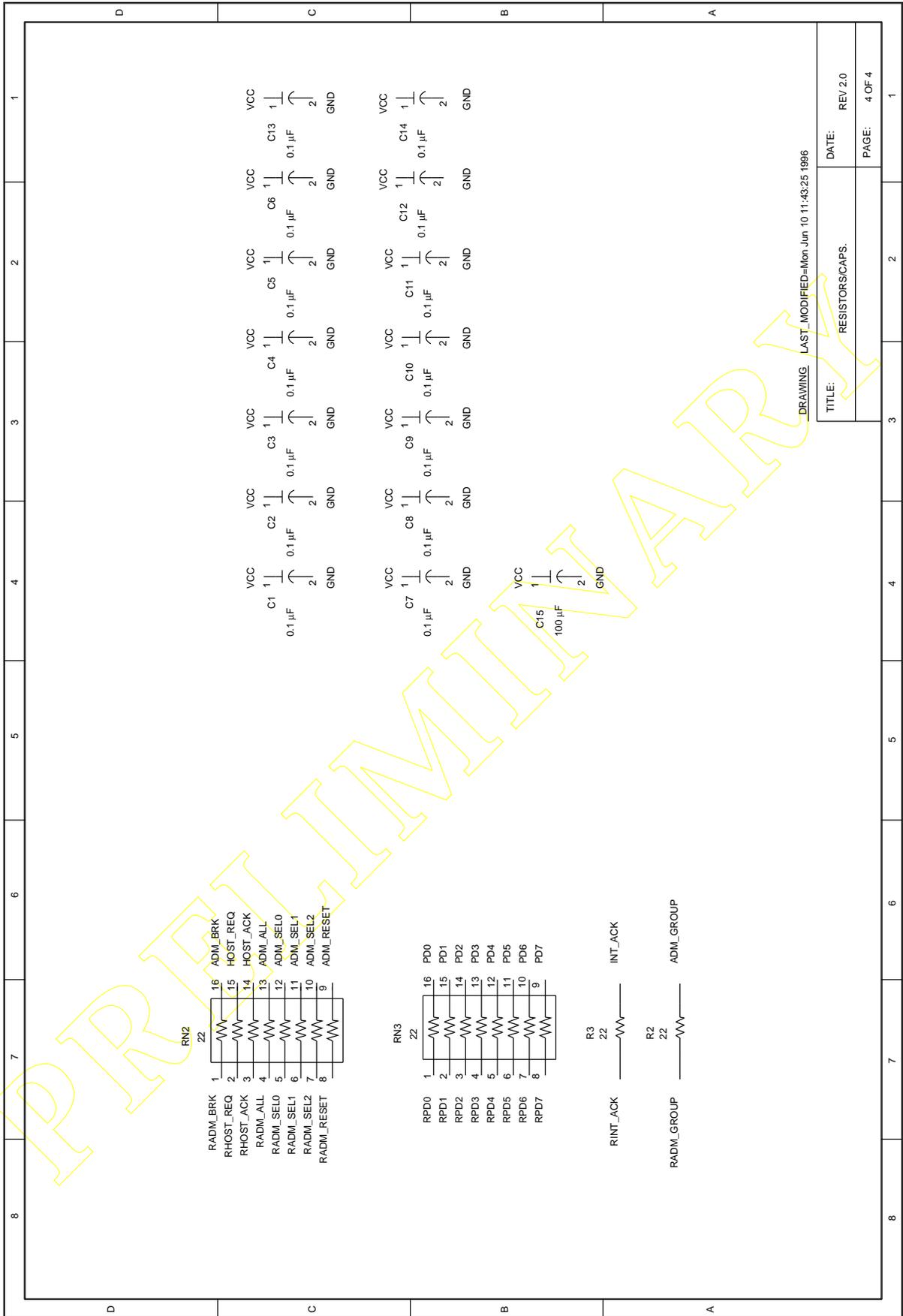
Figure 6-3 PC-Compatible Interface Card Rev. 2.0 (Page 1 of 4)
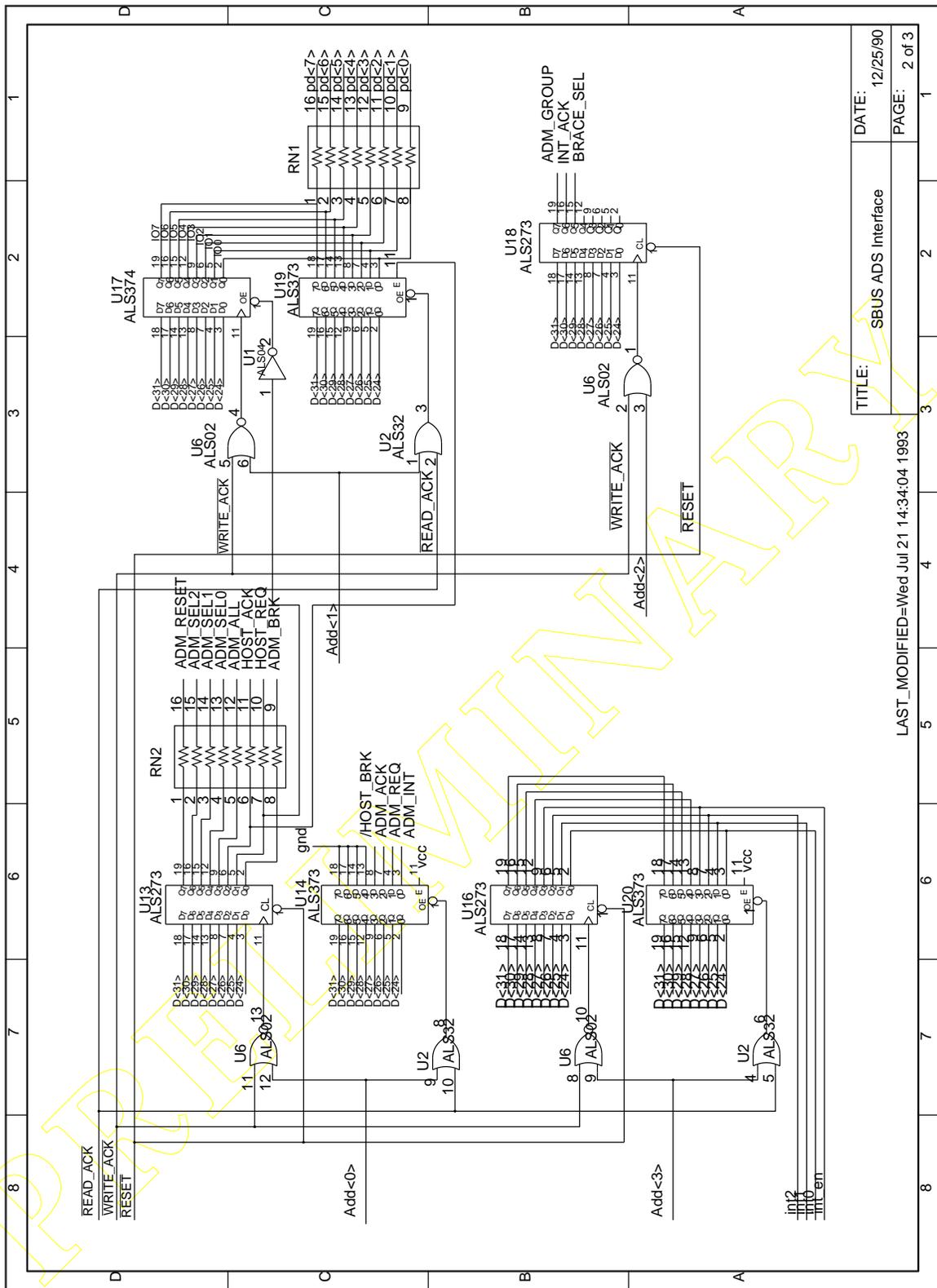
Figure 6-3 PC-Compatible Interface Card Rev. 2.0 (Page 2 of 4)

Figure 6-3 PC-Compatible Interface Card Rev. 2.0 (Page 3 of 4)

Figure 6-3 PC-Compatible Interface Card Rev. 2.0 (Page 4 of 4)
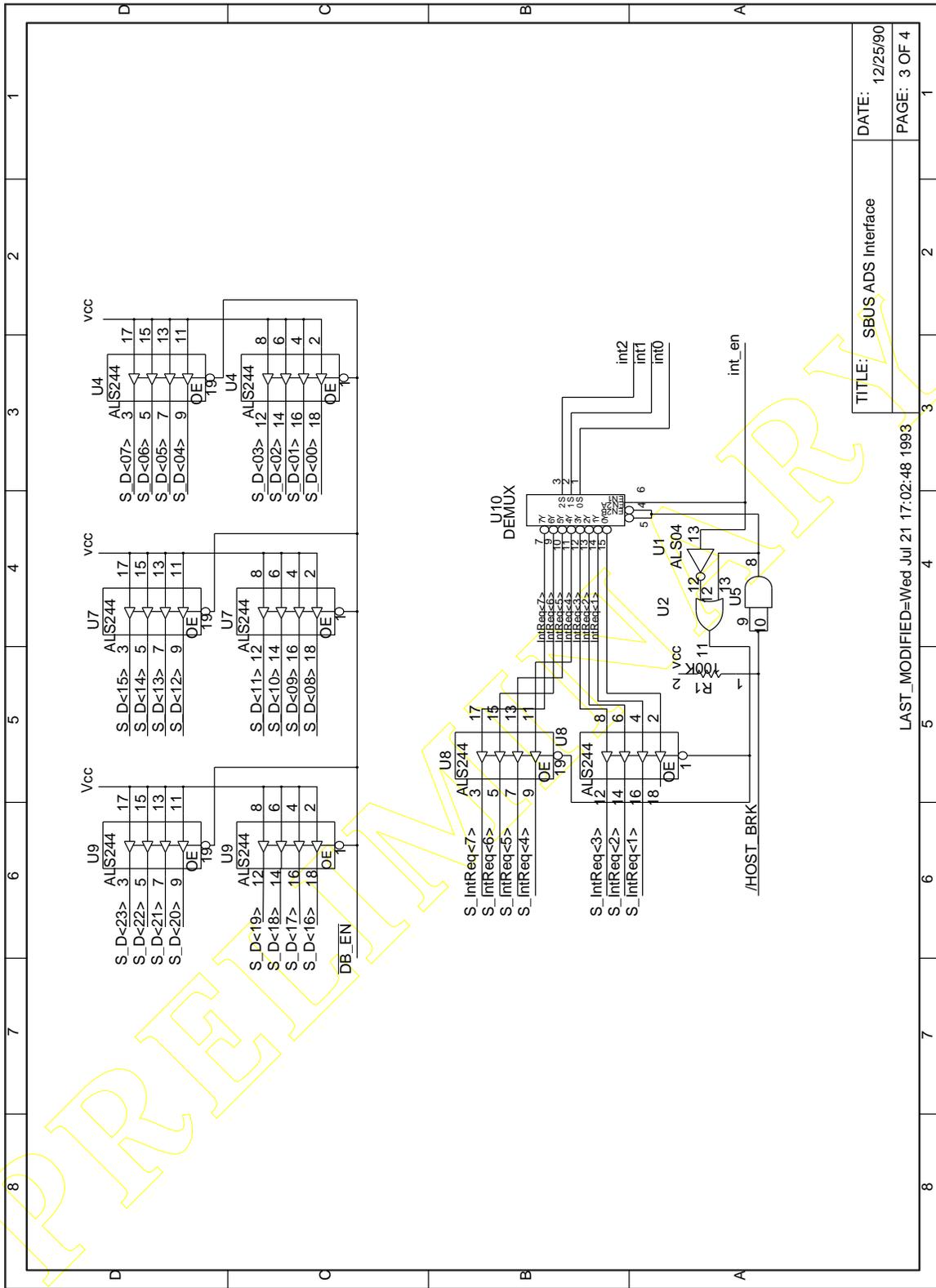
Figure 6-**4** Sun Sparc SBus Interface Card (Page 1 of 4)

Figure 6-4 Sun Sparc SBus Interface Card (Page 2 of 4)

Figure 6-4 Sun Sparc SBus Interface Card (Page 3 of 4)
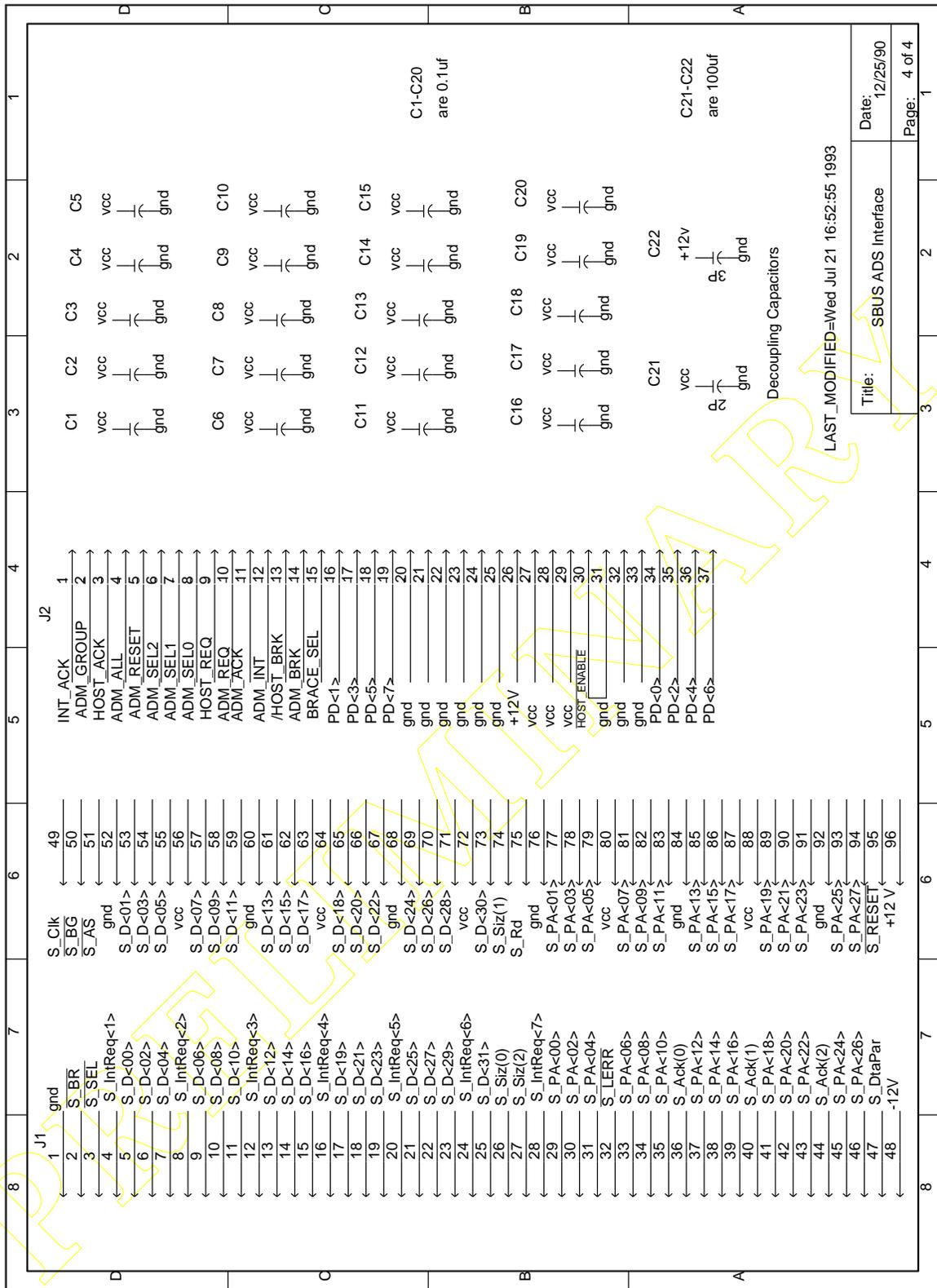
Note: This card is not supported on Sun Sparc 1 and Sun Sparc 1 Plus systems.

Figure 6-4 Sun Sparc SBus Interface Card (Page 4 of 4)

6.8   COMMAND CONVERTER CABLES AND SCHEMATICS

Figure 6-5 Command Converter Power Cable

COMMAND CONVERTOR ONCE INTERFACE CABLE

Pin #14

Place key in pin 8 of both connectors

Pin #2

Ribbon Cable
28 AWG Stranded
0.50" Conductor Spacing.

14 pin mass terminating connector with strain reliefs (2 places).

notch

Pin #11

Pin #1

Use blue band on ribbon cable to indicate pin #1 orientation.

12"
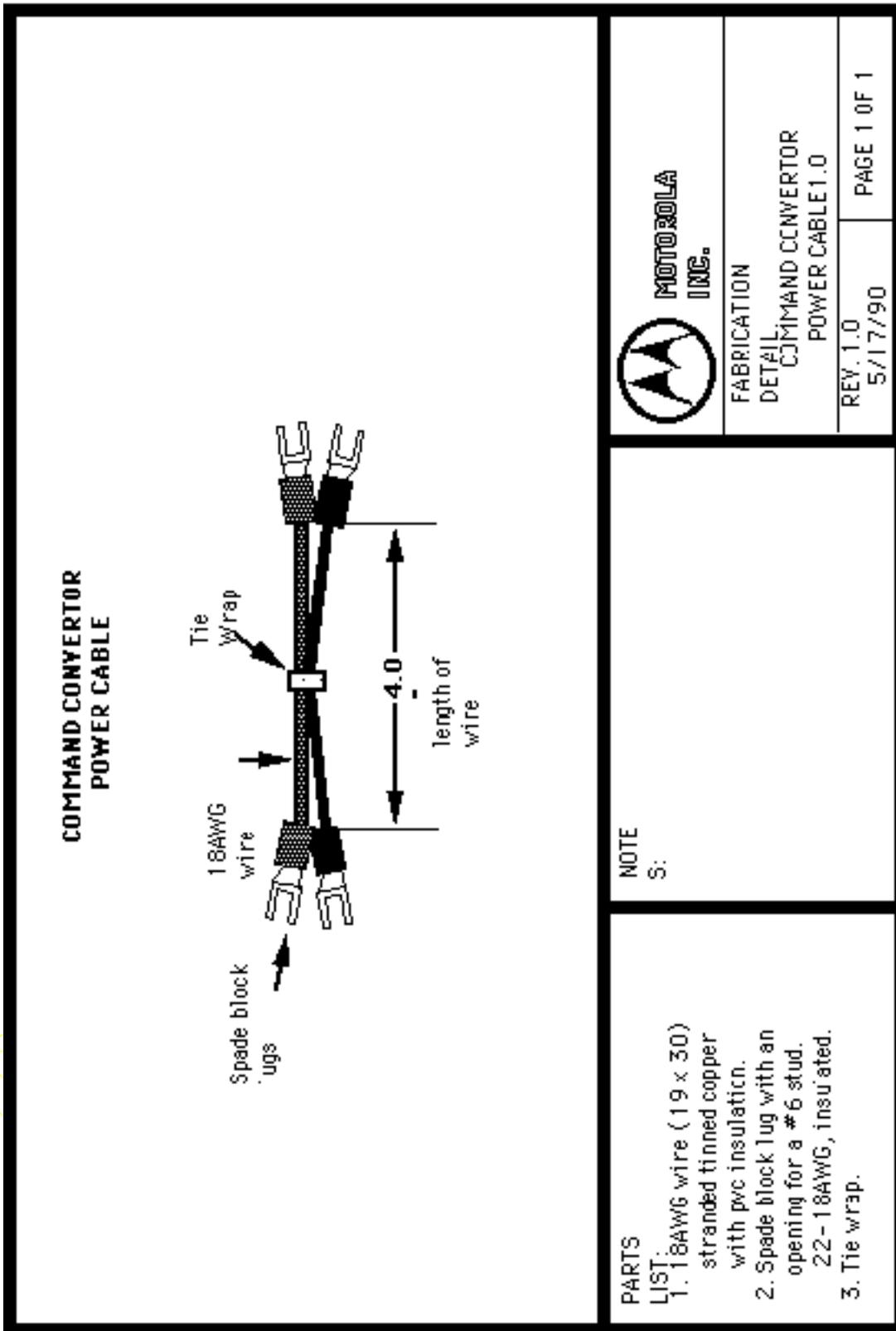
MOTOROLA INC.

FABRICATION DETAIL.
COMMAND CONVERTOR   ONCE
INTERFACE CABLE2.0

PAGE 1 OF 1

REV.2.0
8/31/93

NOTES:

1. Pin 1 orientation from connector to connector must be consistent.

PARTS LIST:
Robinson/Nugent.
1 TDS-C14 NPK-TR (14PIN DIN).
2 SR-14K-IDS (STRAIN RELIEF).
3 PK-1 (KEY).
? & B Ansley.
4 171-14 (RIBBON CABLE).

Figure 6-6 Command Converter OnCE Interface Cable
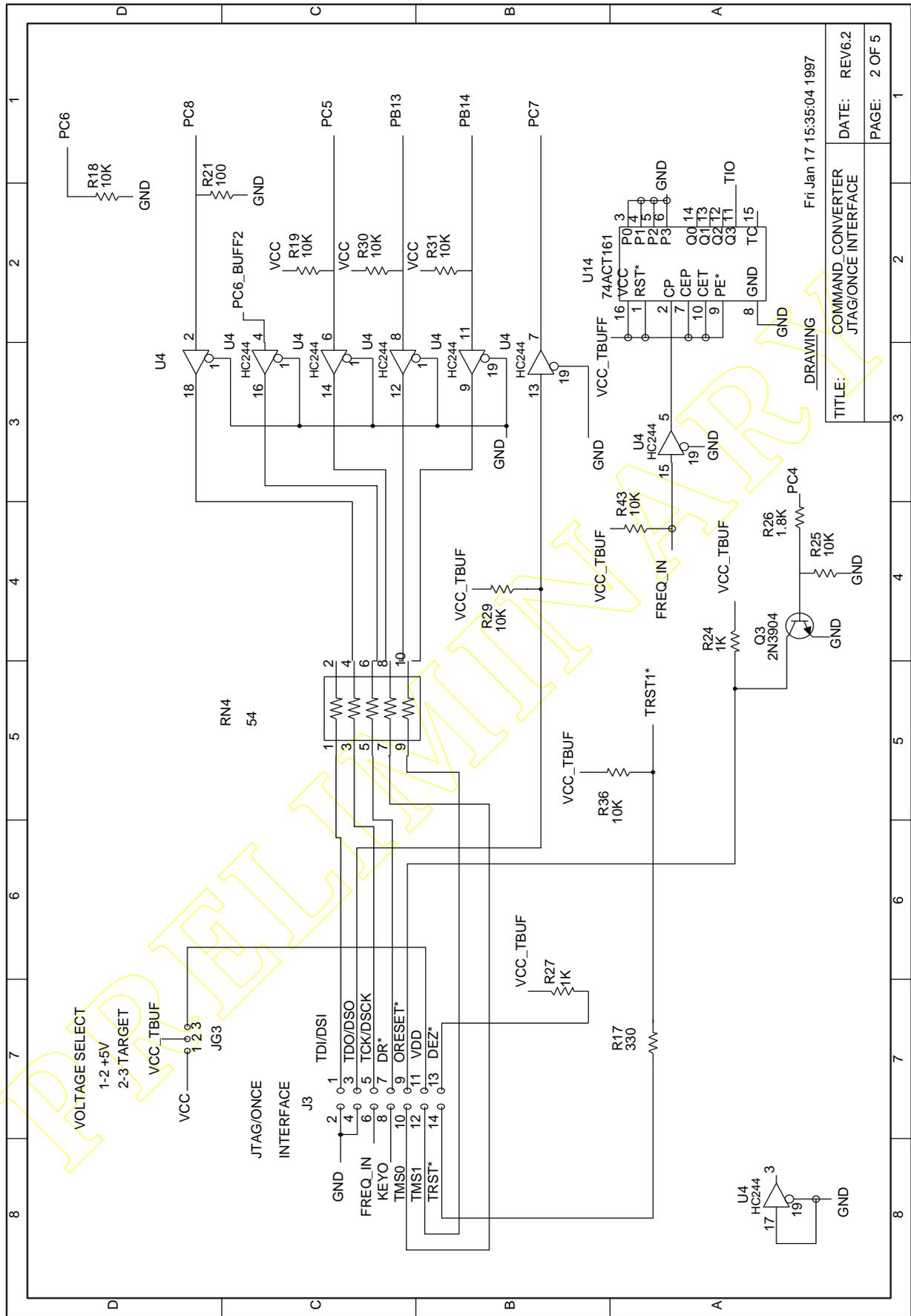
Figure 6-**7** JTAG∕OnCE Command Converter Schematic Rev. 6 (Page 1 of 5)

Figure 6-**7** JTAG/OnCE Command Converter Schematic Rev. 6 (Page 2 of 5)
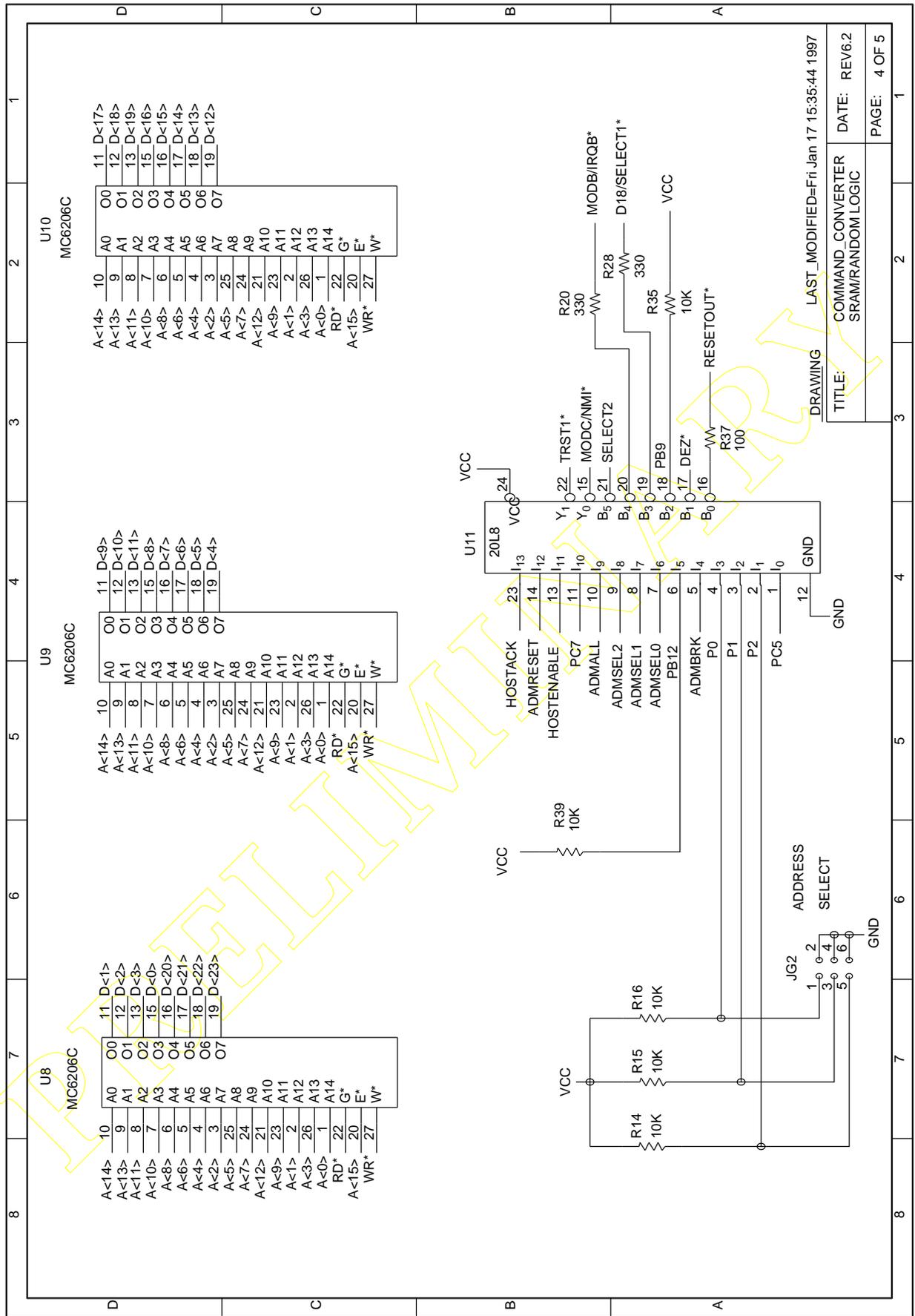
Figure 6-7 JTAG/OnCE Command Converter Schematic Rev. 6 (Page 3 of 5)
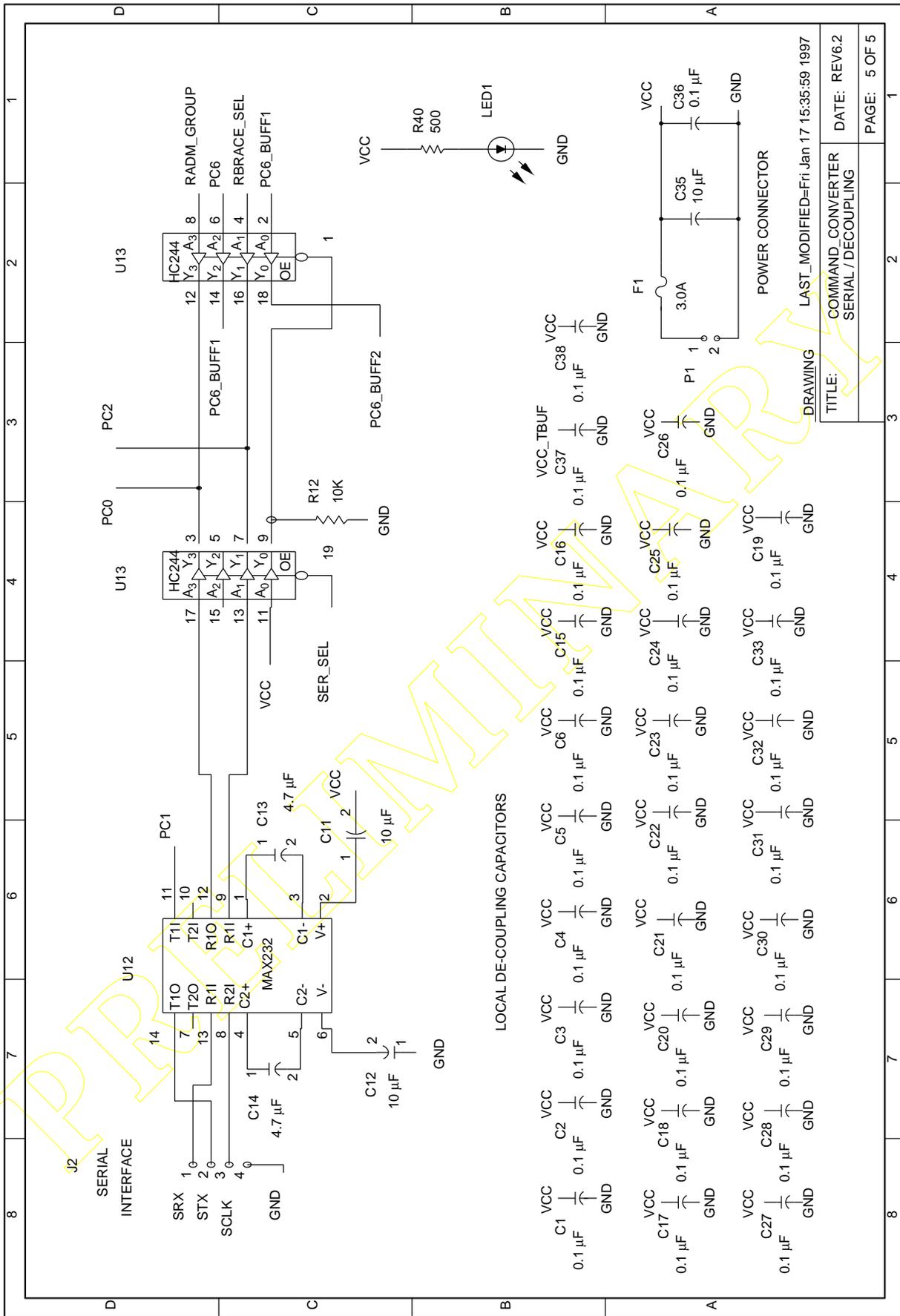
Figure 6-7 JTAG/OnCE Command Converter Schematic Rev. 6 (Page 4 of 5)

Figure 6-7 JTAG/OnCE Command Converter Schematic Rev. 6 (Page 5 of 5)

APPENDIX A

MOTOROLA DSP OBJECT MODULE FORMAT (OMF)

A.1   INTRODUCTION

The Object Module Format (OMF) produced by the DSP cross-assembler 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 each record are separated by whitespace characters (blank, tab, form feed, newline). The general format for a DSP OMF record is illustrated below ("ws" represents whitespace):

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

Every record starts with a Type Definition Field, which 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 Field 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.

A.2   RECORD DEFINITIONS

There are six DSP OMF record types defined

- Start

- End

- Data

- Blockdata

- Symbol

- Comment

Note: Currently, Data records are used for both code and data.

A.2.1   Start Record

Format:

```
_START      <Module id> <Version> <Revision number>
            <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.

A.2.2   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'. The address is in the range 000000000–FFFFFFFF.

A.2.3   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 Memory Space specifier is a single character representing the memory space to be loaded (X, Y, L, or P). The character 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 range of the address is 00000000–FFFFFFFF. 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 range of the values is between 0 and FFFFFFFF. 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.

A.2.4   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 is a single character representing the memory space to be loaded (X, Y, L, or P). The character 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 range of the address is 00000000–FFFFFFFF.

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, range 0–FFFFFFFF).

A.2.5   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 is a single character representing the memory space to be loaded (X, Y, L, or P). The character may be upper or lower case, although the assembler guarantees upper case output.

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 alphanumeric 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. The range of the address is 00000000–FFFFFFFF.

A.2.6 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.

A.3 OBJECT MODULE FORMAT EXAMPLE

Example A-1 DSP56000 assembler code fragment

```
FIR         ident       1,1             ; Complex Correlation/Convolution
n           equ         500
            org         x:$0
aaddr       ds          1024
            org         y:$0
baddr       ds          1024
            org         p:$0
start
            move        #aaddr,r0
            move        #baddr,r4
            clr
            clr         b               x:(r0),x1   y:(r4),y0
loop
            do          #n,endloop
            mac         x1,y0,b         x:(r4)+,x0 y:(r0)+,y1
            mac         x0,y1,b
            mac         x1,x0,a
            mac         -y1,y0,a        x:(r0),x1   y:(r4),y0
endloop
            rnd         a
            rnd         b
end
```

Example A-2 Corresponding Assembler OMF Output File

```
_START FIR 0001 0001
Complex Correlation/Convolution
_DATA P 0000 300000 340000 200013 C4801B 06F481 000009
F19CEA 2000CA 2000A2 C480B6 200011 200019
_END 0000
```

APPENDIX B

MOTOROLA DSP OBJECT FILE FORMAT (COFF)

B.1   INTRODUCTION

The Motorola DSP Assembler and Linker produce a binary object file in a modified form of the AT&T Common Object File Format (COFF). COFF is a formal definition for the structure of machine code files. It originated with Unix System V but has sufficient flexibility and generality to be useful in non-hosted environments. In particular, COFF supports user-defined sections and contains extensive information for symbolic software testing and debugging. Later sections describe the COFF implementation for the Motorola family of Digital Signal Processors. The DSP COFF format has been altered to support multiple memory spaces and normalized to promote transportability of object files among host processors. See **Section B.5 Object File Data Expression Format** on page B-28 for a list of differences between the Motorola DSP object file format and standard COFF.

Note: A general discussion of COFF is provided in the following reference:

Gintaras R. Gircys, *Understanding and Using COFF*, O'Reilly & Associates, 1988 (ISBN 0-937175-31-5).

B.2   OBJECT FILE STRUCTURE

A DSP COFF object file consists of up to eight groups of object file information. Some of these groups are optional, depending on the type of object file generated, and others may have repeating occurrences. The basic object file components are:

- File header
- Optional header
- Section headers
- Section data
- Relocation information
- Line numbers
- Symbol table
- String table

The general structure of the object file is listed in **Table B-1**.

Table B-1   Basic COFF File Structure

File Header
Optional Header
Section 1 Header • • Section *n* Header
Section 1 Contents • • Section *n* Contents
Section 1 Relocation Info • • Section *n* Relocation Info
Section 1 Line Numbers • • Section *n* Line Numbers
Symbol Table
String Table

The file header contains object file information such as timestamp, number of sections, pointer to the symbol table, and file status flags. Depending on how the object file was generated, the Optional Header may hold link or run time information. The Optional Header is followed by a list of Section Headers. Each Section Header contains pointers to Section Data, Relocation Information, and Line Number entries. After the Section Headers comes the raw data for all sections. If the object file is relocatable, the raw data may be followed by a block of Relocation Entries for all sections. If the original source file was compiled or assembled with the -G debug option, the Relocation Information is followed by source Line Number and Address entries. The Symbol Table contains information on program symbols used by both the Linker and the Debugger. The String Table may contain very long symbolic names, comment text, or relocation expressions.

Note: The last four groups (Relocation Info, Line Number entries, Symbol Table, and String Table) may not appear if the linker –S option is used to strip symbols from the object file.

B.3   OBJECT FILE COMPONENTS

The following sections are detailed descriptions of each DSP COFF object file component. The descriptions include the purpose of the component, its structure in the object file, and meanings of individual fields within the component.

B.3.1   FILE HEADER

The file header is the first component in a COFF object file. It contains information about the object file itself and is used for negotiating other components within the file. There is only one file header per object file. **Table B-2** shows the structure of the COFF file header.

Table B-**2**   File Header Format

Bytes	Declaration	Name	Description
0–3	unsigned long	f_magic	Magic number
4–7	unsigned long	f_nscns	Number of sections
8–11	long int	f_timdat	Time and date when file was created
12–15	long int	f_symptr	File pointer to the start of the symbol table
16–19	long int	f_nsyms	Number of symbol table entries
20–23	unsigned long	f_opthdr	Number of bytes in the optional header
24–27	unsigned long	f_flags	Flags (**See Table B-3** on page B-6.)

The magic number is a special code indicating the target machine for the object file (DSP56000, DSP96000, etc.). The number of sections is useful for scanning the list of section headers. The date and time stamp is kept in binary form and may contain a host-dependent time value. The f_symptr field contains a file byte offset to the beginning of the symbol table. The number of symbol table entries provides an upper bound for looping through the symbol table and an indirect means for accessing the start of the string table. The size of the optional header allows for jumping to the start of the section header list. The flags field is a set of bit flags which convey status information about the object file. It is used primarily by linkers, debuggers, and other loader software to determine whether the file is valid for a particular requested operation. The individual bit flags are shown in **Table B-3**.

Table B-3   File Header Flags

Mnemonic	Flag	Meaning
F_RELFLG	0000001	Relocation information stripped from file
F_EXEC	0000002	File is executable (no unresolved external references)
F_LNNO	0000004	Line numbers stripped from file
F_LSYMS	0000010	Local symbols stripped from file
F_CC	0010000	File produced by C compiler (Motorola DSP only)

B.3.2   Optional Header

The COFF optional header ordinarily is used to hold system-dependent or runtime information. This allows different operating environments to store data that only that environment uses without requiring all COFF files to reserve space for that information. General utility programs can be made to work properly with any common object file. This is done by seeking past the optional header using the f_opthdr size field in the file header record. The optional header in a Motorola DSP object file may contain two distinct types of information, depending upon how the file was generated. If the file is a relocatable object file, it will have an optional header containing linker information. If the file is an absolute object file, it will have an optional header containing runtime information. The runtime header is similar to standard COFF a.out optional header formats.

Table B-4 on page B-7 shows the linker optional header. The module size field gives the size of the entire object module. The data size field reflects the size of the entire raw data block within the module. The endstr field points to an expression in the string table which originated with the assembler END directive (see **Chapter 6** of the *DSP Macro -assembler Manual*); it indicates the starting address of the module. If this field is negative or zero, there is no end expression. The logical section count is the count of sections in the object module created via the assembler SECTION directive. The counter count represents the number of COFF sections in the file (analogous to the file header f_nscns field). The relocation entry and line number counts hold the number of all relocation entries and line number records in the file. The buffer and overlay counts give counts for each instance of a buffer or overlay in the module. The major version, minor version, and revision number fields reflect the assembler and linker versions to ensure linker backward compatibility. The optional header flags hold special mode flags for the linker.

Table B-4 Motorola DSP Optional Link Header Format

Bytes	Declaration	Name	Description
0–3	long int	modsize	Object module size
4–7	long int	datasize	Module raw data size
8–11	long int	endstr	End directive expression string
12–15	long int	secnt	Logical section count
16–19	long int	ctrcnt	Counter count
20–23	long int	relocnt	Relocation entry count
24–27	long int	lnocnt	LIne number entry count
28–31	long int	bufcnt	Buffer count
32–35	long int	ovlcnt	Overlay count
36–39	long int	majver	Major version number
40–43	long int	minver	Minor version number
44–47	long int	revno	Revision number
48–51	long int	optflags	Optional header flags

Table B-5 on page B-8 defines the contents of the runtime optional header. This header is similar to the standard COFF a.out header but there are differences. The magic number in this header is not the same as the magic number in the file header; this magic number is used indicate the file type to a host operating system. The magic number and version stamp fields currently are not used by the Motorola DSP tools and are set to zero. The text size field gives the size of all text-type data (executable code) in the object file. The data size field holds a count of all initialized data (apart from code) in the file. The uninitialized data size field is not used and is set to zero.

The program entry field represents the address given in the assembler END directive. The text start and data start values contain the low addresses for text and data segments, respectively. The text and data end values contain the high addresses for text and data segments, respectively. Note that addresses are expressed in terms of the C language typedef CORE_ADDR. A CORE_ADDR is a structure containing a long (4 byte) address and an enumeration type which classifies the address according to memory space (X, Y, L, P) and memory mapping (internal, external, etc.). See **Section B.4.1 Multiple Memory Spaces** on page B-24 for more information on the CORE_ADDR structure.

Table B-5   Motorola DSP Optional Runtime Header Format

Bytes	Declaration	Name	Description
0–3	long int	magic	Magic number
4–7	long int	vstamp	Version stamp
8–11	long int	tsize	Size of text in words
2–15	long int	dsize	Size of data in words
16–19	long int	bsize	Size of uninitialized data in words
20–27	CORE_ADDR	entry	Program entry point
28–35	CORE_ADDR	text_start	Base address of text
36–43	CORE_ADDR	data_start	Base address of data
44–51	CORE_ADDR	text_end	End address of text
52–59	CORE_ADDR	data_end	End address of data

B.3.3   Sections

A section is the smallest portion of an object file that is treated as one separate and distinct entity. Sections can accommodate program text, initialized and uninitialized data, and block data. COFF sections in DSP object files may be grouped under a logical section defined by the assembler SECTION directive.

It is a mistake to assume that every COFF file will have a specific number of sections, or to assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the object file has been created. Programs manipulating COFF files should obtain it from file and section headers in the file.

B.3.3.1   Section Headers
Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in **Table B-6** on page B-9.

Table B-6 Section Header Format

Bytes	Declaration	Name	Description
0–7	char	s_name	Section name (null padded)
8–15	CORE_ADDR	s_paddr	Physical address
16–23	CORE_ADDR	s_vaddr	Virtual address
24–27	long int	s_size	Section size in words
28–31	long int	s_scnptr	File pointer to raw data
32–35	long int	s_relptr	File pointer to relocation entries
36–39	long int	s_lnnoptr	File pointer to line number entries
40–43	unsigned long	s_nreloc	Number of relocation entries
44–47	unsigned long	s_nlnno	Number of line number entries
48–51	long int	s_flags	Section flags (see Figure E–7)

The section name is an 8-byte character array padded with null (zero) bytes if required. In Motorola relocatable object files, section names may be longer than eight characters. In this case, the convention used for long symbol names is followed, where if the least significant four bytes of the section name field contain zeroes, the name is in the symbol table at the offset given by the most significant four bytes of the name field. See **Section B.3.3.5 Symbol Name** on page B-14 for more information on the handling of long symbol names.

The physical address is the address where the section text or data will reside in memory. The address value depends upon whether the section is absolute or relocatable. If the section is absolute, then the physical address is the actual address where the section will be loaded into memory. If the section is relocatable, then the physical address is an offset from the start of the logical section (implicit or defined by the SECTION directive) in which the section is defined.

In most cases, the virtual address is the same as the physical address. However, for block data sections in Motorola DSP object files, the virtual address field holds the repeat count for the single raw data value associated with this section. For example, if the assembly language source file included a directive of the form BSC $400,$FFFF, the s_vaddr field would contain the value $400, the s_size field would be 1 (or 2 if in L memory), and the single raw data word associated with the section would be $FFFF.

Table B-7   Section Header Flags

Mnemonic	Flag	Meaning
STYP_REG	$0000	Regular section
STYP_DSECT	$0001	Dummy section
STYP_NOLOAD	$0002	Noload section
STYP_GROUP	$0004	Grouped section
STYP_PAD	$0008	Padding section
STYP_COPY	$0010	Copy section
STYP_TEXT	$0020	Executable text section
STYP_DATA	$0040	Initialized data section
STYP_BSS	$0080	Uninitialized data section
STYP_BLOCK	$0400	Block data section
STYP_OVERLAY	$0800	Overlay section
STYP_MACRO	$1000	Macro section

The section size is the count of raw data words associated with the section. This is in contrast to standard COFF section sizes which usually are given in bytes. Raw data words currently are stored in the object file as long (4-byte) integers independent of the target processor word size. The file pointer fields are file byte offsets into the object file to the start of the current section raw data, relocation entries, and line number information. The counts of relocation and line number entries provide an upper bound for scanning these tables. The section flags comprise the section attributes and are described in **Table B-7**. Text sections are reserved for code to be loaded into program memory (P space). Data sections hold initialized data, generated by assembler DC directives for example, bound for data (X, Y, L) memory. Bss sections are used for uninitialized blocks resulting from assembler DS and similar directives. Padding sections are generated to provide alignment when a modulo or reverse-carry buffer is declared. The block section attribute flags a block data section, described above. The overlay flag indicates the section is part of an overlay. Macro sections represent code and data generated during a macro expansion. Dummy sections are used internally by the assembler to mark empty sections after the first assembly pass. Empty sections may still appear in the object file if a symbol is associated with a section which contains no data. The noload, group, and copy attributes are not used at present.

B.3.3.2   Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in **Table B-8**.

Table B-**8**   Relocation Entry Format

Bytes	Declaration	Name	Description
0–3	long	r_vaddr	Address of reference
4–7	long	r_symndx	String table index
8–11	unsigned long	r_type	Relocation type

The address field represents the relocatable address within the section raw data where a modification is needed. In standard COFF the r_symndx field points to an entry in the symbol table corresponding to the reference requiring modification. The relocation type encodes how the raw data is to be changed to reflect the resolved symbol value.

In Motorola DSP COFF r_symndx is an offset into the string table which points to a relocation expression. The linker interprets this expression and updates the word at r_vaddr with the result of the expression evaluation. The relocation type is always zero. See **Section B.5 Object File Data Expression Format** on page B-28, for more information on relocation expressions.

B.3.3.3   Line Numbers

When the compiler or assembler is invoked with the –G debug option an entry is made in the object for every source line where a breakpoint can be inserted. It is then possible to reference source line numbers when using a debugger. The structure of an object file line entry is shown in **Table B-9**.

Table B-**9**   Line Number Entry Format

Bytes	Declaration	Name	Description
0–3	long	l_symndx	Function name symbol table index
0–7	CORE_ADDR	l_paddr	Line number physical address
8–11	unsigned long	l_lnno	Source file line number

All line numbers in a section are grouped by function as shown in **Table B-10** on page B-12. The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the

program text corresponding to the line numbers. The line number entries are relative to the beginning of the function, and appear in increasing order of address.

Table B-10   Line Number Grouping

Symbol index	0
Physical address	Line number
Physical address	Line number
. . .	. . .
Symbol index	0
Physical address	Line number
Physical address	Line number

B.3.3.4   Symbol Table

The COFF symbol table serves a dual purpose: it provides resolution for symbolic references in relocation expressions during linking, and it establishes a framework for the handling of symbolic debug information. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size.

Because of symbolic debugging requirements the order of symbols in the symbol table is very important. Whereas an individual symbol table entry can completely describe a single debugging entity, the entities exist within the framework of the source language that produced them. For example, symbol scoping and function blocks in C are represented by the appropriate ordering of begin-end block entries in the symbol table. Symbols in the symbol table appear in the sequence shown in **Table B-11**.

Table B-11   COFF Symbol Table Ordering

Filename 1
Function 1
Local symbols for function 1
Function 2
Local symbols for function 2
. . .
Statics

Table B-11 COFF Symbol Table Ordering (Continued)

. . .
Filename 1
Function 1
Local symbols for function 1
. . .
Statics
. . .
Defined global symbols
Undefined global symbols

The entry for each symbol is a structure that holds the symbol value, its type, and other information. There are symbol table entries used for relocation and linking and there are special symbols used only for debugging. The two kinds of entries are distinguished by combinations of field values in the symbol record. The structure of a symbol table entry is illustrated in **Table B-12**.

Table B-12 Symbol Table Entry Format

Bytes	Declaration	Name	Description
0–7	char	n_name	Symbol name (null padded)
0–3	long int	n_zeroes	Zero in this field indicates name is in string table
4–7	long int	n_offset	Offset of name in string table
8–15	CORE_ADDR	n_address	Symbol address value
8–15	unsigned long	n_val[2]	Symbol value
16–19	long int	n_scnum	Symbol section number
20–23	unsigned long	n_type	Symbol basic and derived type
24–27	long int	n_sclass	Symbol storage class
28–31	long int	n_numaux	Number of auxiliary entries

B.3.3.5 Symbol Name

The first eight bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is seven characters or less, the null-padded symbol name is stored there. If the symbol name is longer than seven characters, then the entire symbol name is stored in the string table. In this case, the eight bytes contain two long integers: the first is zero and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first four bytes serve to distinguish a symbol table entry with an offset from one with a name in the first eight bytes.

B.3.3.6 Symbol Value

The symbol value is a union of a CORE_ADDR typedef and an array of two longs. If the symbol value is an address the contents will be stored as a CORE_ADDR structure with memory and mapping attributes. Otherwise the contents are stored in the n_val field. Whether the symbol value is an address or not depends on the storage class of the symbol. See **Section B.3.3.9 Symbol Storage Class** on page B-16 for more information on the relationship of symbol value and storage class.

B.3.3.7 Section Number

The section number maps a symbol to its corresponding section in the object file (e.g. the section in which the symbol is defined). A special section number (-2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of -1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and end-of-structure symbols. A section number of 0 flags a relocatable external symbol that is not defined in the current file. Section numbers greater than zero correlate to the ordinal sequence of sections in the object file.

Table B-13 Fundamental Types

Mnemonic	Value	Type
T_NULL	0	Type not assigned
T_VOID	1	Void
T_CHAR	2	Character
T_SHORT	3	Short integer
T_INT	4	Integer
T_LONG	5	Long integer
T_FLOAT	6	Floating point

Table B-13 Fundamental Types (Continued)

Mnemonic	Value	Type
T_DOUBLE	7	Double word floating point
T_STRUCT	8	Structure
T_UNION	9	Union
T_ENUM	10	Enumeration
T_MOE	11	Member of enumeration
T_UCHAR	12	Unsigned character
T_USHORT	13	Unsigned short
T_UINT	14	Unsigned integer
T_ULONG	15	Unsigned long

B.3.3.8   Symbol Type

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the compiler and assembler only if the –G debug option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The type information is encoded as sets of bits in the field. Bits 0–3 hold one of the fundamental type values given in **Table B-13**. Bits 4–15 are arranged as six 2-bit subfields. These subfields represent levels of the derived types given in **Table B-14**.

Table B-14 Derived Types

Mnemonic	Value	Type
DT_NON	0	No derived type
DT_PTR	1	Pointer
DT_FCN	2	Function
DT_ARY	3	Array

As an example of encoding fundamental and derived types, consider a function returning a pointer to a character. The fundamental type is character, giving bits 0–3 of the symbol type field the value 2. Bits 4–5 would hold a 2 for the derived type of function and bits 6–7 would contain a 1 for the pointer derived type. The value in the symbol entry type field would result in %01100010 binary, or $62 hexadecimal.

B.3.3.9 Symbol Storage Class

The symbol storage class indicates how a symbol will be used during execution or debugging. Some storage classes actually reflect how a symbol will be stored, e.g. as a register parameter. Other storage classes provide information for special symbols used in debugging, such as the beginning of blocks or the end of functions. Storage classes are outlined in **Table B-15**. The value of a symbol depends on its storage class. This relationship is summarized in **Table B-16** on page B-17.

Table B-15   Storage Classes

Mnemonic	Value	Type
C_EFCN	-1	Physical end of function
C_NULL	0	No storage class
C_AUTO	1	Automatic variable
C_EXT	2	External symbol
C_STAT	3	Static symbol
C_REG	4	Register variable
C_EXTDEF	5	External definition
C_LABEL	6	Label
C_ULABEL	7	Undefined label
C_MOS	8	Member of structure
C_ARG	9	Function argument
C_STRTAG	10	Structure tag
C_MOU	11	Member of union
C_UNTAG	12	Union tag
C_TPDEF	13	Type definition
C_USTATIC	14	Uninitialized static
C_ENTAG	15	Enumeration tag
C_MOE	16	Member of enumeration
C_REGPARAM	17	Register parameter
C_FIELD	18	Bit field
C_BLOCK	100	Beginning and end of block
C_FCN	101	Beginning and end of function
C_EOS	102	End of structure
C_FILE	103	C language source filename
C_LINE	104	-
C_ALIAS	105	Duplicated tag
C_HIDDEN	106	-

Table B-15 Storage Classes (Continued)

Mnemonic	Value	Type
A_FILE	200	Assembly source filename
A_SECT	201	Beginning and end of section
A_BLOCK	202	Beginning/end of COFF section
A_MACRO	203	Macro expansion
A_GLOBAL	210	Global assembly language symbol
A_XDEF	211	XDEFed symbol
A_XREF	212	XREFed symbol
A_SLOCAL	213	Section local label
A_ULOCAL	214	Underscore local label
A_MLOCAL	215	Macro local label

Table B-16 Storage Class and Value

Storage Class	Value
C_AUTO	Stack offset in words
C_EXT	Relocatable address
C_STAT	Relocatable address
C_REG	Register number
C_LABEL	Relocatable address
C_MOS	Offset in words
C_ARG	Stack offset in words
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	Enumeration value
C_REGPARAM	Register number
C_FIELD	Bit displacement
C_BLOCK	Relocatable address
C_FCN	Relocatable address
C_EOS	Size of structure in words
C_FILE	(see below)
C_ALIAS	Tag index

Table B-16   Storage Class and Value (Continued)

Storage Class	Value
C_HIDDEN	Relocatable address
A_FILE	(see below)
A_SECT	String table offset to section name
A_BLOCK	Relocatable address
A_MACRO	String table offset to macro name
A_GLOBAL	Relocatable address
A_XDEF	Relocatable address
A_XREF	String table offset to symbol name
A_SLOCAL	Relocatable address
A_ULOCAL	Relocatable address
A_MLOCAL	Relocatable address

If a symbol has storage class C_FILE or A_FILE, the value of that symbol equals the symbol table entry index of the next C_FILE or A_FILE symbol. That is, the C_FILE and A_FILE entries form a one-way linked list in the symbol table. If there are no more C_FILE or A_FILE entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the relocatable address of that symbol. When the section is relocated by the linker, the value of these symbols changes.

B.3.3.10   Auxiliary Entries
Every symbol table entry may have zero, one, or more auxiliary entries. These auxiliary entries are used to hold additional information about the primary symbol. The number of auxiliary entries associated with a given symbol can be determined by examining the n_numaux field of the main symbol entry.

An auxiliary symbol table entry contains the same number of bytes as its associated symbol table entry and is contiguous with the primary entry in the object file. Unlike primary symbol table entries, however, the format of an auxiliary entry depends on the type and storage class of the main symbol.

B.3.3.10.1   Filenames
The auxiliary table entry for a filename contains a 14-character array followed by an unsigned long integer. If the integer is zero then the filename is in the array. Otherwise it is in the string table at the offset given by the integer value.

B.3.3.10.2 Sections

Section auxiliary entries have the format shown in **Table B-17**. This information is analogous to selected fields in the corresponding section header. If the object file is relocatable a section symbol entry will have a second auxiliary entry with the format shown in **Table B-18**.

Table B-17 Section Symbol Auxiliary Entry

Bytes	Declaration	Name	Description
0–3	long int	x_scnlen	Section length
4–7	unsigned long	x_nreloc	Number of relocation entries
8–11	unsigned long	x_nlinno	Number of line numbers
12–31	—	—	Unused (zero filled)

Table B-18 Section Symbol Auxiliary Entry

Bytes	Declaration	Name	Description
0–3	long int	secno	Logical section number
4–7	long int	rsecno	Logical relocation section number
8–11	long int	flags	Section type flags
12–27	struct mematt	mem	Section memory attributes
28–31	—	—	Unused (zero filled)

The logical section number is the ordinal related to a SECTION directive in the assembler source file. The relocation section number usually is the same as the logical section number, but may be different if the logical section is static within an enclosing section. The memory mapping is an alternate encoding of the CORE_ADDR information in the section header. Section type flags indicate whether this COFF section represents a buffer or overlay block. If the current COFF section is a buffer or overlay block a third auxiliary entry is produced. The layout of that entry is shown in **Table B-19** on page B-20.

Table B-19 Relocatable Buffer/Overlay Auxiliary Entry

Bytes	Declaration	Name	Description
0–3	long int	bufcnt	Buffer section number
4–7	long int	buftyp	Buffer type
8–11	long int	buflim	Buffer limit
0–15	struct mematt	ovlmem	Overlay memory attributes
16–19	long int	ovlcnt	Overlay section number
20–23	long int	ovlstr	Overlay origin expression
24–31	—	—	Unused (zero filled)

Buffers and overlays are mutually exclusive so their respective fields share storage space in the object file. The buffer section number is really the buffer instance count in this file. Buffer type is either modulo or reverse carry. The buffer limit gives the upper bound for the buffer size even though the block may contain less initialized data than this limit suggests. The overlay memory structure gives the runtime memory attributes for this block. The overlay section number is really the overlay instance count in this file. The overlay origin expression is the expression given for the runtime counter in the assembler ORG directive.

B.3.3.10.3   Tag Names
Auxiliary entries for C language structure and union tag names have the format described in **Table B-20**.

Note: In Motorola DSP COFF the size of the associated structure or union is in words as opposed to bytes as in standard COFF. The x_endndx field is used to create a linked list of tag name entries through the symbol table.

Table B-20 Tag Name Symbol Auxiliary Entry

Bytes	Declaration	Name	Description
0–7	—	—	Unused (zero filled)
8–11	unsigned long	x_size	Size of structure, union, or enumeration in words
12–15	—	—	Unused (zero filled)
16–19	long int	x_endndx	Index of next structure, union, or enumeration entry
20–31	—	—	Unused (zero filled)

B.3.3.10.4 END OF STRUCTURES

The format for C language end-of-structure auxiliary entries is given in **Table B-21**. Note that the size of the structure, union, or enumeration is given in words rather than bytes. The tag index holds the symbol table index for the tag record associated with this structure.

Table B-**21** End of Structure Auxiliary Entry

Bytes	Declaration	Name	Description
0–3	long int	x_tagndx	Tag index
4–7	—	—	Unused (zero filled)
8–11	unsigned long	x_size	Size of structure, union, or enumeration in words
12–31	—	—	Unused (zero filled)

B.3.3.10.5 FUNCTIONS

Function auxiliary entries have the format shown in **Table B-22**.

Note: The size of the function is given in words rather than bytes.

The function tag index holds the symbol table index to the begin-function symbol for this function. The x_endndx field points to the next function symbol table entry. The x_lnnoptr field contains a byte offset pointer within the object file to the line number entry that signals the start of this function (see **Section B.3.3.3 Line Numbers** on page B-11 for more information).

Table B-**22** Function Symbol Auxiliary Entry

Bytes	Declaration	Name	Description
0–3	long int	x_tagndx	Tag index
4–7	long int	x_fsize	Size of function in words
8–11	long int	x_lnnoptr	File pointer to line number entry
12–15	long int	x_endndx	Index of next function entry
16–31	—	—	Unused (zero filled)

B.3.3.10.6 Arrays

The format for C language array auxiliary entries is given in **Table B-23**. The tag index contains the offset to the next array symbol in the symbol table. The line number field gives the source file line number for the array declaration.

Table B-23 Array Symbol Auxiliary Entry

Bytes	Declaration	Name	Description
0–3	long int	x_tagndx	Tag index
4–7	unsigned long	x_lnno	Line number of array declaration
8–11	unsigned long	x_size	Size of array
12–15	unsigned long	x_dimen[0]	First array dimension
16–19	unsigned long	x_dimen[1]	Second array dimension
20–23	unsigned long	x_dimen[2]	Third array dimension
24–27	unsigned long	x_dimen[3]	Fourth array dimension
28–31	—	—	Unused (zero filled)

B.3.3.10.7 End of Blocks and Functions

The format for C language symbol entries for the end of blocks and functions is given in **Section B.3.3.10.8 Beginning of Blocks and Functions** on page B-23. Only the source file line number for the end of the block or function is stored.

Table B-24 End of Block or Function Auxiliary Entry

Bytes	Declaration	Name	Description
0–3	—	—	Unused (zero filled)
4–7	unsigned long	x_lnno	Source file line number
8–31	—	—	Unused (zero filled)

B.3.3.10.8 Beginning of Blocks and Functions

The format for C language symbol entries for the beginning of blocks and functions is described in **Table B-25**. The source file line number is retained. The x_endndx provides a link to the next beginning of block or function symbol in the symbol table.

Table B-**25**   Beginning of Block or Function Auxiliary Entry

Bytes	Declaration	Name	Description
0–3	—	—	Unused (zero filled)
4–7	unsigned long	x_lnno	Source file line number
8–15	—	—	Unused (zero filled)
16–19	long int	x_endndx	Index of next beginning of block or function
20–31	—	—	Unused (zero filled)

B.3.3.10.9 Structure, Union, and Enumeration Names

The format for auxiliary entries related to structure, union, and enumeration names is given in **Table B-26**. The tag index is used to access the tag symbol record that describes this structure. Note that in Motorola DSP COFF the size of the associated structure or union is in words as opposed to bytes as in standard COFF.

Table B-**26**   Structure, Union, or Enumeration Name Auxiliary Entry

Bytes	Declaration	Name	Description
0–3	long int	x_tagndx	Tag index
4–7	—	—	Unused (zero filled)
8–11	unsigned long	x_size	Size of structure, union, or enumeration in words
12–31	—	—	Unused (zero filled)

B.3.3.10.10 String Table

Symbol and section names longer than seven characters and comment text are stored contiguously in the string table with each string delimited by a zero byte. The first four bytes represent the size of the string table in bytes; offsets into the string table, therefore, are always greater than or equal to 4. An empty string table has a length field with value zero.

B.4 DIFFERENCES BETWEEN DSP OBJECT FORMAT AND STANDARD COFF

Motorola DSP COFF is substantially the same as generic COFF and usage of format elements is similar. However, the original COFF specification did not envision aspects of machine architecture which the Motorola DSP family possesses. Moreover, standard COFF encompasses a file format which is quite adaptable among host processors, but is not necessarily portable among those hosts.   It is straightforward enough to adapt COFF to a new host machine, but the intent is that the derived host format will be recognized and executed only on that target host. For Motorola DSP COFF the format had to be extended for cross-development such that a given object file would be usable on all targeted host systems. The following sections outline the differences and changes between standard COFF and Motorola DSP COFF.

B.4.1 Multiple Memory Spaces

Standard COFF has no built-in mechanism for accommodating multiple memory spaces. It does handle the notion of separate text and data sections, and a possible extension would have been to define section types for the new memory areas. This quickly becomes unwieldy when mapping information (internal, external, port A/B) is considered as well.

The solution was to extend addressing information to include the memory and mapping with the address value itself. This is done by defining a C language typedef called CORE_ADDR which holds both the memory and mapping data along with the memory address. For any address context in the COFF file a CORE_ADDR is used rather than, for example, an unsigned long. A description of the CORE_ADDR format is shown in **Table B-27**.

Table B-27   CORE_ADDR Format

Bytes	Declaration	Name	Description
0–3	long	w0.l	Memory address
4–7	enum	w1.mape	Memory mapping

The enumeration values for the memory mapping field are shown in order in **Table B-28** on page B-25.

Table B-28 Memory Mapping Enumerations

Mnemonic	Value	Mnemonic	Value
memory_map_p	0	memory_map_xa	16
memory_map_x	1	memory_map_xb	17
memory_map_y	2	memory_map_xe	18
memory_map_l	3	memory_map_xi	19
memory_map_none	4	memory_map_xr	20
memory_map_laa	5	memory_map_ya	21
memory_map_lab	6	memory_map_yb	22
memory_map_lba	7	memory_map_ye	23
memory_map_lbb	8	memory_map_yi	24
memory_map_le	9	memory_map_yr	25
memory_map_li	10	memory_map_pt	26
memory_map_pa	11	memory_map_pf	27
memory_map_pb	12	memory_map_error	666666
memory_map_pe	13	—	—
memory_map_pi	14	—	—
memory_map_pr	15	—	—

B.4.2   OBJECT FILE TRANSPORTABILITY

There are many different structure definitions in the COFF specification. These
definitions consist of fields comprised of varying C data types. These data types are
recognized by any reasonable C compiler, but their characteristics and sizes may change
from machine to machine. This is acceptable if the COFF files are to be used only on a
particular machine architecture. But if COFF files are produced on one machine to be
used on another several problems may arise. One is that since the data fields can vary in
size there could be alignment problems when accessing structures or individual fields.
Another issue is byte ordering between machines. Given an arbitrary byte stream, some
machines store the bytes in a word starting at the least significant bit (LSB) end of the
word, while others store bytes starting at the most significant bit (MSB) end of the word.

The Motorola DSP version of COFF addresses these potential problems by normalizing
the object file. Normalization occurs in a number of ways. All structure and union
elements are converted to long values, and raw data is stored in 4-byte quantities

independent of the word size of the target processor. In some cases this wastes space in the object file and in memory but it was considered worth the price for transportability among supported hosts. Also it is not a completely portable solution by any means (e.g. for machines with larger than 4-byte word sizes).

The byte ordering issue was dealt with by establishing a baseline ordering, providing compliance for foreign hosts with conversion code. This introduces overhead logic on machines that do not support the baseline word order but again it was seen as a reasonable trade-off to insure transportability of object files among development environments. Note that byte swapping logic only comes into play for fields that are not byte-atomic, such as integer fields. Character arrays in structures, for example, should not have their bytes exchanged.

The byte ordering for Motorola DSP COFF is shown in **Table B-29**. It adheres to what sometimes is called the big-Endian approach to byte and word ordering.

Table B-29   Motorola DSP COFF Byte Ordering

Addr *n*	Addr *n*+1	Addr *n*+2	Addr *n*+3
MSB	MSB - 1	LSB + 1	LSB

B.4.3   Structure Size Fields

In some of the COFF data structures there is a size field which gives the size of a block in the target processor environment. For example, there are several symbol table auxiliary entries that specify the size of a structure or union for debug purposes. In standard COFF these sizes ordinarily are in bytes but in Motorola DSP COFF they are given in words unless otherwise indicated. The use of word sizes for debug entities should be distinguished from file pointer offset values in the object file. File pointers are indeed byte offsets within the object file that are used by utilities to process information in the object file itself.

B.4.4   Relocation Information

In standard COFF the r_symndx field of any given relocation record points to an entry in the symbol table corresponding to a symbol reference requiring modification. When the standard COFF linker performs symbol resolution, pairing symbol definitions with matching references, it updates the relocation entry to point to the symbol definition and discards the reference symbol. When the relocation entries are processed, the resolved

symbol value is used to modify the raw data indicated by the relocation entry at r_vaddr. In Motorola DSP COFF r_symndx is an offset into the string table which points to a relocation expression. The linker interprets this expression and updates the entire word at r_vaddr with the result of the expression evaluation. The relocation type is always zero. See **Section B.5 Object File Data Expression Format** on page B-28, for more information on relocation expressions.

B.4.5   Block Data Sections

Generic COFF does not make allowance for a block data section. A block data section results from use of the assembler BSC directive, where a large block of memory is initialized with a single value. Block data sections are handled in Motorola DSP COFF by making special use of the section s_vaddr field and adding an informative flag. In most cases the section virtual address is the same as the physical address. However, for block data sections in Motorola DSP object files the virtual address field holds the repeat count for the single raw data value associated with the section. For example, if the assembly language source file included a directive of the form BSC $400,$FFFF the s_vaddr field would contain the value $400, the s_size field would be 1 (or 2 if in L memory), and the single raw data word associated with the section would be $FFFF. In addition, the STYP_BLOCK flag is set in the section s_flags field.

B.4.6   Other Extensions

If the object file is relocatable there are extra structures which the assembler and linker generate to support special constructs such as logical sections, buffers, and overlays. The optional link file header contains information which the linker requires; it is described in **Section B.3.2 Optional Header** on page B-6. Every symbol table entry for a section in a relocatable file has an extra auxiliary entry described in **Section B.3.3.10.2 Sections** on page B-19. One special DSP COFF structure not documented elsewhere is the comment symbol. A comment symbol table entry is emitted either indirectly via the assembler IDENT directive or directly with the COBJ directive (see **Chapter 6** of the *DSP Macro-assembler User Manual*). A comment symbol table entry may be identified by a symbol name of .cmt and a type and storage class of zero. The value field of a comment symbol holds the offset into the string table of the comment text. The section number for a comment symbol produced with the IDENT directive is always –1. Comment symbols generated with the COBJ directive have the section number of the section where the COBJ directive appears in the source file. Comment symbols have no auxiliary entry.

B.5 OBJECT FILE DATA EXPRESSION FORMAT

Object file data expressions are used in data relocation records to represent values to be loaded into memory. An expression is a combination of symbols, constants, operators, and parentheses. Expressions may contain user-defined labels, integers, floating point numbers, or literal strings. An object file data expression generally follows the guidelines of assembler expressions, except that functions are not supported (e.g. they must be evaluated at assembly time), and operators are provided for linker-specific operations. Also, floating point terms found in these expressions are converted to binary values.

B.5.1 Data Expression Generation

Link file data expressions are generated when external or relocatable operands are encountered during assembly or incremental link processing. In most cases the operand expression is copied verbatim from the source and embellished with link evaluation control constructs. For example, consider the source line below:

```
MOVE        #FOO,R0
```

The DSP96000 assembler produces the following encoding for this line in the object file:

```
$3A8D2000   {FOO}@0#0
```

Since the symbol FOO is not known to the assembler it generates a two-word instruction and places a relocation reference to the expression in the position of the second instruction word. The braces ({ }) indicate that this is a user expression that should adhere to certain integrity constraints such as those governing absolute and relative terms. Otherwise the braces are treated much like parentheses. The at sign (@) is a binary operator indicating the memory space of the left operand by the right. The pound sign (#) is a binary operator signifying the size in bits of the left operand by the right. More information on these special operators and their operands is given below.

Here is another example of data expression generation:

```
JCLR        #1,X:LOC,LABEL
```

For this conditional jump the assembler produces the following object file code:

```
(($02A00481&~(~(~0<<8)<<12))I(({LOC}@1#8&~(~0<<8))<<12))   {LABEL}@0#0
```

The first expression is evaluated such that the relative address LOC, resolved at link time, is shifted and masked into the middle eight bits of the base instruction word

($02A00481). The expression could have been more complex if the bit number was an external reference. The relative value of the symbol LABEL occupies the second instruction word.

B.5.2   Data Expression Interpretation

Object file data expressions are similar to standard assembler expressions which generally follow the rules of algebra and boolean arithmetic. They are written using infix notation in conjunction with unary and binary operators and parentheses. There are also extensions to the usual set of assembler arithmetic and grouping operators. These are control constructs that assist the linker in determining the size, type, and characteristics of an expression operand.

B.5.2.1   User Expression—{ ... }
The curly braces ({ }) delimit a user expression within a data expression. A user expression is that part of a data expression that was written by the programmer and not generated by the assembler or linker as part of its control requirements. It is useful to isolate the user expression in order to check for relationships among absolute and relative terms. In all other respects the curly braces behave like parentheses.

B.5.2.2   Relocatable Expression—[...]
The square brackets ([ ]) are used to enclose a relocatable expression. The value contained in the square brackets is an offset from the base of the current section. Usually this grouping operator is placed around the value of an assembler local label (underscore label) since these symbols do not migrate to the link file.

B.5.2.3   Memory Space Operator—@
The at sign (@) is a binary operator that checks the memory space compatibility of the left operand based on the value of the right operand. The right operand can have the following values:

    0 = None
    1 = X space
    2 = Y space
    3 = L space
    4 = P space

The compatibility check is made based on the matrix outlined in the *DSP Macro-assembler User Manual*, EXPRESSION MEMORY SPACE ATTRIBUTE.

B.5.2.4 Bit Size Operator—#

The pound sign (#) is a binary operator used to verify the size in bits of the left operand given the value of the right operand. The following bit sizes and operand type correspondences are defined:

−16	=	16-bit signed short immediate or offset
−7	=	7-bit signed short immediate
−5	=	5-bit signed short offset
−1	=	Negated immediate shift
0	=	DSP word size immediate or absolute
1	=	Immediate shift
5	=	5-bit short immediate
6	=	6-bit short immediate or absolute
7	=	7-bit short immediate or absolute
8	=	8-bit short immediate or absolute
12	=	2-bit short immediate or absolute
15	=	5-bit short absolute
19	=	9-bit short immediate
85,86,87	=	5,6,7-bit I/O short absolute

B.5.2.5 Memory Attribute Operator—:

The colon (:) is used to assign a memory space and counter encoded in the right operand to the left operand. The low sixteen bits of the right operand contain the counter designator for the left operand. The high sixteen bits contain the memory space designator for the left operand. The value here corresponds to the memory space values given for the memory space operator (@) described above.

APPENDIX C

MOTOROLA S-RECORD INFORMATION

C.1 INTRODUCTION

The Motorola S-record format is recognized and supported by a variety of EPROM/EEPROM programmer manufacturers. This appendix describes the S-record format and explains how to convert DSP object module format files to S-record format files.

The Motorola S-record format was devised for the purpose of encoding programs or data files in a printable format for transportation between computer systems as well as development tools. This transportation process can therefore be monitored and the S-records can be easily edited.

C.2 S-RECORD CONTENT

S-records are character strings made of several fields which identify the record type, record length, memory address, code/data, and checksum. Each byte of binary data is encoded as a 2-character hexadecimal number: the first character representing the high-order 4 bits, and the second the low-order 4 bits of the byte. The five fields which comprise an S-record are shown in **Table C-1.** Accuracy of transmission is ensured by the record length (byte count) and checksum fields.

Table C-1   S-Record Fields

Field	Printable Characters	Contents
Type	2	S-record type—S0, S1, S9, etc.
Record	2	Character pair count in record, exclude type, record length.
Address	4, 6, or 8	2-, 3-, or 4-byte address at which the data field is to be loaded into memory.
Code/data	0–2n	From 0 to n bytes of executable code, memory loadable data, or descriptive information. For compatibility with teletypewriters, some programs may limit the number of bytes to as few as 28 (56 printable characters in the S-record).
Checksum	2	Least significant byte of the one's complement of the sum of the values represented by the pairs of characters making up the record length, address, and the code/data fields. Each record may be terminated with a CR/LF/NULL. Additionally, an S-record may have an initial field to accommodate other data such as line numbers generated by some time-sharing systems.

C.3   S-RECORD TYPES

Eight types of S-records have been defined to accommodate the several needs of the encoding, transportation, and decoding functions. The various Motorola upload, download, and other record transportation control programs, as well as cross assemblers, linkers, and other file-creating or debugging programs, utilize only those S-records which serve the purpose of the program. For specific information on which S-records are supported by a particular program, consult the user manual for that program. An S-record may include any of record types listed in **Table C-2**. Only one termination record is used for each block of S-records. Normally only one header record is used, although it is possible for multiple header records to be used.

Table C-2   S-record Types

Type	Description
S0	Header record for each block of S-records. The code/data field may contain any descriptive information identifying the following block of S-records. The address field is normally zeroes.
S1	Code/data record and the 2-byte address at which the code/data is to reside.
S2	Not applicable to DSP56000 programming.
S3	Code/data record and the 4-byte address at which the code/data is to reside.
S4–S6	Not applicable to DSP56000 programming.
S7	Termination record for a block of S3 records. Address field may optionally contain the 4-byte address of the instruction to which control is to be passed. If not specified, the first entry point specification encountered in the input will be used. There is no code/data field.
S8	Not applicable to DSP56000 programming.
S9	Termination record for a block of S1 records. Address field may optionally contain the 2-byte address of the instruction to which control is to be passed. If not specified, the first entry point specification encountered in the input will be used. There is no code/data field.

C.4   S-RECORD CREATION

To convert an object module formatted file to an S-record formatted file a utility program (SREC) is provided with the DSP56000 macro assembler program. A detailed description of this utility program may be found in the DSP56000 macro assembler file SREC.DOC. There are various command line options available when executing the SREC utility. When entering no command line options the default file output will be ".X" for X Data memory values, ".Y" for Y Data memory values, ".L" for Long Data memory values, and ".P" for Program memory values.

To create a file which may be loaded into an EPROM programmer which supports the Motorola S record format and is suitable for "BOOTLOADING", a user must enter the "-b" command line option. This option uses byte addressing when transferring load addresses to S-record addresses. Following example is a listing of the example code fragment used in **Appendix A**:

Example C-1   S-record File, 32-bit Data

```
S0044649521A
S3240000300000340000200013C4801B06F481000009F19CEA2000CA2000A2C480B620001113
S30600212000199F
S7030000FC
```

Entering the "–m" command line option will create 4 separate files with a ".P0", ".P1", ".P2" and ".P3" suffixes for Program memory files. This option splits the 32 bit word into 4 bytes where.P0 files contain S-records for the bits 0–7, .P1 files contain S-records for the bits 8–15, .P2 files contain the bits 16–23 and .P3 files contain the bits 24–31. This option allows the user to program 4 separate EPROMs/EEPROMs where the data byte is aligned to the correct address on each device.

 Entering the "–b" command line option will use byte addressing when transferring load addresses to S-record addresses. This means that load file DATA record start addresses are multiplied by the DSP bytes/word and subsequent S1/S3 record addresses are computed based on the data byte count.

Entering the "–s" command line option will write data to a single file, putting memory space information into the address field of the S0 header record. Bytes may be reversed with the "–r" option.

Entering the "–w" command line option will use word addressing when transferring load addresses to S-record addresses. This means that load file DATA record start addresses are moved unchanged and subsequent S1/S3 record addresses are computed based on the data word count.

S-Record Creation

The following are example listings of the example code fragment used in Appendix A in each of the three files:

Example C-2   S-record File, Low-order Byte

/* This is the FIR.P0 low order byte file of the 24 bit words */

```
S0044649521A
S10F00000000131B8109EACAA2B6111902
S9030000FC
```

Example C-3   S-record File, Middle-order Byte

/* This is the FIR.P1 middle byte file of the 24 bit words */

```
S0044649521A
S10F000000000080F4009C000080000060
S9030000FC
```

Example C-4   S-record File, High-order Byte

/* This is the FIR.P2 high order byte file of the 24 bit words*/

```
S0044649521A
S10F0000303420C40600F12020C420206D
S9030000FC
```

Each file has the same S0 header record and S9 terminator record. The source code for the SREC program is also available to the user and is written in the "C" programming language.

APPENDIX D

C LIBRARY FUNCTIONS

D.1   INTRODUCTION

The ADSDSP emulator package includes several libraries of functions which were used to build the emulator. These libraries allow the user to build his own customized emulator and integrate it with his unique project. The source code for many of the ADSDSP functions is provided, including 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 emulator. The source code can be modified to create an emulator customized for a particular application.

A custom emulator may be built with or without display support. Omitting display support reduces the program size by about half, but sacrifices the screen output facilities used in the ADS, and also the user interface and command parsing and execution routines (which rely on the display routines). A non-display ADS may be used to provide direct program control of the hardware by calling the low-level routines, creating reports and activity logs using standard C functions. Omitting display support does not preclude the use of the standard C input/output functions or the creation of an alternative user interface.

The rest of the appendix covers various aspects of the specification and use of the libraries:

- **Section D.2** lists and groups the functions
- **Section D.3** defines each function
- **Section D.4** defines the display (terminal I/O) functions
- **Section D.5** discusses display and non-display support
- **Section D.6** describes the emulation of multiple DSP devices
- **Section D.7** lists reserved function names
- **Section D.8** describes global data used by the emulator
- **Section D.9** covers tailoring items in the global structures

D.2   ADS OBJECT LIBRARY FILES

D.2.1   ADS OBJECT LIBRARY ENTRYPOINTS

The library functions are listed below. They are divided into groups by their function name prefix. The prefix indicates to which part of the ADS they belong, and if they are available in the display or non-display versions of the emulator.

- `ads_`   ADS-specific routines; both versions
- `dspd_`   driver level; not for use by user code; both versions
- `spd_cc_` Command Converter driver level; not for use by user code; both
- `dspt_`   DSP device dependent routines; both versions
- `dsp_`   both versions
- `dsp_cc_` Command Converter routines; both versions
- `sim_`   display only; user interface routines

The driver-level routines (both those referred to above and those documented in the rest of this appendix) are not intended to be called by emulator code. They are designed to be (directly or indirectly) by the interface routines. They are documented so that the user can rewrite them to drive alternate emulator hardware. Other lower-level functions mentioned in this appendix are not intended to be called by user code; these functions are not documented and are specified by the prefixes `dspl_`, `siml_` and `dsptl_`.

D.2.2   LIBRARY ENTRYPOINTS LISTED BY PREFIX

D.2.2.1   `ads_`—ADS-Specific Utility Routines

`ads_cache_registers(devn);`	Read OnCE and core registers from device
`ads_startup(devp, devtype);`	Initialize ADS database and driver

D.2.2.2   `dspd_cc_`—Command Converter Driver Level Routines

`dspd_cc_architecture` `(devn, device_type);`	Initialize Command Converter for DSP family
`dspd_cc_read_flag` `(devn, flag, value);`	Read Command Converter flag word
`dspd_cc_read_memory(devn, mem, addr, count,` `value);`	Read from Command Converter memory
`dspd_cc_reset(devn);`	Reset Command Converter
`dspd_cc_revision` `(devn, revstring);`	Read Command Converter revision number
`dspd_cc_write_flag` `(devn, flag, value);`	Write Command Converter flag word
`dspd_cc_write_memory(devn, mem, addr, count,` `value);`	Write to Command Converter memory

D.2.2.3   `dspd_`—Driver Level Routines

`dspd_break(devn, command);`	Force running DSP into debug mode
`dspd_check_service_request` `(devn);`	See if DSP is requesting service from host
`dspd_fill_memory(devn, mem, addr, count, value);`	Initialize DSP memory buffer to single value
`dspd_go(devn, opcode, operand);`	Begin execution on target DSP device
`dspd_jtag_reset(devn, reset_type);`	Reset JTAG communications
`dspd_read_core_registers` `(devn, reg_num, count, value);`	Read core registers from DSP device
`dspd_read_memory(devn,` `mem_space, addr, count, value);`	Read memory block from DSP device
`dspd_read_once_registers` `(devn, reg_num, count, value);`	Read once registers from DSP device
`dspd_reset(devn, reset_mode);`	Reset specified DSP device
`dspd_status(devn, mode);`	Determine DSP status
`dspd_write_core_registers` `(devn, reg_num, count, value);`	Write core registers to DSP device
`dspd_write_memory(devn, mem_space, addr, count,` `value);`	Write to memory in DSP device
`dspd_write_once_registers` `(devn, reg_num, count, value);`	Write once registers from DSP device

D.2.2.4   dspt_—DSP DEVICE-SPECIFIC ROUTINES

`dspt_masm_xxxxx` `(mnemonic, ops, err);`	Assemble mnemonic string to ops
`dspt_unasm_xxxxx` `(ops, sr, omr, sdbp);`	Disassemble DSP opcodes

D.2.2.5   dsp_cc_—Command Converter Interface Routines

`dsp_cc_fmem(device, mtype, addr, count, value);`	Fill Command Converter memory block
`dsp_cc_go(devn);`	Start program on Command Converter
`dsp_cc_ldmem(devn, loadfn);`	Load Command Converter Memory from file
`dsp_cc_reset(device);`	Reset Command Converter
`dsp_cc_revision` `(devn, revstring);`	Read Command Converter Monitor Revision
`dsp_cc_rmem` `(device, mtype, addr, value);`	Read Command Converter Memory
`dsp_cc_rmem_blk(device, mtype, addr, count, value);`	Read Command Converter memory block
`dsp_cc_wmem` `(device, mtype, addr, value);`	Write Command Converter memory
`dsp_cc_wmem_blk(device, mtype, addr, count, value);`	Write Command Converter memory block

D.2.2.6   dsp_—ADS Interface Routines

`dsp_alloc(nbytes,clearmem);`	Allocate Memory
`dsp_check_service_request` `(devn);`	Determine if device is requesting service
`dsp_findmem(devn,memname,map);`	Get map index for memory prefix
`dsp_findreg` `(devn,regname,pval,rval);`	Get peripheral and register index
`dsp_fmem` `(devn,map,addr,blocksz,val);`	Fill memory block with a value
`dsp_free(devn);`	Free memory allocated for a DSP device
`dsp_free_mem(cp);`	Free memory block
`dsp_go(devn);`	Initiate program execution
`dsp_go_address(devn, addr);`	Initiate program execution from addr
`dsp_go_reset(devn);`	Initiate program execution after reset
`dsp_init(devn);`	Initialize selected device
`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_realloc(mem_blk, nbytes);`	Reallocate memory block
`dsp_reset(devn);`	Reset specified DSP device
`dsp_rmem (devn,map,addr,mem_val);`	Read DSP memory map addr to mem_val
`dsp_rmem_blk (devn,map,addr,count,value);`	Read Block of DSP Memory Locations
`dsp_rreg (devn,periphn,regn,regval);`	Read DSP peripheral register to regval
`dsp_save(filename);`	Save the state of all devices to filename
`dsp_spath(base, sufx, retn);`	Search path for specified file
`dsp_startup();`	Initialize emulator structures
`dsp_status(devn, mode);`	Determine DSP Device Status
`dsp_step(devn, step);`	Execute counted instructions
`dsp_stop(devn);`	Force DSP device into Debug Mode
`dsp_unlock (device_type,password);`	Unlock password protected device type
`dsp_wmem(devn,map,addr,val);`	Write DSP memory map addr with val
`dsp_wmem_blk (devn,map,addr,count,value;`	Write DSP Memory Block
`dsp_wreg (devn,periphn,regn,regval);`	Write DSP peripheral register with regval

D.2.2.7 sim_—User Interface Routines

`sim_docmd(devn,command_string);`	Perform emulator 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

D.3   LIBRARY FUNCTION DESCRIPTIONS

D.3.1   `ads_cache_registers`—Cache OnCE and Core Registers

```
#include "simcom.h"
#include "protocom.h"
int ads_cache_registers(device_index)
int device;                      /* device affected by command */
```

`ads_cache_registers()` caches all OnCE and core registers on the target device `device_index` to ensure the integrity of values returned by `dsp_rreg()`.

This routine must always be called before calling `dsp_rreg()` each time the device returns to debug mode. If it is not called, the values returned by `dsp_rreg()` may be unreliable.

The return value is `TRUE` on successful completion, `FALSE` otherwise.

Example D-1   `ads_cache_registers()`

```
/* cache registers */
#include "simcom.h"
#include "protocom.h"

int devn = 0;
int periphnum_pc, regnum_pc;
unsigned long pc_value;
int status;

/* Find register reference numbers */
status=dsp_findreg(devn,"pc",&periphnum_pc,&regnum_pc)
status=dsp_check_service_request(devn)
/* Is device requesting service? */
ckerr(status);
if (status)
/* Yes, so... */
{
     status=ads_cache_registers(devn);/* cache the registers */
                                   /* get pc addr for service request */
     status=dsp_rreg(devn,periphnum_pc,regnum_pc,&pc_value);
}
```

D.3.2 `ads_startup`—Initialize ADS Database and Driver

```
#include "simcom.h"
#include "protocom.h"
int ads_startup(devp, devtype)
char *devp;/* device driver name (UNIX) or board address (PC) */
int devtype;/* Device family */
```

`ads_startup()` initializes the device driver and allocates certain ADS data structures. `devp` points to a character string containing the name of the device driver on UNIX™ systems, or the board address on PC systems (DOS or WINDOWS). `devtype` indicates which family of DSP devices is being used.

`ads_startup()` must be called before calling `dsp_startup()`.

Valid values for `devtype` are:

- `ADSP56000`

- `ADSP56300`

- `ADSP96000`

The function return value is `TRUE` on successful completion, `FALSE` otherwise.

Example D-2 `ads_startup()`

```
/* Initialize ADS software (PC)*/
#include "simcom.h"
#include "protocom.h"

int status;

status=ads_startup("100", ADSP56000);



/* Initialize ADS software (UNIX™)*/
#include "simcom.h"
#include "protocom.h"

int status;

status=ads_startup("/dev/mdsp0", ADSP56000);
```

D.3.3   `dspd_break`—Force Running DSP into Debug Mode

```
#include "cc.h"
#include "simcom.h"
#include "driver.h"
int dspd_break(device_index, command)
int device_index;/* DSP device to be affected by command */
int command;/* command to be sent to DSP device */
```

dspd_break() forces the target device device_index into debug mode, using the method specified by the parameter command.

Valid values for command are:

- DSP_JTAG_BREAK—used for devices with JTAG port

- DSP_ONCE_BREAK—used for devices with OnCE port

The function returns DSP_OK if the operation succeeds, DSP_ERROR otherwise.

Example D-3   dspd_break()

```
/* Force DSP into debug mode */
#include "cc.h"
#include "simcom.h"
#include "driver.h"

int devn;
int break_status;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

.....

break_status=dspd_break(devn,DSP_ONCE_BREAK)/* Force device into DEBUG mode */
```

D.3.4   `dspd_cc_architecture`—Initialize Command Converter for DSP Family

```
#include "driver.h"
int dspd_cc_architecture(device_index, device_type)
int device_index;/* DSP device affected by command */
int device_type;/* Type of DSP device */
```

Initializes the Command Converter `device_index` for the target architecture specified by `device_type`. The device attached to the Command Converter may be any member of the specified DSP family.

Valid values for `device_type` are:

- `DSP_CC_56000`

- `DSP_CC_56300`

- `DSP_CC_96000`

Example D-4 `dspd_cc_architecture()`

```
/* Initialize Command Converter for device 0, a 56002 */

#include "driver.h"

int devn, status;

devn=0;

status=dspd_cc_architecture(devn,DSP_CC_56000);/* setup CC for 56K family */

dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */
```

D.3.5   `dspd_cc_read_flag`—Read Command Converter Flag Word

```
#include "cc.h"
#include "simcom.h"
#include "driver.h"
int dspd_cc_read_flag(device_index, flag, value)
int device_index;/* Command Converter affected by command */
int flag;   /* Specify flag to read */
unsigned long *value;/* Location to receive flag value */
```

`dspd_cc_read_flag()` reads flag word `flag` from Command Converter `device_index`, storing the value obtained in the location pointed to by `value`.

Valid values for `flag` are:

- `DSP_CC_FLAGS`

- `DSP_CC_STATUS`

- `DSP_CC_XPTR`

- `DSP_CC_YPTR`

- `DSP_CC_DEVICE_ADDRESS`

- `DSP_CC_CLOCK_RATE`

- `DSP_CC_DEVICE_COUNT`

- `DSP_CC_DEVICE_ACTIVE`

The function returns DSP_OK on success, DSP_ERROR otherwise.

Example D-5 `dspd_cc_read_flag()`

```
/* Read Command Converter flag word */
#include "cc.h"
#include "simcom.h"
#include "driver.h"

int status, flag_value, devn;

devn=0;

status=dspd_cc_read_flag(devn, DSP_CC_XPTR, &flag_value);
```

D.3.6   `dspd_cc_read_memory`—Read from Command Converter Memory

```
#include "simcom.h"
#include "driver.h"
#include "cc.h"
int dspd_read_memory(device_index, mem_space, address, count, value)
int device_index;/* Command Converter affected by command */
int mem_space;/* Memory space to read */
unsigned long address;/* Address of first location to read */
unsigned long count;/* Number of locations to read */
unsigned long *value;/* Address of buffer to receive read values */
```

dspd_cc_read_memory() reads a block of memory starting at address `address`, length `count`, in memory space `mem_space` on Command Converter `device_index`. The values read are stored in the buffer pointed to by `value`.

Valid values for `mem_space` are:

- DSP_CC_PMEM

- DSP_CC_XMEM

- DSP_CC_YMEM

The return value is `DSP_OK` for success, `DSP_ERROR` if the transaction fails.

Example D-6 `dspd_cc_read_memory()`

```
/* Read memory block from Command Converter*/
#include "simcom.h"
#include "driver.h"
#include "cc.h"

int devn, status;
unsigned long read_addr, read_len;
unsigned long *read_buf;

devn=0;

read_buf = (unsigned long *)dsp_alloc(100*sizeof(unsigned long));
read_addr=0x0400l;
read_len=100l;

status=dspd_cc_read_memory(devn,DSP_CC_PMEM,read_addr,read_len,read_buf);
```

D.3.7 `dspd_cc_reset`—Reset Command Converter

```
#include "simcom.h"
#include "driver.h"
int dspd_cc_reset(device_index)
int device_index;
```

`dspd_cc_reset()` resets the Command Converter `device_index`.

The return value is `DSP_OK` for success, `DSP_ERROR` if the reset fails.

Example D-7 `dspd_cc_reset()`

```
/* Reset Command Converter */
#include "simcom.h"
#include "driver.h"

int devn, status;


devn=0;

status=dspd_cc_reset(devn);
```

D.3.8   dspd_cc_revision—Read Command Convertor Revision Number

```
#include "simcom.h"
#include "driver.h"
int dspd_cc_revision(device_index, revstring)
int device_index;/* Command Converter affected by command */
char *revstring;/* pointer to buffer to receive revision number string */
```

dspd_cc_revision() returns a text string containing the version number of the monitor for Command Converter device_index in the buffer pointed to by revstring.

The message is created with the format:

```
"Command Converter monitor revision {%4.2f}"
```

The return value is DSP_OK for success, DSP_ERROR if the reset fails.

Example D-8 dspd_cc_revision()

```
/* Read CC monitor revision number */
#include "simcom.h"
#include "driver.h"

int devn, status;
char *rev_buf;

devn=0;

rev_buf = (unsigned long *)dsp_alloc(100*sizeof(char));

status=dspd_cc_revision(devn,rev_buf);
```

D.3.9   `dspd_cc_write_flag`—Write Command Converter Flag Word

```
#include "cc.h"
#include "simcom.h"
#include "driver.h"
int dspd_cc_write_flag(device_index, flag, value)
int device_index;
int flag;
unsigned long value;
```

`dspd_cc_write_flag()` writes to flag word `flag` on Command Converter
`device_index`, fetching the value from the location pointed to by `value`.

Valid values for `flag` are:

- `DSP_CC_FLAGS`

- `DSP_CC_STATUS`

- `DSP_CC_XPTR`

- `DSP_CC_YPTR`

- `DSP_CC_DEVICE_ADDRESS`

- `DSP_CC_CLOCK_RATE`

- `DSP_CC_DEVICE_COUNT`

- `DSP_CC_DEVICE_ACTIVE`

The function returns `DSP_OK` on success, `DSP_ERROR` otherwise.

Example D-9 `dspd_cc_write_flag()`

```
/* Write Command Converter flags */
#include "cc.h"
#include "simcom.h"
#include "driver.h"

int devn, status;

devn=0;

status=dspd_cc_write_flags(devn,DSP_CC_YPTR,0x0400l);
```

D.3.10 `dspd_cc_write_memory`—Write to Command Converter Memory

```
#include "simcom.h"
#include "driver.h"
#include "cc.h"
int dspd_write_memory(device_index, mem_space, address, count, value)
int device_index;/* Command Converter affected by command */
int mem_space;/* Memory space to write */
unsigned long address;/* Address of first location to write */
unsigned long count;/* Number of locations to write */
unsigned long *value;/* Address of buffer of values to write */
```

dspd_cc_write_memory() writes a block of memory starting at address `address`, length `count`, in memory space `mem_space` on Command Converter `device_index`. The values read are retrieved from the buffer pointed to by `value`.

Valid values for `mem_space` are:

- `DSP_CC_PMEM`—program memory

- `DSP_CC_XMEM`—X data memory

- `DSP_CC_YMEM`—Y data memory

The return value is `DSP_OK` for success, `DSP_ERROR` if the transaction fails.

Example D-10 `dspd_cc_write_memory()`

```
/* Write memory block to Command Converter*/
#include "simcom.h"
#include "driver.h"
#include "cc.h"

int devn, status;
unsigned long write_addr, write_len;
unsigned long *write_buf;

devn=0;

write_buf = (unsigned long *)dsp_alloc(100*sizeof(unsigned long));

get_values_in(write_buf, 100l)/* fetch values to write... */

write_addr=0x0400l;
write_len=100l;

status=dspd_cc_write_memory(devn,DSP_CC_PMEM,write_addr,write_len,write_buf);
```

D.3.11 `dspd_check_service_request`—Check for Service Request

```
#include "simcom.h"
#include "driver.h"
int dspd_check_service_request(device_index)
int device_index;/* DSP index affected by command */
```

`dspd_check_service_request()` checks to see if the target device `device_index` is requesting service from the host computer

The return value is TRUE if the device is requesting service, FALSE if it is not requesting service, and DSP_ERROR if the function cannot complete successfully.

Example D-11   `dspd_check_service_request()`

```
/* Check to see if a DSP device is requesting service */
#include "simcom.h"
#include "driver.h"

int devn;
int status;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

.....
                /* wait for device to request service */
while ((status=dspd_check_service_request(devn))==FALSE);

if (status==DSP_ERROR)
   return(DSP_ERROR);

/* device requested service. Now find out why..... */
```

D.3.12 `dspd_fill_memory`—Initialize DSP Memory Buffer to Single Value

```
#include "simcom.h"
#include "driver.h"
int dsp_fill_memory(device_index,mem_space,address,count,value)
int device_index;/* index of DSP device affected by command */
int mem_space;/* memory space to fill */
unsigned long address;/* address of start of memory block */
unsigned long count;/* length of memory block */
unsigned long value;/* fill value */
```

`dspd_fill_memory()` initializes a block of memory on the specified DSP device `device_index` to the specified value. `count` locations starting at `address` are filled with `value`.

Valid values for `mem_space` are:

- `P_MEM`—program memory

- `X_MEM`—X data memory

- `Y_MEM`—Y data memory

The return value is `DSP_OK` for success, `DSP_ERROR` if the transaction fails.

Example D-12 `dspd_fill_memory()`

```
/* clear P memory on DSP device 1 */
#include "simcom.h"
#include "driver.h"

int devn;
int status;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */
              /* init buffer to 'no value' */
status=dsp_fill_memory(devn,X_MEM,0x0100l,0x144l,0xffffffl)
```

D.3.13 `dspd_go`—Begin Execution on Target DSP Device

```
#include "simcom.h"
#include "driver.h"
int dspd_go(device_index, opcode, operand)
int device_index;/* Index of DSP device affected by command */
unsigned long opcode;/* values to load into pipeline */
unsigned long operand;/* before starting program execution */
```

`dspd_go()` is called to start program execution on the target device `device_index`.

The target device's pipeline is loaded with the parameters `opcode` and `operand`, and the device is forced to start executing.

The values loaded into the pipeline via `opcode` and `operand` should either be the values saved from the pipeline when the device entered debug mode, or if execution is required to continue from a specific address, the values loaded should form a long jump instruction to the required execution start address:

opcode: opcode for long jump to required execution start address.
              Symbolic names are defined for JUMP opcodes for the device
              families in simcom.h.

operand: address of long jump target (execution start address)

The function returns `DSP_OK` on successful completion, `DSP_ERROR` otherwise.

Example D–13   dspd_go()

```
/* Start execution from address 0x1000 */
#include "simcom.h"
#include "protocom.h"
#include "driver.h"

int err, status, devn;

devn=0;

err=dsp_load("lunchbrk.adm");/* reload devices and program data */

status=dspd_go(devn, DSP_LONGJUMP_56K, 0x1000l); /* & execute from 0x1000 */
```

D.3.14 `dspd_jtag_reset`—Reset JTAG Communications

```
#include "simcom.h"
#include "driver.h"
int dspd_jtag_reset(device_index, reset_type)
int device_index;/* device affected by command */
int reset_type;/* type of reset to perform */
```

`dspd_jtag_reset()` resets the JTAG TAP controller for the device `device_index`.

`reset_type` may be set to:

- `DSP_JTAG_RESET_HARDWARE`—Force reset by asserting trst pin on JTAG port

- `DSP_JTAG_RESET_SOFTWARE`—Force reset by toggling tms pin on JTAG port until the JTAG TAP controller state machine returns to its reset state.

The function returns `DSP_OK` if the operation succeeds, `DSP_ERROR` otherwise.

Example D-14 `dspd_jtag_reset()`

```
/* reset JTAG communications with */
#include "simcom.h"
#include "driver.h"

int devn;
int status;

devn=0;
dsp_new(devn,"96002");/* Allocate structure for device 0, a 96002 */

status=dspd_jtag_reset(devn, DSP_JTAG_RESET_HARDWARE)
```

D.3.15 `dspd_read_core_registers`—Read Core Registers from DSP Device

```
#include "adsreg56.h"
#include "simcom.h"
#include "driver.h"
int dspd_read_core_registers(device_index, reg_num, count, value)
int device_index;/* Index of DSP device affected by command */
int reg_num;/* First register to read */
unsigned long count;/* Number of registers to read */
unsigned long *value;/* Pointer to area to receive register values */
```

`dspd_read_core_registers()` reads `count` core registers starting at register `reg_num` for target device `device_index`, storing the values in the memory pointed to by `value`.

The order of the registers is specified in the header file adsregXX.h

Note: Calling order is important. `dspd_read_once_registers()` must be called before calling `dspd_read_core_registers()`. If the calling order is reversed, the values in the OnCE registers will have been altered.

Some registers, for example the `a` register, may be too large to be held in a single location. These registers are also defined as a number of smaller registers, `a0`, `a1`, and `a2`. Such registers require an element in the value array for the compound register, and one for each of its component parts. The value returned for the compound register is undefined. Each of the component values is returned, and must be assembled by the calling program if the value of the compound register is required. The return value is `DSP_OK` if the operation succeeds, `DSP_ERROR` if it fails.

Example D-15 `dspd_read_core_registers()`

```
/* Read core registers */
#include "adsreg56.h"
#include "simcom.h"
#include "driver.h"

int devn, status;
unsigned long count_once, buf_once[15], count_core, buf_core[ADSCOREMAX];
devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */
count_once=15; /* all OnCE registers */
count_core = ADSCOREMAX;/* all core registers */
                /* get OnCE first, then core */
status=dspd_read_once_registers(devn, OSCR, count_once, buf_once);
status=dspd_read_core_registers(devn, ADS_A, count_core, buf_core);
```

D.3.16   `dspd_read_memory`—Read Memory Block from DSP Device

```
#include "simcom.h"
#include "driver.h"
int dspd_read_memory(device_index, mem_space, address, count, value)
int device_index;/* Index of DSP device affected by command */
int mem_space;/* Memory space to read */
unsigned long address;/* Address of first location to read */
unsigned long count;/* Number of locations to read */
unsigned long *value;/* Pointer to area to receive memory values */
```

`dspd_read_memory()` reads `count` words of memory from the DSP device in memory space `mem_space` starting at address `address`, and stores them in the memory pointed to by `value`.

Valid values for `mem_space` are:

- `P_MEM`—program memory

- `X_MEM`—X data memory

- `Y_MEM`—Y data memory

The return value is set to `DSP_OK` if the operation succeeds, `DSP_ERROR` if it fails.

Example D-16   `dspd_read_memory()`

```
/* Read DSP memory block */
#include "simcom.h"
#include "driver.h"

int devn, status;
unsigned long x_mem_buf[0x100];

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

.....
            /* Read back work buffer */
status = dspd_read_memory(devn, X_MEM, 0x00001, 0x01001, x_mem_buf)
```

D.3.17 `dspd_read_once_registers`—Read OnCE Registers from DSP Device

```
#include "adsreg56.h"
#include "simcom.h"
#include "driver.h"
int dspd_read_once_registers(device_index, reg_num, count, value)
int device_index;/* Index of DSP device affected by command */
int reg_num;/* First once register to read */
unsigned long count;/* Number of registers to read */
unsigned long *value;/* Pointer to area to receive register values */
```

`dspd_read_once_registers()` reads `count` OnCE registers starting from register `reg_num` from target device `device_index` and stores the values in the memory pointed to by `value`.

The order of the registers is specified in the header file adsregXX.h.

Note: Calling order is important. `dspd_read_once_registers` must be called before calling `dspd_read_core_registers`. If the calling order is reversed, the values in the OnCE registers will have been altered.

The function returns `DSP_OK` on success, `DSP_ERROR` otherwise.

Example D-17 `dspd_read_once_registers()`

```
/* Read once registers */
#include "adsreg56.h"
#include "simcom.h"
#include "driver.h"

int devn, status;
unsigned long count_once, buf_once[15], count_core, buf_core[ADSCOREMAX];

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

count_once=15;
count_core = ADSCOREMAX;

status=dspd_read_once_registers(devn, OSCR, count_once, buf_once);
status=dspd_read_core_registers(devn, ADS_A, count_core, buf_core);
```

D.3.18   `dspd_reset`—Reset DSP Device to Debug or User Mode

```
#include "simcom.h"
#include "driver.h"
int dspd_reset(device_index, reset_mode)
int device_index;/* Index of affected DSP device */
int reset_mode;/* type of reset to perform */
```

`dspd_reset()` resets the target device `device_index`. The device may be reset into debug mode or user mode, based on the value of `reset_mode`.

Valid values for `reset_mode` are:

- `DSP_RESET_DEBUG`—reset device into debug mode

- `DSP_RESET_USER`—reset device into user mode

The function returns `DSP_OK` on success, `DSP_ERROR` otherwise.

Example D-18   dspd_reset()

```
/* Place DSP device 3 into debug mode */
#include "simcom.h"
#include "driver.h"

int devn, status;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */


status = dspd_reset(devn, DSP_RESET_DEBUG);/* Reset device into DEBUG mode */
```

D.3.19 `dspd_status`—Determine DSP Status

```
#include "driver.h"
#include "simcom.h"
int dspd_status(device_index, mode)
int device_index;/* DSP device affected by command */
int *mode; /* address of buffer to receive device status */
```

`dspd_status()` places the execution status of the target device device_index in the int pointed to by mode.

Valid values for *mode are:

- `DSP_USER_MODE`—device is executing a user program

- `DSP_DEBUG_MODE`—device is in debug mode

The function returns TRUE on success, or FALSE otherwise.

Example D-19 `dspd_status()`

```
/* determine status of DSP 0 */
#include "driver.h"
#include "simcom.h"

int devn, status;
int device_mode;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */


status = dspd_status(devn, &device_mode);/* get device status */
```

D.3.20 `dspd_write_core_registers`—Write Core Registers to DSP Device

```
#include "adsreg56.h"
#include "simcom.h"
#include "driver.h"
int dspd_write_core_registers(device_index, reg_num, count, value)
int device_index;/* Index of DSP device affected by command */
int reg_num;/* First once register to write */
unsigned long count;/* Number of register words to write */
unsigned long *value;/* Pointer to area holding register values */
```

`dspd_write_core_registers()` writes `count` core registers starting at `reg_num` to target device `device_index`.The values written are taken from the memory pointed to by `value`.

The order of the registers is specified in the header file adsregXX.h

Some registers, for example the `a` register, may be considered to be an entity in its own right, or a number of smaller registers, that is `a0`, `a1`,and `a2`. Such registers require an element in the value array for the compound register, and one for each of its component parts. The value for the compound register is ignored. Each of the component values is loaded, thereby changing the value of the compound register.

The function returns `DSP_OK` on success, `DSP_ERROR` otherwise.

Example D-20 `dspd_write_core_registers()`

```
/* Write core registers */
#include "adsreg56.h"
#include "simcom.h"
#include "driver.h"

int devn, status;
unsigned long core_reg_values[15];

devn=0;
dsp_new(devn,"56002");/* Allocate structures for device 0, a 56002 */
.
.
.               /* write all core registers from buffer */
status=dspd_write_core_registers(devn, ADS_A, 15l, core_reg_values)
```

D.3.21   `dspd_write_memory`—Write to Memory in DSP Device

```
#include "simcom.h"
#include "driver.h"
int dspd_write_memory(device_index, mem_space, address, count, value)
int device_index;/* Index of DSP device affected by command */
int mem_space;/* Memory space to write */
unsigned long address;/* Address of first location to write */
unsigned long count;/* Number of locations to write */
unsigned long *value;/* Pointer to area holding memory values */
```

`dspd_write_memory()` writes `count` words of memory in the DSP device. The values
are taken from the buffer pointed to by `value`.

Valid values for `mem_space` are:

- `P_MEM`—program memory

- `X_MEM`—X data memory

- `Y_MEM`—Y data memory

The return value is set to `DSP_OK` if the operation succeeds, `DSP_ERROR` if it fails.

Example D-21 `dspd_write_memory()`

```
/* Write DSP memory block */
#include "simcom.h"
#include "driver.h"

int devn, status;
unsigned long x_mem_buf[0x100];

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

status = dspd_write_memory(devn, X_MEM, 0x0400l, 0x0100l, &x_mem_buf[0])
```

D.3.22 `dspd_write_once_registers`—Write OnCE Registers to DSP Device

```
#include "adsreg56.h"
#include "simcom.h"
#include "driver.h"
int dspd_write_once_registers(device_index, reg_num, count, value)
int device_index;/* Index of DSP device affected by command */
int reg_num;/* First once register to write */
unsigned long count;/* Number of registers to write */
unsigned long *value;/* Pointer to area holding register values */
```

dspd_write_once_registers() writes count OnCE registers starting at register reg_num to the target device device_index. The values written are taken from the memory pointed to by value.

The order of the registers is specified in the header file adsregXX.h.

The function returns DSP_OK on success, DSP_ERROR otherwise.

Example D-22 `dspd_write_once_registers()`

```
/* Write once registers */
#include "adsreg56.h"
#include "simcom.h"
#include "driver.h"

int devn, status;
unsigned long once_reg_values [15];

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */
.
.
.                /* write all OnCE registers from buffer */
status=dspd_write_once_registers(devn, OSCR, 15l, once_reg_values)
```

D.3.23 `dspt_masm_xxxxx`—Assemble DSP Mnemonic

```
#include "proto56n.h"/* n= k,1,3,8 */
#include "proto96k.h"
int dspt_masm_xxxxx(mnemonic,ops,error_ptr)
char *mnemonic;/* Pointer to assembler mnemonic string */
unsigned long *ops;/* Array for words of assembled code */
char **error_ptr;/* Will point to message if an error occurs */
```

`dspt_masm_xxxxx()` 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 `error_ptr` 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 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 placed in `ops[0]`, the second in `ops[1]`, the third in `ops[2]`.

Note: The xxxxx in the function name should be replaced by a device family number. It should be 56k for the 56000 family devices, 56n00 for the 56n00 family devices and 96k for the 96000 family devices.

Example D-23 `dspt_masm_xxxxx()`

```
/* Assemble the instruction "move r0,r1" */
#include "proto56k.h"

unsigned long opcodes[3];
char *error_ptr;
int retval;

retval=dspt_masm_56k("move r0,r1",&opcodes[0],&error_ptr);
```

D.3.24 `dspt_unasm_xxxxx`—Disassemble DSP Mnemonics

```
#include "proto56n.h"/* n= k,1,3,8 */
#include "proto96k.h"
int dspt_unasm_xxxxx(ops,return_string,sr,omr,gdbp)
unsigned long *ops;/* Pointer to opcodes to be disassembled */
char *return_string;/* Pointer to return character buffer */
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*/
```

`dspt_unasm_xxxxx()` 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: The xxxxx in the function name should be replaced by a device family number. It should be 56k for the 56000 family devices, 56n00 for the 56n00 family devices, and 96k for the 96000 family devices.

Example D-24 `dspt_unasm_xxxxx()`

```
/* Disassembly of the opcode representing NOP */
#include "proto56n.h"

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" */
```

D.3.25 `dsp_alloc`—Allocate Memory

The routines `dsp_alloc`, `dsp_free_mem` and `dsp_realloc` are replacements for the standard C functions `malloc`, `free` and `realloc`. They are used in much the same way as the standard functions, for allocating space for structures, buffers, etc. These functions are used in the debugger libraries, and must also be used exclusively in the user debugger code. Any attempt to use the standard routines will have unpredictable results.

```
#include "simcom.h"
#include "protocom.h"
void *dsp_alloc(nbytes,clearmem)
unsigned int nbytes;/* Number of bytes to allocate */
int clearmem;/* Clear allocated memory */
```

`dsp_alloc()` allocates the number of bytes of memory specified in `nbytes`. The memory block allocated is aligned for use as any data type. If `clearmem` is true (nonzero), the allocated memory is cleared to zero.

The address of the allocated buffer is returned as the return value, type `void*`.

If the requested memory cannot be allocated, the error message `"Insufficient memory: dsp_alloc"` is output, and the return value is the `NULL` pointer.

Example D-25 `dsp_alloc()`

```
/* Allocate temporary buffer of 50 int. Buffer is cleared. */
#include "simcom.h"
#include "protocom.h"

int *tbptr

tbptr = (int *) dsp_alloc(50*sizeof(int), 1)
if (tbptr == NULL)
{...handle error...
```

```
}
```

D.3.26 `dsp_cc_fmem`—Fill Command Converter Memory with a Value

```
#include "simcom.h"
#include "protocom.h"
#include "cc.h"
#include "coreaddr.h"
int dsp_cc_fmem(device, mtype, address, count, value)
int device;            /* Command Converter affected by command */
enum memory_map mtype;/* Memory space to fill */
unsigned long address;/* Address of start of memory block */
unsigned long count;   /* Length of memory block */
unsigned long *value; /* Fill value */
```

`dsp_cc_fmem()` initializes a block of memory in the Command Converter device starting at address `address`, length `count`, in address space `mtype` with the value pointed to by `value`.

Valid values for `mtype` are:

- `DSP_CC_YMEM`

- `DSP_CC_XMEM`

- `DSP_CC_PMEM`

The return value is `TRUE` for success, `FALSE` otherwise. If the return value is `FALSE`, then some but not all of the specified locations may have been changed.

Example D-26 `dsp_cc_fmem()`

```
/* Initialize CC statistics buffer */
#include "simcom.h"
#include "protocom.h"
#include "cc.h"
#include "coreaddr.h"

int devn, fill_value, status;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

fill_value=0;

dsp_cc_fmem(devn, DSP_CC_XMEM, 0xe410l, 0x20l, &fill_value)
```

D.3.27   `dsp_cc_go`—Start Command Converter Program Execution

```
#include "simcom.h"
#include "protocom.h"
int dspd_cc_go(device_index)
int device_index;
```

`dsp_cc_go()` starts the Command Converter for the target device `device_index` executing from the address indicated by the current program counter.

The return value is `TRUE` for success, `FALSE` otherwise.

Example D-27 `dsp_cc_go()`

```
/* Start CC running */
#include "simcom.h"
#include "protocom.h"

int devn, status;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

status = dspd_cc_go(devn);/* Start Command Converter monitor */
```

D.3.28   `dsp_cc_ldmem`—Load Command Converter Memory from File

```
#include "simcom.h"
#include "protocom.h"
int dsp_cc_ldmem(device_index, loadfn)
int device_index;        /* Command Converter affected by command */
char *loadfn;            /* Name of file to be loaded */
```

`dsp_cc_ldmem()` loads memory in the Command Converter `device_index` from the file `loadfn`. This is a lower-level routine which does not call the user-level filename routines; `loadfn` must contain the fully-specified name of the file to be opened. The file is OMF format, the file extension should be .lod

The return value is `TRUE` on successful completion, `FALSE` otherwise.

Example D-28   `dsp_cc_ldmem()`

```
/* Load Command Converter memory from file */
#include "simcom.h"
#include "protocom.h"

int devn, status;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

status = dsp_cc_ldmem(devn,"c:\dspdev\bin\loadfile.lod");
```

D.3.29   `dsp_cc_reset`—Reset Command Converter

```
#include "simcom.h"
#include "protocom.h"
int dsp_cc_reset(device)
int device;              /* CC device affected by command */
```

`dsp_cc_reset()` resets the Command Converter `device`. Function `dspd_cc_architecture()` is called to configure the Command Converter for the type of DSP device attached.

This procedure may be called as part of the initialization procedures to guarantee the Command Converter is in a known state, to recover from a lockup, or at any other time when the Command Converter needs to be restarted.

The return value is TRUE if the operation succeeds, FALSE otherwise.

Example D-29 `dsp_cc_reset()`

```
/* reset Command Converter */
#include "simcom.h"
#include "protocom.h"

int devn, status;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

status= dsp_cc_reset(devn);/* reset the Command Converter for device 0 */
```

D.3.30 `dsp_cc_revision`—Read Command Converter Monitor Revision

```
#include "simcom.h"
#include "protocom.h"
int dspd_cc_revision(device_index, revstring)
int device_index;/* Command Converter affected by command */
char *revstring;/* receives CC monitor revision string */
```

`dsp_cc_revision()` interrogates the Command Converter `device_index` to determine the monitor revision, and returns a formatted string in `revstring` containing the monitor revision.

The format used to create the revstring is:

```
"Command Converter monitor revision {%4.2f}"
```

The return value is TRUE on successful completion, FALSE otherwise.

Example D-30 `dsp_cc_revision()`

```
/* obtain Command Converter monitor revision number */
#include "simcom.h"
#include "protocom.h"

int devn, status;
char revision_string[80];

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

status = dspd_cc_revision(devn, revision_string);
```

D.3.31 `dsp_cc_rmem`—Read Command Converter Memory

```
#include "simcom.h"
#include "protocom.h"
#include "cc.h"
#include "coreaddr.h"
int dsp_cc_rmem(device, mtype, address, value)
int device;/* Command Converter affected by command */
enum memory_map mtype;/* Memory space to read */
unsigned long address;/* Address of location to read */
unsigned long *value;/* Target location for read value */
```

`dsp_cc_rmem()` reads one location from the Command Converter `device`, memory space `mtype`, address `address`, and stores the value in the location pointed to by `value`.

Valid values for mtype are:

- `DSP_CC_YMEM`

- `DSP_CC_XMEM`

- `DSP_CC_PMEM`

The return value is `TRUE` on successful completion, `FALSE` otherwise.

Example D-31 `dsp_cc_rmem()`

```
/* Read location from Command Converter memory */
#include "simcom.h"
#include "protocom.h"
#include "cc.h"
#include "coreaddr.h"

int devn, status;
enum memory_map read_memtyp,/* memory space to read */
unsigned long read_address,/* address of location to read */
            read_store;/* location to hold read value */

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

read_memtyp = DSP_CC_XMEM;/* set memory space for read */
read_address=0x0040l;/* set read address */
            /* read required location */
status = dsp_cc_rmem(devn, read_memtyp, read_address, &read_store);
```

D.3.32 `dsp_cc_rmem_blk`—Read Command Converter Memory Block

```
#include "simcom.h"
#include "protocom.h"
#include "cc.h"
#include "coreaddr.h"
int dsp_cc_rmem_blk(device, mtype, address, count, value)
int device;/* Command Converter affected by command */
enum memory_map mtype;/* Memory space to read */
unsigned long address;/* Address of location to read */
unsigned long count/* number of locations to read */
unsigned long *value;/* Target location for read value */
```

`dsp_cc_rmem_blk()` reads `count` locations from the target Command Converter `device`, memory space `mtype`, address `address`, and stores the values in the buffer pointed to by `value`.

Valid values for mtype are: `DSP_CC_YMEM`, `DSP_CC_XMEM`, and `DSP_CC_PMEM`.

The return value is `TRUE` on successful completion, `FALSE` otherwise.

Example D-32 `dsp_cc_rmem_blk()`

```
/* Read memory block from Command Converter memory */
#include "simcom.h"
#include "protocom.h"
#include "cc.h"
#include "coreaddr.h"

int devn, status;

enum memory_map read_memtyp;/* memory space to read */
unsigned long read_address,/* address of first location to read */
              read_length,/* number of locations to read */
              read_store[1024];/* buffer to hold read values */
devn=0;      /* set device number */
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

read_memtyp = DSP_CC_XMEM;/* set memory space */
read_addr=0x0040l;/* start address */
read_length=55l;/* and block length */
              /* now read the memory block into read_store */
status = dsp_cc_rmem_blk(devn, read_memtyp, read_addr, read_length, read_store);
```

D.3.33 `dsp_cc_wmem`—Write Command Converter Memory

```
#include "simcom.h"
#include "protocom.h"
#include "cc.h"
#include "coreaddr.h"
int dsp_cc_wmem(device, mtype, address, value)
int device;/* Command Converter affected by command */
enum memory_map mtype;/* Memory space to write */
unsigned long address;/* Address of location to write */
unsigned long *value;/* Source location for value to write */
```

`dsp_cc_wmem()` writes one location in the Command Converter `device`, memory space `mtype`, address `address`, using the value in the location pointed to by `value`.

Valid values for `mtype` are:

- DSP_CC_YMEM

- DSP_CC_XMEM

- DSP_CC_PMEM

The return value is TRUE on successful completion, FALSE otherwise.

Example D-33 `dsp_cc_wmem()`

```
/* Write to a location in Command Converter memory */
#include "simcom.h"
#include "protocom.h"
#include "cc.h"
#include "coreaddr.h"

int devn, status;
enum memory_map write_memtyp,/* memory space to write */
unsigned long write_address,/* address of location to write */
            write_store;/* location to hold value to write */

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

write_memtyp = DSP_CC_XMEM;/* set memory space for write */
write_address=0x0040l;/* set write address */
            /* write required location */
status = dsp_cc_wmem(devn, write_memtyp, write_address, &write_store);
```

D.3.34 `dsp_cc_wmem_blk`—Write Command Converter Memory Block

```
#include "protocom.h"
#include "cc.h"
#include "coreaddr.h"
int dsp_cc_wmem_blk(device, mtype, address, count, value)
int device;/* Command Converter affected by command */
enum memory_map mtype;/* Memory space to write */
unsigned long address;/* Address of location to write */
unsigned long count/* number of locations to write */
unsigned long *value;/* Source buffer for write values */
```

`dsp_cc_wmem_blk()` writes `count` locations in the Command Converter `device`, memory space `mtype`, address `address`, and obtaining the values from the buffer pointed to by `value`.

Valid values for mtype are: DSP_CC_YMEM, DSP_CC_XMEM, and DSP_CC_PMEM.

The return value is TRUE on successful completion, FALSE otherwise.

Example D-34 `dsp_cc_wmem_blk()`

```
/* Write several locations from Command Converter memory */
#include "simcom.h"
#include "protocom.h"
#include "cc.h"
#include "coreaddr.h"

int devn, status;

enum memory_map write_memtyp;/* memory space to write */
unsigned long write_addr,/* address of first location to write */
              write_length,/* number of locations to write */
              write_store[1024];/* buffer holding values to write*/

devn=0;        /* set device number */
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

write_memtyp = DSP_CC_XMEM;/* set memory space */
write_addr=0x0040l;/* start address */
write_length=55l;/* and block length */
               /* now write the memory block to device 0*/
status=dsp_cc_wmem_blk(devn, write_memtyp, write_addr, write_length, write_store);
```

D.3.35   `dsp_check_service_request`—Check for Service Request

```
#include "simcom.h"
#include "protocom.h"
int dsp_check_service_request(device_index)
int device_index;/* device affected by operation */
```

`dsp_check_service_request()` checks to see if the target device `device_index` is requesting service from the host computer, that is, checks to see if the target device is in Debug Mode.

The return value is `TRUE` if the device is requesting service, `FALSE` if it is not, and `DSP_ERROR` if the function cannot complete successfully.

Example D-35 `dsp_check_service_request()`

```
/* Check if device 0 is requesting service */
#include "simcom.h"
#include "protocom.h"

int devn = 0;
int ok;

dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

ok = dsp_cc_reset(devn);/* reset Command Converter */

ok = dsp_reset(devn);/* and device 0 to debug mode*/

ok = dsp_check_service_request(devn)/* Is device 'devn' requesting service? */
```

D.3.36 `dsp_findmem`—Get Map Index for Memory Prefix

```
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"
dsp_findmem(device_index,memory_name,memory_map)
int device_index;       /* DSP device to be affected by command */
char *memory_name;      /* memory space name */
enum memory_map *memory_map;/* return memory map type */
```

dsp_findmem() searches the dt_var.mem structure for device device_index 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 emulator help mem command.

Example D–36   dsp_findmem()

```
/* Get map index for memory space "P" */
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"

int devn;
enum memory_map map;
int ok;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

ok=dsp_findmem(devn,"P",&map)/* Get memory map index for "P" memory */
```

D.3.37 `dsp_findreg`—Get Peripheral and Register Index

```
#include "simcom.h"
#include "protocom.h"
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 */
```

`dsp_findreg()` 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 emulator "help reg" command to obtain a list of the valid `periph_num` and `reg_num` values, and `reg_val` size for each register.

Example D-37   `dsp_findreg()`

```
/* Get peripheral index and register number for register 'n3' */
#include "simcom.h"
#include "protocom.h"

int devn;
int regnum, pnum;
int ok;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

ok=dsp_findreg(devn,"n3",&pnum,&regnum);/* Get index for "n3" register */
```

D.3.38 `dsp_fmem`—Fill Memory Block with a Value

```
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"
dsp_fmem(device_index,memory_map,address,block_size,value)
int device_index;      /* DSP device 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 */
```

`dsp_fmem()` initializes a block of DSP memory with a single value.

The `memory_map` parameter is a memory type that selects the appropriate `dt_memory` structure from `dt_var.mem` for the selected device `device_index`. These structures are described in the simdev.h file which is included with the emulator. The `memory_map` parameter can be obtained with the function `dsp_findmem()` by using the memory name as a key. Use the emulator help mem command for a list of valid memory names. The `memory_map` enum is `memory_map_` concatenated with a valid memory map 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.

Example D-38 `dsp_fmem()`

```
/* Write 300 locations beginning at P:$200 with the value 4 */
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"

int devn;
unsigned long address, memval, blocksize;

address=0x200L;
blocksize=300;
memval=4L;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

dsp_fmem(devn,memory_map_p,address,blocksize,&memval);
```

D.3.39 `dsp_free`—Free a Device Structure

```
#include "simcom.h"
#include "protocom.h"
dsp_free(device_index)
int device_index;      /* DSP device index to be affected by command */
```

`dsp_free()` frees all allocated memory associated with a device structure for `device_index`, and closes any open files associated with the device structure.

Example D-**39** `dsp_free()`

```
/* Create three new device structures, then get rid of device 2. */
#include "simcom.h"
#include "protocom.h"

ads_startup("100",ADSP56000);
dsp_startup();

dsp_new(0,"56002");/* Allocate structure for device 0, a 56002 */
dsp_new(1,"56002");/* Allocate structure for device 1, a 56002 */
dsp_new(2,"56002");/* Allocate structure for device 2, a 56002 */

dsp_free(1);   /* Free structure for device 1 */
```

D.3.40   `dsp_free_mem`—Free Memory Block

The routines `dsp_alloc`, `dsp_free_mem` and `dsp_realloc` are replacements for the standard C functions `malloc`, `free` and `realloc`. They are used in much the same way as the standard functions, for allocating space for structures, buffers, etc. These functions are used in the debugger libraries, and must also be used exclusively in the user debugger code. Any attempt to use the standard routines will have unpredictable results.

```
#include "simcom.h"
#include "protocom.h"
void dsp_free_mem(cp)
char *cp    /* pointer to memory block to be freed */
```

`dsp_free_mem()` releases a memory block previously allocated with `dsp_alloc()`. Its argument `cp` is the address of the data block.

Example D-40 `dsp_free_mem()`

```
/* Allocate and release memory block */
#include "simcom.h"
#include "protocom.h"

int *buffer;
int i;
buffer = (int *) dsp_alloc(sizeof(int)*10, 0)

/* Use the buffer */

for (i=0; i<10; i++)
   buffer[i]++;

/* And discard it */
dsp_free_mem((char*)buffer);
```

D.3.41   `dsp_go`—Initiate DSP Program Execution

```
#include "simcom.h"
#include "protocom.h"
int dsp_go(device_index)
int device_index;/* DSP device to start executing */
```

`dsp_go()` starts the target device `device_index` to begin executing, in real time, from the address specified by the current program counter.

The function returns `TRUE` on successful completion, `FALSE` otherwise.

Example D-41 `dsp_go()`

```
/* start DSP 0 executing */
#include "simcom.h"
#include "protocom.h"

int devn, status;
char *load_fname="filter2.cld";

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

status = dsp_ldmem(devn,load_fname);/* load program into memory */

status = dsp_go(devn);/* start program execution */
```

D.3.42 `dsp_go_address`—Initiate Program Execution from Address

```
#include "simcom.h"
#include "protocom.h"
dsp_go_address(device_index, address)
int device_index;/* DSP device affected by command */
unsigned long address;/* address from which to start executing */
```

`dsp_go_address()` causes the target device `device_index` to begin executing, in real time, from the address specified by `address`.

The function returns TRUE on success, FALSE otherwise.

Example D-42   dsp_go_address()

```
/* Start execution from specified address */
#include "simcom.h"
#include "protocom.h"

int devn, status;
char *load_fname="filter2.cld";

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

status=dsp_ldmem(devn,load_fname);/* load program into memory */

status=dsp_go_address(devn,0x1400l);/* start program execution */
```

D.3.43 `dsp_go_reset`—Initiate Program Execution after Device Reset

```
#include "simcom.h"
#include "protocom.h"
dsp_go_reset(device_index)
int device_index;/* DSP device affected by command */
```

`dsp_go_reset()` causes the target device specified by `device_index` to be reset into User Mode. Execution starts at the reset value for pc.

To place a device into debug mode, see `dsp_reset()`.

The function returns TRUE on success, FALSE otherwise.

Example D-43   dsp_go_reset()

```
/* Initiate program execution by reset */
#include "simcom.h"
#include "protocom.h"

int devn, status;
char *load_fname="filter2.cld";

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

status=dsp_ldmem(devn,load_fname);/* load program into memory */

status=dsp_go_reset(devn);/* start program execution after device reset */
```

D.3.44   `dsp_init`—Initialize a Single DSP Device Structure

```
#include "simcom.h"
#include "protocom.h"
dsp_init(device_index)
int device_index;      /* DSP device index to be affected by command */
```

`dsp_init()` initializes a device `device_index` to the same state that existed following the `dsp_new()` call which created it. It is equivalent to performing the emulator FORCE S command. All memory spaces are cleared, the registers are reset, breakpoints and input/output file assignments are cleared.

Example D-**44** `dsp_init()`

```
/* re-initialize a device */
#include "simcom.h"
#include "protocom.h"

dsp_new(0,"56002");/* Create new DSP structure */
.
.
.
dsp_init(0);   /* Re-initialize device 0 */
```

D.3.45 `dsp_ldmem`—Load DSP Memory from OMF or COFF File

```
#include "simcom.h"
#include "protocom.h"
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 */
```

`dsp_ldmem()` loads the memory space of a specified DSP device `device_index` from an object file `filename`. The file may be created as the output from the DSP Macro Assembler, or by using the Emulator 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 D-45 `dsp_ldmem()`

```
/* Create DSP device structures for a three device emulation. */
#include "simcom.h"
#include "protocom.h"

int devn;
int err;

for (devn=0;devn<3;devn++)
  dsp_new(devn,"56002");/* Create new DSP structures */

/* Load device 1 with a program named filter2.lod.*/

err=dsp_ldmem(1,"c:\p42\bin\filter2.lod");
```

D.3.46 `dsp_load`—Load All DSP Structures from State File

```
#include "simcom.h"
#include "protocom.h"
int dsp_load(filename)
char *filename;/* Full name of State File to be loaded */
```

`dsp_load()` loads the emulator state of all devices from a specified emulator state file `filename`. 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 ".adm" suffix extension; the entire filename must be specified.

dsp_load returns `1` if the load was successful, `0` if there was an error.

Example D-46 `dsp_load()`

```
/* Load the emulator state from the file 'lunchbreak.adm' */
#include "simcom.h"
#include "protocom.h"

int err;
ads_startup("100",ADSP56000);
dsp_startup();

err=dsp_load("lunchbrk.adm");
```

D.3.47 `dsp_new`—Create New DSP Device Structure

```
#include "simcom.h"
#include "protocom.h"
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 */
```

dsp_new() creates a new DSP structure that represents a DSP device device_index, 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 D-47 `dsp_new()`

```
/* Create DSP device structures for a three device emulation. */
#include "simcom.h"
#include "protocom.h"

int devn;
ads_startup("100",ADSP56000);
dsp_startup();

for (devn=0;devn<3;devn++)
  dsp_new(devn,"56002");/* Create new DSP structures */
```

D.3.48   `dsp_path`—Construct Filename

```
#include "simcom.h"
#include "protocom.h"
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 */
```

`dsp_load()` constructs a fully-specified path and filename from the user-provided `path_name`, `base_name` and `suffix`. The constructed filename is returned in `new_name`.

If `base_name` begins with a pathname separator or with a device designator, `path_name` will not be prefixed to `base_name`. If `base_name` already ends with "`.`" and some filename extension, the string in `suffix` will not be appended.

Example D-48 `dsp_load()`

```
/* Load file filter2.lod from the current working directory for device 0. */
#include "simcom.h"
#include "protocom.h"
#include "simdev.h"

extern struct dev_const dv_const;/* emulator device structures */
char newfn[80];

dsp_new(0,"56002");/* Create new DSP structure */

/* Create file name from: */
/* Path specified in device structure dv_const.sv[0]->pathwork */
/* created by "path..." command */
/* Base file name filter2 */
/* Filename suffix .lod. Note the "." is not explicitly specified.*/

dsp_path(dv_const.sv[0]->pathwork,"filter2","lod",newfn);

dsp_ldmem(0,newfn); /* Load file into DSP device 0 */
```

D.3.49 `dsp_realloc`—Reallocate Memory Block

The routines `dsp_alloc`, `dsp_free_mem` and `dsp_realloc` are replacements for the standard C functions `malloc`, `free` and `realloc`. They are used in the much same way as the standard functions, for allocating space for structures, buffers, etc. These functions are used in the debugger libraries, and must also be used exclusively in the user debugger code. Any attempt to use the standard routines will have unpredictable results.

```
#include "simcom.h"
#include "protocom.h"
void *dsp_realloc(mem_blk, nbytes)
char *mem_blk;/* Address of existing allocated block */
unsigned int nbytes;/* Required size of memory block */
```

`dsp_realloc()` changes the size of a memory block `mem_blk` previously allocated with `dsp_alloc()` to the specified size `nbytes`. The address of the reallocated block may not be the same as the address of the original block. The contents of the original block is preserved—either completely if the block size is increased, or to the end of the new block if the block size is reduced. If the requested block cannot be allocated, the original block is unchanged.

Example D-**49** `dsp_realloc()`

```
/* increase buffer size */
#include "simcom.h"
#include "protocom.h"

int status;
char *bufptr;

/* Allocate temporary buffer of 50 characters. Buffer is cleared. */

bufptr = (char *) dsp_alloc(50*sizeof(char), 1)
if (bufptr == NULL)
{...handle error...}
  .
  .
  .

/* increase buffer size to 82 characters, preserving contents */

bufptr = (char *) dsp_realloc(bufptr, 82*sizeof(char))
```

D.3.50 `dsp_reset`—Reset Specified DSP Device

```
#include "simcom.h"
#include "protocom.h"
int dsp_reset(device_index)
int device_index;/* Index of affected DSP device */
```

dsp_reset() resets the target device device_index into Debug mode. To reset a device into User mode, see dsp_go_reset().

The function returns TRUE on success, FALSE otherwise.

Example D-50 `dsp_reset()`

```
/* Place DSP device 3 into debug mode */
#include "simcom.h"
#include "protocom.h"

int devn, status;


devn=3;

dsp_new(devn,"56002");/* Create new DSP structure */

            /* ensure device in known state */
status = dsp_reset(devn);/* Reset device into DEBUG mode */
```

D.3.51 `dsp_rmem`—Read DSP Memory Location

```
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"
int dsp_rmem(device_index,memory_map,address,return_value)
int device_index;      /* DSP device to be affected by command */
enum memory_map memory_map;/* memory designator */
unsigned long address;/* DSP memory address to read */
unsigned long *return_value;/* Returned memory value (or values) */
```

`dsp_rmem()` reads the contents of a selected DSP memory location specified by `memory_map` and `address`, 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`, and 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 emulator. The `memory_map` parameter can be obtained with the function `dsp_findmem()` by using the memory name as a key. Use the emulator 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 D-51 `dsp_rmem()`

```
/* Read X memory location 100 from device 0. */
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"

unsigned long address;
unsigned long memval;
int devn;
int ok;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */
address=100L;

ok=dsp_rmem(devn,memory_map_x,address,&memval);
```

D.3.52 `dsp_rmem_blk`—Read Block of DSP Memory Locations

```
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"
int dsp_rmem_blk(device_index,memory_map,address,count,return_value)
int device_index;      /* DSP device to be affected by command */
enum memory_map memory_map;/* memory designator */
unsigned long address;/* DSP memory address to read */
unsigned long count;   /* number of memory locations to read */
unsigned long *return_value;/* Returned memory value(s) */
```

`dsp_rmem_blk()` reads `count` locations starting at `address` from memory `memory_map` in device `device_index`, and writes it to the memory block pointed to by `return_value`. If `memory_map` implies a two word value, the values will be returned in order, with the low-order half of the first word followed by the high-order half, then the low-order half of the second word, etc. Thus for word `n` (counting from 0) in the memory block, the low-order value is stored in `return_value[2*n]`, the high-order value in `return_value[2*n+1]`. The function return value indicates whether or not the memory locations exist. It returns `1` if all the locations exist, `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 emulator. The `memory_map` parameter can be obtained with the function `dsp_findmem()` by using the memory name as a key. Use the emulator 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 D-52 `dsp_rmem_blk()`

```
/* Read 20 X memory locations starting at 100 from device 0. */
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"

unsigned long address;
unsigned long count;
unsigned long memval[20];
int devn;
int ok;

devn=0;

dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

address=100L;
count=20L;

ok=dsp_rmem(devn,memory_map_x,address,count,&memval[0]);
```

D.3.53 `dsp_rreg`—Read a DSP Device Register

```
#include "simcom.h"
#include "protocom.h"
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 */
```

`dsp_rreg()` reads a register specified by `periph_num` and `reg_num` from device `device_index` and stores the value in the location pointed to by `reg_val`. Registers which return more than one word as the register value will return the least significant word in `reg_val[0]`, the most significant word in `reg_val[1]`.

Use the emulator "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 D-53 `dsp_rreg()`

```
/* Read register r3 from device 0, a 56002. Use dsp_findreg to obtain the */
/* peripheral and register numbers corresponding to the register name "r3". */
#include "simcom.h"
#include "protocom.h"

int devn;
int periph_num, reg_num;
unsigned long regval;

devn=0;
dsp_new(devn,"56002"); /* Allocate structure for device 0, a 56002 */

if (dsp_findreg(devn,"r3",&periph_num,&reg_num))
    dsp_rreg(devn,periph_num,reg_num,&regval);
```

D.3.54   `dsp_save`—Save All DSP Structures to State File

```
#include "simcom.h"
#include "protocom.h"
int dsp_save(filename)
char *filename;/* Full name of State File to be saved */
```

`dsp_save` saves all DSP device structures to an emulation state file. This function does not invoke the user interface functions which provide pathname and .adm 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 D-54 `dsp_save`

```
/* Save debugger status in file lunchbreak.adm */
#include "simcom.h"
#include "protocom.h"

int ok;

dsp_new(0,"56002");/* Allocate structure for device 0, a 56002 */
dsp_new(1,"56002");/* Allocate structure for device 1, a 56002 */
          /* Save device 0 and 1 to state file lunchbrk.adm. */
ok=dsp_save("lunchbrk.adm");
```

D.3.55   `dsp_spath`—Search Path for Specified File

```
#include "simcom.h"
#include "protocom.h"
int dsp_spath(base, sufx, retn)
char *base;/* Ptr to base file name */
char *sufx;/* Ptr to file extension */
char *retn;/* Ptr to buffer to receive completed file name */
```

`dsp_spath()` takes the base filename `*base` and suffix `*sufx` supplied as arguments, and searches the working directory and alternate source paths until the required file is found.

The working directory is searched first, then each of the alternate source directories in the order in which the paths were specified. The search terminates immediately if a match is found. If multiple files exist on the search path with the same name, the only way to access files after the first file encountered in the search path is to specify the full path explicitly in the input filename `*base`.

If a match is found, the filename return buffer `*retn` contains the fully-specified filename. `*retn` is used as working storage and will be changed whether or not a match is found.

This function returns TRUE if the file is found, FALSE otherwise.

Example D-55   dsp_spath()

```
/* find required file on search path*/
#include "simcom.h"
#include "protocom.h"

int devn, status;
char full_name[80];

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

sim_docmd(devn, "path c:\work\temp");         /* set up working directory */
sim_docmd(device_index, "path + c:\work\bin");/* and alternate source path */

                    /* find first occurrence of 'myfile' on path */
status = dsp_spath("myfile", "lod", full_name);
```

D.3.56 `dsp_startup`—Initialize DSP Structures

```
#include "simcom.h"
#include "protocom.h"
int dsp_startup();
```

`dsp_startup()` initializes general DSP structures which are not device specific. It should be called once (and only once) during program initialization before any calls to `ads_startup()` and `dsp_new()`.

Example D-56 `dsp_startup()`

```
/* startup */
#include "simcom.h"
#include "protocom.h"

ads_startup("100",ADSP56000);
dsp_startup();          /* Initialize DSP structures */

dsp_new(0,"56002");    /* Allocate structure for device 0, a 56002 */
dsp_new(1,"56002");    /* Allocate structure for device 1, a 56002 */
```

D.3.57　`dsp_status`—Determine DSP Device Status

```
#include "simcom.h"
#include "protocom.h"
int dsp_status(device_index, mode)
int device_index;/* DSP device affected by command */
int *mode; /* address of buffer to receive device status */
```

`dsp_status()` places the execution status of the target device `device_index` in the int pointed to by `*mode`.

Valid values for `*mode` are:

- `DSP_USER_MODE`—device is executing a user program

- `DSP_DEBUG_MODE`—device is in debug mode

The function returns `TRUE` on success, or `FALSE` otherwise.

Example D-57 `dsp_status()`

```
/* determine status of DSP 0 */
#include "simcom.h"
#include "protocom.h"

int devn, status;
int device_mode;        /* receive mode of device 0 */

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */
.
.
.
status = dsp_status(devn, &device_mode); /* get current mode for device 0 */
```

D.3.58 `dsp_step`—Execute Counted Instructions

```
#include "simcom.h"
#include "protocom.h"
dsp_step(device_index, step)
int device_index;/*
unsigned long step;
```

`dsp_step()` causes the target device specified by `device_index` to execute `step` instructions. The device then returns to Debug mode.

Note: A success return code means the Command Converter has been instructed to make the target device execute the required number of instructions. It is necessary to call `dsp_check_service_request()` to find out when the target device has executed the instruction and returned to Debug mode.

The function returns TRUE on successful completion, FALSE otherwise,

Example D-58 `dsp_step()`

```
/* Execute 1 DSP instruction on device 0 */
#include "simcom.h"
#include "protocom.h"

int devn, status;
unsigned long step_count;

devn = 0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

step_count = 1l;

status = dsp_step(devn, step_count);
```

D.3.59   `dsp_stop`—Force DSP Device into Debug Mode

```
#include "simcom.h"
#include "protocom.h"
dsp_stop(device_index)
int device_index;/* DSP device affected by command */
```

`dsp_stop()` forces the device `device_index` into Debug mode.

The function returns TRUE on successful completion, FALSE otherwise.

Example D-59 `dsp_stop()`

```
/* force DSP device to debug mode */
#include "simcom.h"
#include "protocom.h"

int devn, status;

devn = 0;
dsp_new(devn,"56002"); /* allocate structure for device 0, a 56002 */
.
.
.
status = dsp_ldmem(devn,"x14.lod");/* load program file */
status = dsp_go(devn); /* start the device running user program */
.
...later...
.
status = dsp_stop(devn);/* stop the device now */
.
.
status = ads_cache_registers(devn); /* cache regs on entry to debug mode */
```

D.3.60 `dsp_unlock`—Unlock Password Protected Device Type

```
#include "simcom.h"
#include "protocom.h"
dsp_unlock(device_type, password)
char *password;/* Pointer to string containing password */
char *device_type;/* Name corresponding to DSP device type */
```

`dsp_unlock()` provides the `password` for protected `device_type`s. It must be used prior to the `dsp_new()` function call if the `device_type` is password protected.

Example D-60 `dsp_unlock()`

```
/* Create a device emulation of the password protected 56001 device */
#include "simcom.h"
#include "protocom.h"

int devn;
ads_startup("100",ADSP56000);
dsp_startup();
devn=0;

dsp_unlock("56001","x51-234"); /* provide password for device */

dsp_new(devn,"56001");/* Create structures for protected device type*/
```

D.3.61 `dsp_wmem`—Write DSP Memory Location

```
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"
dsp_wmem(device_index,memory_map,address,value)
int device_index;      /* DSP device 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 */
```

`dsp_wmem()` writes a selected DSP memory location.

The `memory_map` parameter selects the appropriate `dt_memory` structure from `dt_var.mem` for the selected device. These structures are described in the simdev.h file which is included with the emulator. The `memory_map` parameter can be obtained with the function `dsp_findmem` by using the memory name as a key. Use the emulator help mem command for a list of valid memory names. Valid `memory_map` values are `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.

Example D-61 `dsp_wmem()`

```
/* Write a zero value to address P:200 in device 0 */
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"

int devn;
unsigned long address, memval;
address=200L;
memval=0L;

devn=0;

dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

dsp_wmem(devn,memory_map_p,address,&memval);
```

D.3.62 `dsp_wmem_blk`—Write DSP Memory Block

```
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"
dsp_wmem_blk(device_index,memory_map,address,count,value)
int device_index;      /* DSP device to be affected by command */
enum memory_map memory_map;/* memory designator */
unsigned long address;/* DSP memory address to write */
unsigned long count;   /* Number of locations to write */
unsigned long *value;  /* Pointer to value to write to memory location */
```

`dsp_wmem_blk()` writes `count` locations starting at `address` in memory space `memory_map` in device `device_index`, with values taken from the memory block pointed to by `value`.

If `memory_map` implies a two-word value, the values will be retrieved from `value` in order, with the low-order half of the first word followed by the high-order half, then the low-order half of the second word, etc. Thus for word n (counting from 0) in the memory block, the low-order value is taken from `value[2*n]`, the high-order value from `value[2*n+1]`.

The `memory_map` parameter selects the appropriate `dt_memory` structure from `dt_var.mem` for the selected device. These structures are described in the simdev.h file which is included with the emulator. The `memory_map` parameter can be obtained with the function `dsp_findmem()` by using the memory name as a key. Use the emulator help mem command for a list of valid memory names. Valid `memory_map` values are `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 D-62 `dsp_wmem_blk()`

```
/* Copy 100 values in X:$c400 to P:$a000 */
#include "simcom.h"
#include "protocom.h"
#include "coreaddr.h"

int devn;
unsigned long address, count, memval[100];

memval=0L;
devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */

address=0xc400;
dsp_rmem(devn,memory_map_x,address,100L,&memval[0]); /* fetch 100 values */
address=0xa000;                                   /* from X:$c400 */
dsp_wmem(devn,memory_map_p,address,100L,&memval[0]); /* and write to P:$a000 */
```

D.3.63 `dsp_wreg`—Write a DSP Device Register

```
#include "simcom.h"
#include "protocom.h"
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 */
```

`dsp_wreg()` writes a selected register in the a DSP device.

Use the emulator "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 D-63 `dsp_wreg()`

```
/* Write value 100 to pc register in device 0. Use dsp_findreg to determine
   device and register numbers for pc. */
#include "simcom.h"
#include "protocom.h"

int devn;
int periph_num, reg_num;
unsigned long regval;

devn=0;
dsp_new(devn,"56002");/* Allocate structure for device 0, a 56002 */
regval=100L;

if (dsp_findreg(devn,"pc",&periph_num,&reg_num))
    dsp_wreg(devn,periph_num,reg_num,&regval);
```

D.3.64   `sim_docmd`—Execute Emulator User Interface Command

```
#include "simcom.h"
#include "protocom.h"
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 */
```

`sim_docmd()` executes any emulator command that the emulator normally accepts from the terminal. ADSDSP 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 emulation) 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()`. ADSDSP 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.

Some commands initiate device execution (such as go or trace). The target executes until execution of a breakpoint or the completion of the requested number of instruction steps.

Example D-64   `sim_docmd()`

```
/* Use sim_docmd to execute device 0 from address P:40 to breakpoint at P:80 */
#include "simcom.h"
#include "protocom.h"

int devn;

devn=0;
dsp_new(devn,"56002"); /* Allocate structure for device 0, a 56002 */

sim_docmd(devn,"change pc $40");/* Change device 0 pc register to $40 */
sim_docmd(devn,"break h P:$80");/* Set a breakpoint for device 0 */
sim_docmd(devn,"go");  /* Begin execution of device 0 */
```

D.3.65 `sim_gmcmd`—Get Command String from Macro File

```
#include "simcom.h"
#include "protocom.h"
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 */
```

`sim_gmcmd()` reads the next emulator 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 D-65 `sim_gmcmd()`

```
/* Execute the macro command file startup.cmd on DSP device structure 0. */
#include "simcom.h"
#include "protocom.h"
#include "simusr.h"
extern struct sim_const sv_const; /* Emulator device structures */
char command_string[80];
int devn;

devn=0;
dsp_new(devn,"56002");            /* Create new DSP structure */

sim_docmd(devn,"startup");        /* Begin the startup macro */
while (sv_const.in_macro){        /* Until end of macro file */
   sim_gmcmd(devn,command_string);/*  Get command from macro file */
   sim_docmd(devn,command_string);/*  Execute command string */
   }
```

D.3.66 `sim_gtcmd`—Get Command String from Terminal

```
#include "simcom.h"
#include "protocom.h"
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 */
```

sim_gtcmd() 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 D-66 sim_gtcmd()

```
/* Get and execute emulator commands for device until a go type command is */
/* entered. */
#include "simcom.h"
#include "protocom.h"
#include "simusr.h"

extern struct sim_const sv_const; /* Emulator device structures */
char command_string[80];
int devn;

devn=0;
dsp_new(devn,"56002");                  /* Create new DSP structure */

while (!sv_const.sv[devn]->stat.executing){  /* Check for go. If not, */
   sim_gtcmd(devn,command_string);          /*  Get command and */
   sim_docmd(devn,command_string);          /*  Execute command */
 }
```

D.4   EMULATOR SCREEN MANAGEMENT FUNCTIONS

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

The following is a quick reference list of the emulator 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 emulator 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

D.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.

D.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 emulation.

D.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.

D.4.4 `simw_endwin`—End Emulator Window

`simw_endwin()`

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

D.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 emulator—only when returning from the execution of the system command prior to the time when the emulator's special terminal I/O processing is reinitialized.

D.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 emulator. 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 emulator.

D.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 emulator, and it is not very time critical when it is used, so the emulator implementation is just to call `simw_puts()` after creating a temporary string from the character `c`.

D.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 the string in `text` 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`.

D.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 reentering the emulator following a system command, after loading the device

state with the load s filename command, or after switching devices in a multiple device emulation with the device command.

D.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.

D.4.11 `simw_refresh`—Screen Update after Buffering Output

```
simw_refresh()
```

The emulator 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.

D.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 emulator. The sim_const.scrnest variable is incremented each time this function is called.

D.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.

D.4.14 simw_winit—Initialize Window Parameters

`simw_winit()`

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

D.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 `text` 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 emulator log s or log c commands, depending on `flag`.

D.5   NON-DISPLAY EMULATOR

The emulator package contains object libraries which support both display and non-display versions of the emulator. The library nwads contains functions available to the non-display version of the emulator. The library wwads contains functions that may only be used in a display version of the emulator. 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.

The source code contained in anwdsp.c can be linked with the nwxxxxx and nwsim libraries to create a non-display version of the emulator. Elimination of the user interface functions cuts the code size of the emulator almost in half. However, all of the functions listed in **Section D.4** and `sim_docmd()`, `sim_gmcmd()` and `sim_gtcmd()` described in Sections **Section D.3.64**, **Section D.3.65**, **Section D.3.66**, are sacrificed. The remainder of the functions in **Section D.3** are available in the non-display emulator libraries.

Some major features of the emulator 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 memory. 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. The use of these basic functions to support breakpoints is discussed in **Section D.5.4**. 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 emulator for visual examination of the registers and memory contents.

The following sections cover several topics that concern the non-display version of the emulator. **Section D.5.1** deals with creating a new device.   **Section D.5.2** describes how to load a program or state file. **Section D.5.3** describes how to execute device cycles. **Section D.5.4** describes how to test breakpoint conditions.

D.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 D.3.47**.

The following C source code illustrates the steps necessary to create 3 DSP devices. Note that the numbers used for device number (0, 1, 2 below) reflect the device numbers set up on the target hardware, and do not need to be consecutive.

Example D-67 Device Structures Creation

```
ads_startup("100",ADSP56000);
dsp_startup();
dsp_new(0,"56002");    /* Allocate structure for device 0, a 56002 */
dsp_new(1,"56002");    /* Allocate structure for device 1, a 56002 */
dsp_new(2,"56002");    /* Allocate structure for device 2, a 56002 */
```

D.5.2 Loading Program Code or Device State

The display version of the emulator provides the high level sim_docmd() function interface. It allows the user to simply execute the high level load or load s emulator commands to load program code or a emulator state file. The non-display version of the emulator makes use of the lower level function calls, dsp_ldmem() and dsp_load(), to accomplish the same results. They are described in **Section D.3**. The major difference from their high-level counterparts is that no filename 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 Emulator save command. The OMF file format is described in **Section A.1**. The COFF format files are the output of the Macro Assembler beginning with release 4.0, or those saved by the Emulator save command with the suffix ".cld". The COFF file format is described in **Section B.1**. The Emulator state loaded by the dsp_load() function may have previously been saved by a display or non-display version of the emulator. The formats are the same. The dsp_save() function is provided as a low-level entry point that saves the Emulator state for a non-display version of the Emulator. 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 emulator calls. The dsp_save() function is described in **Section D.3.54**.

D.5.3   Executing Device Instructions

After creating a new device—as described in **Section D.5.1**—and loading a program or state file—as described in **Section D.5.2**—the emulator is ready to execute the program code. Execution begins at the start address specified in the load file, or continues from the previous location in an emulator state file. The user's code may select a new execution address by writing register "pc" using the `dsp_wreg()` function, or with `dsp_go_address()`.

D.5.4   Testing Breakpoint Conditions

The command line interface in the display version of the emulator provides facilities to specify breakpoint conditions. When the breakpoint condition is met during user program execution, the emulator displays the enabled registers (assuming the breakpoint action is halt). The non-display emulator 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 execution. The device registers and memory can be examined using the `dsp_rreg()` and `dsp_rmem()` functions. The emulator hardware executes independently of the emulator control software. After successfully completing a call to initiate instruction execution (e.g. `dsp_go_address()`, `dsp_go,` `dsp_step()`), the device executes until a termination condition is encountered. Termination conditions include:

- Hardware breakpoint condition satisfied

- DEBUG instruction executed

- Specified number of instructions executed in dsp_step

- Call to dsp_reset, etc.

When the device stops executing, the function `dsp_check_service_request()` returns `TRUE`. It is then the responsibility of the user program to determine the reason for the service request. Hardware breakpoints are described in the appropriate hardware documentation and are not covered here.

Software breakpoints are caused by executing conditional or unconditional DEBUG instructions. These instructions must be inserted into the user DSP code. This may be achieved in several possible ways (not all of which are suitable for production code):

- Hard-coded DEBUG instructions included in the DSP code.

- NOP instructions in code which may be overwritten with DEBUG instructions.

- Save instruction word and overwrite with an unconditional DEBUG instruction. MUST be the first word of multi-word instructions.

It is the responsibility of the user code to retain the addresses of the DEBUG instructions and carry out the desired checks and actions when they are encountered.

In the first two possibilities, no special action is needed to continue program execution. A call to dsp_go() or dsp_step() will resume device execution at the location following the DEBUG instruction. Although inappropriate for production code, this approach is simple.

With the third approach, it is necessary to restore the original instruction before continuing, and then reinstate the breakpoint as necessary. The full operation of setting and handling the breakpoint is outlined below:

1. save contents of breakpoint location(s)

2. overwrite with DEBUG instruction(s)

3. start execution with call to dsp_go()

4. repeatedly call dsp_check_service_request() until return value is TRUE

5. save registers with ads_cache_registers()

6. read pc address

7. check for breakpoint address and carry out required checks and actions

8. write the saved instruction word back to memory

9. reset the pc register to the breakpoint address

10. execute 1 instruction with dsp_step

11. repeatedly call dsp_check_service_request() until return value is TRUE

12. continue from (2) above.....

Note: This is only a general outline and may need to be modified for specific circumstances:

Consideration always needs to be given to the case where the first instruction to be executed is itself the target of a breakpoint. In this case, step over that instruction before inserting the DEBUG instructions. Adding or deleting of breakpoints needs to be handled. Steps (4) and (11) above do not need to be tight loops. Calls to dsp_check_service_request() may be made at convenient points in other processing to determine when the device is ready.

D.6   MULTIPLE DEVICE EMULATION

The ADSDSP emulator may be used to emulate a single DSP device or multiple devices. As many devices as are required by the target configuration may be configured using the device command or dsp_new function(). The following sections describe some details about the way the emulator handles multiple devices. **Section D.6.1** describes the required steps to allocate and initialize multiple DSP structures. **Section D.6.2** describes the method of controlling multiple devices. **Section D.6.3** describes display of device output in the multiple device environment.

D.6.1   Allocation and Initialization of Multiple Devices

Most of the higher level emulator functions require a device index as one of the parameters. The emulator 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 emulator, the device command handles the details of calling dsp_new(). The proper sequence of instructions necessary to allocate three DSP devices is shown below.

```
ads_startup("100",ADSP56000);
dsp_startup();
dsp_new(0,"56002");    /* Allocate structure for device 0, a 56002 */
dsp_new(1,"56002");    /* Allocate structure for device 1, a 56002 */
dsp_new(2,"56002");    /* Allocate structure for device 2, a 56002 */
```

D.6.2   Controlling Multiple DSP Devices

Each target device operates independently under the control of the ADSDSP emulator. The basic execution sequence for a single target device is unchanged:

1.   Initialize DSP device

2. Load memory

3. initialize breakpoints as desired

4. initiate execution (`dsp_go()`, `dsp_step()`, etc.)

5. IF DESIRED, interrupt execution with `dsp_reset()` or `dsp_stop()`

6. call `dsp_check_service_request()` to detect return to Debug Mode

When controlling multiple devices, all devices require initialization as above. Whenever a device starts executing, as a result of a command like step or a function call like `dsp_go()`, execution starts immediately for the specified device and continues independently of the ADS. Each device must be started by a separate command or function call. Execution continues until an event in the device causes the device to enter Debug Mode, such as reaching a breakpoint or the required number of instructions being executed, or the ADS stops execution for a target, with a force b command or a call to function `dsp_stop()`. All other devices will continue executing while that device is being serviced. The emulator needs to check each device to see if it has entered Debug Mode by calling `dsp_check_service_request()` for each executing device in turn. This may be coded as a tight loop for maximum response, or interleaved with other activity as required.

D.6.3   Multiple DSP Emulator Display

The emulator display functions are contained in the source file scrmgr.c in the emulator package. This code supports the scrolling virtual screen for the ADS. The supplied display code uses a single window. The lines above the command line form a scrolling region in which session output is displayed. The command line, error line and help line are the three bottom lines of the display. The default size of the scroll buffer holds 100 lines of output. As each device causes output to the screen, a message is output specifying which device caused the output. 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.

D.7   RESERVED FUNCTION NAMES

The public function names used in the emulator all begin with the prefixes dsp_, ads_, or sim_. Functions which begin with sim_ are only available when a display version of the emulator is created. Functions which begin with dsp_ and ads_ 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 dspd_, dspl_ or siml_ are meant only for internal use by the emulator.   The higher level functions and the screen management functions are documented in **Section D.3** and **Section D.4**. The public functions are listed in the file named global.sym which is included with the distribution.

D.8   EMULATOR GLOBAL VARIABLES

In order to reduce conflicts with user variable names, the emulator 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 emulator; while those beginning with d are defined in simdev.h and are used by both the display and non-display versions of the emulator. 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 emulator global variables is included in the distribution file named global.sym.

D.9 MODIFICATION OF EMULATOR GLOBAL STRUCTURES

The source file simglob.c, which is included in the emulator 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 emulator package.

- **DSP_MAXDEVICES**—This define constant determines the maximum number of devices that can be allocated using the emulator'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 emulator 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_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 emulator 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.

MOTOROLA Wireless Signal Processing

Suggestion/Problem Report

Motorola DSP Department welcomes your comments on its products and publications. Please use this form to report any problems, enhancement requests or ideas you have concerning this product and send it to the following address:

Motorola
Attention: Wireless Signal Processors
Mail Stop OE314
6501 William Cannon Drive West
Austin, Texas 78735-8598

For fastest response, Motorola DSP Department may also be may also be contacted electronically. Send e-mail to:

dsphelp@dsp.sps.mot.com
doc-update@dsp.sps.mot.com

Or visit our web site. You may browse our extensive database, including all our latest documentation and product news. Our web address is:

http://www.motorola-dsp.com

Please tell us about yourself.

Name . . . . Date . . . .

Company. . . . .

E-mail address. . . . . Phone . . . .

Street Address . . . . City. . . . .

State . . . . Zip . . . . Country . . . .

Please give a complete description of the configuration you are using, with any comments or problems. This should include the development environment in use, the type of host computer, software and any external circuitry you are using with this product and, if possible, how to reproduce any problem reported.

ALL INFORMATION WILL BE TREATED AS STRICTLY CONFIDENTIAL.

DSP device type: . . . . Host Computer type (PC, SUN, HP): . . . .

Host Operating System . . . . Version: . . . .

Assembler / Compiler Revision: . . . . Linker Revision: . . . .

ADS Interface / Simulator Revision: . . . . . . If UNIX host, Device Driver Revision: . . . .

Command Convertor Monitor Revision: . . . . . Other S/W: . . . .

Problem description: . . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

. . . .

(Please continue on reverse if necessary)

Motorola WSP Technical Helpline   **(512) 891-3230**

MOTOROLA Wireless Signal Processing

Report Continuation

. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .

(Please continue on extra sheets if necessary)

INDEX

D

D

E

N

N

O

P

R

S

T

U

W