TMS320C2x Evaluation Module

Addendum to the TMS320C2x C Source Debugger User's Guide

Literature Number: SPRU085
Reprinted October 1997

Contents

Tables

Chapter 1

Installing the Evaluation Module and the C Source Debugger

This chapter helps you install the ’C2x evaluation module (EVM) and the C source debugger on a PC running MS-DOS or PC-DOS. You can also use the debugger with MS-Windows.

Topic **Page**

1.1 What You'll Need

The following checklists detail items that are shipped with the 'C2x C source debugger and EVM and additional items you'll need to use these tools.

Hardware checklist

☐	**host**	An IBM PC/AT or 100% compatible ISA/EISA-based PC with a hard-disk system and a 1.2M floppy-disk drive
☐	**memory**	Minimum of 640K; in addition, if you are running under Microsoft Windows, you'll need at least 256K of extended memory.
☐	**display**	Monochrome or color (color recommended)
☐	**slot**	One 16-bit slot
☐	**EVM board power requirements**	Approximately 1.5 amperes @ 5 volts (15 watts)
☐	**optional hardware**	A Microsoft-compatible mouse
☐		An EGA- or VGA-compatible graphics display card and a large monitor. The debugger has several options that allow you to change the overall size of the debugger display. If you have an EGA- or VGA-compatible graphics card, you can take advantage of some of these larger screen sizes. These larger screen sizes are most effective when used with a large (17" or 19") monitor. (To use a larger screen size, you must invoke the debugger with an appropriate option. For more information about options, refer to Section 1.5.)
☐	**miscellaneous materials**	Blank, formatted disks

Software checklist

☐	**operating system**		MS-DOS or PC-DOS (version 3.0 or later) Optional: Microsoft Windows (version 3.0 or later)
☐	**software tools**		TMS320 fixed-point family DSP ('C1x/'C2x/'C5x) assembler and linker Optional: TMS320C2x/C5x C compiler
☐	**required files**	†	*evmrst.exe* resets the EVM
☐		†	*c2xevm.out* runs on the 'C2x, allowing the target processor and the debugger to communicate.
☐	**optional files**	†	*evminit.cmd* is a file that contains debugger commands. The version of this file that's shipped with the debugger defines a 'C2x memory map. If this file isn't present when you invoke the debugger, then all memory is invalid at first. When you first start using the EVM, this memory map should be sufficient for your needs. Later, you may want to define your own memory map. For information about setting up your own memory map, refer to the *Defining a Memory Map* chapter in the *TMS320C2x C Source Debugger User's Guide.*
☐		†	*init.clr* is a general-purpose screen configuration file. If this file isn't present when you invoke the debugger, the debugger uses the default screen configuration.
☐		†	The default configuration is for color monitors; an additional file, *mono.clr*, can be used for monochrome monitors. When you first start to use the debugger, the default screen configuration should be sufficient for your needs. Later, you may want to define your own custom configuration. For information about these files and about setting up your own screen configuration, refer to the *Customizing the Debugger Display* chapter in the *TMS320C2x C Source Debugger User's Guide.*

† Included as part of the debugger package

1.2 Step 1: Installing the EVM Board in Your PC

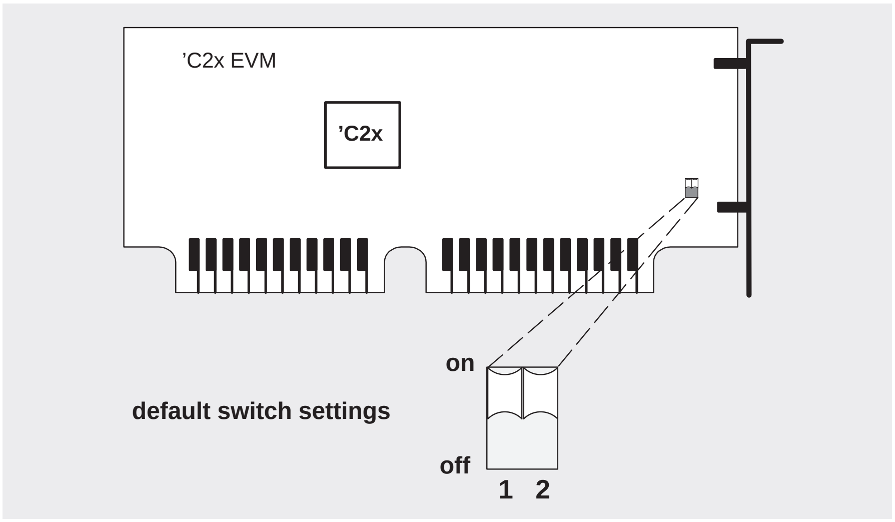
This section contains the hardware installation information for the EVM.

Preparing the EVM board for installation

Before you install the EVM board, you must be sure that the board's switches are set to correctly identify the I/O space that the board can use. The EVM board has two switches that identify your system's I/O address space. You can change these switch settings to identify the I/O address space that the EVM uses in your system.

Figure 1–1 shows where these switches are on the EVM board and identifies the switch numbers.

Figure 1–1. EVM Board I/O Switches



Switches are shipped in the default settings shown here and described in Table 1–1. If you use an I/O space that differs from the default, change the switch settings. Table 1–1 shows you how to do this.

In most cases, you can leave the switch settings in the default position. However, you must ensure that the EVM I/O address space does not conflict with other bus settings. For example, if you've installed a bus mouse in your system, you may not be able to use the default switch settings for the I/O address space—the mouse might use this space. Refer to your PC technical reference manual and your other hardware-board manuals to see if there are any I/O space conflicts. If you find a conflict, use one of the settings in Table 1–1.

Table 1–1. EVM Board Switch Settings

		Switch #	
	Address Range	1	2
default	0x0240–0x025F	on	on
	0x0280–0x029F	on	off
	0x0320–0x033F	off	on
	0x0340–0x035F	off	off

Some of the other installation steps require you to know which switch settings you used. If you reset the I/O switches, note the modified settings here for later reference.

Table 1–2. Your Switch Settings

	Switch #	
Address Range	1	2

Setting the EVM board into your PC

After you've prepared the EVM board for installation, follow these steps.

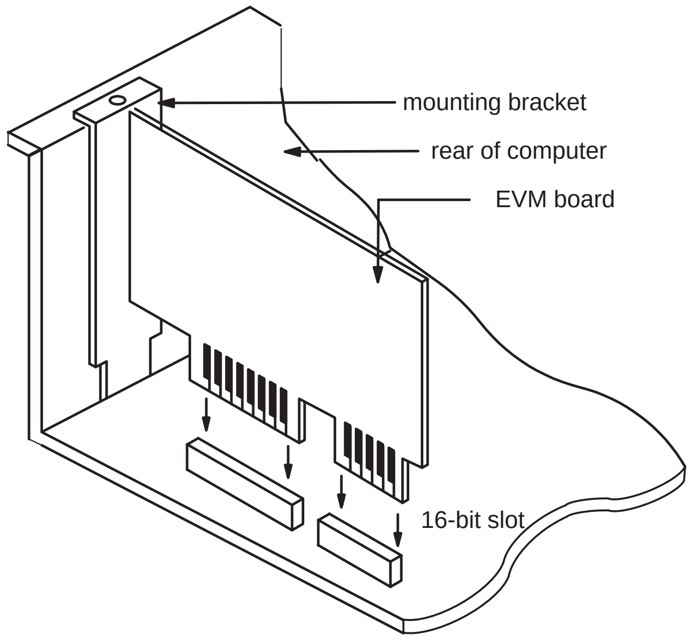
Step 1: Turn off your PC's power and unplug the power cord.

Step 2: Remove the cover of your PC.

Step 3: Remove the mounting bracket from an unused 16-bit slot.

Step 4: Install the EVM board in a 16-bit slot (see Figure 1–2).

Figure 1–2. EVM Board Installation



Step 5: Tighten down the mounting bracket.

Step 6: Replace the PC cover.

Step 7: Plug in the power cord and turn on the PC's power.

1.3 Step 2: Installing the Debugger Software

This section explains the process of installing the debugger software on a hard-disk system.

1) Make a backup copy of each disk. (If necessary, refer to the DOS manual that came with your computer.)

2) On your hard disk or system disk, create a directory named *c2xhll.* This directory will contain the debugger software.

   ```
   MD C:\C2XHLL ↵
   ```

3) Insert a product disk into drive A. Copy the debugger software onto the hard disk or system disk.

   ```
   COPY A:\*.* C:\C2XHLL\*.* /V ↵
   ```

   Repeat this step for each product disk.

4) You must set up to use the correct executable file according to whether or not you plan to use Microsoft Windows. If you plan to use Microsoft Windows, delete evm2x.exe from your hard disk and change the name of evm2xw.exe to evm2x.exe:

   ```
   DEL EVM2X.EXE ↵
   REN EVM2XW.EXE EVM2X.EXE ↵
   ```

   If you do not plan to use Microsoft Windows, delete evm2xw.exe from your hard disk:

   ```
   DEL EVM2XW.EXE ↵
   ```

1.4 Step 3: Setting Up the Debugger Environment

To ensure that your debugger works correctly, you must:

- ❑ Modify the PATH statement to identify the c2xhll directory.
- ❑ Define environment variables so that the debugger can find the files it needs.
- ❑ Identify any nondefault I/O space used by the EVM.

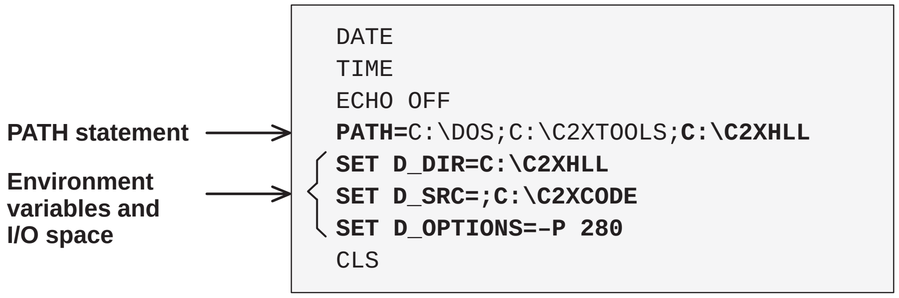
Not only must you do these things before you invoke the debugger for the first time, *you must do them any time you power up or reboot your PC.*

You can accomplish these tasks by entering individual DOS commands, but it's simpler to put the commands in a batch file. You can edit your system's autoexec.bat file; in some cases, modifying the autoexec may interfere with other applications running on your PC. So, if you prefer, you can create a separate batch file that performs these tasks.

Figure 1–3 *(a)* shows an example of an autoexec.bat file that contains the suggested modifications (highlighted in bold type). Figure 1–3 *(b)* shows a sample batch file that you could create instead of editing the autoexec.bat file. (For the purpose of discussion, assume that this sample file is named *initdb.bat*.) The subsections following the figure explain these modifications.

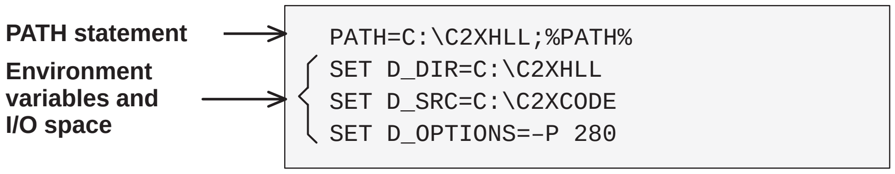
Figure 1–3. DOS-Command Setup for the Debugger

(a) Sample autoexec.bat file to use with the debugger and EVM



```
DATE
TIME
ECHO OFF
PATH=C:\DOS;C:\C2XTOOLS;C:\C2XHLL
SET D_DIR=C:\C2XHLL
SET D_SRC=;C:\C2XCODE
SET D_OPTIONS=–P 280
CLS
```

(b) Sample initdb.bat file to use with the debugger and EVM



```
PATH=C:\C2XHLL;%PATH%
SET D_DIR=C:\C2XHLL
SET D_SRC=C:\C2XCODE
SET D_OPTIONS=–P 280
```

Invoking the new or modified batch file

- ❑ If you modify the autoexec.bat file, be sure to invoke it before invoking the debugger for the first time. To invoke this file, enter:

  ```
  AUTOEXEC
  ```

- ❑ If you create an initdb.bat file, you must invoke it before invoking the debugger for the first time. If you are using Microsoft Windows, invoke initdb.bat *before* entering Microsoft Windows. You'll need to invoke initdb.bat any time that you power-up or reboot your PC. To invoke this file, enter:

  ```
  INITDB
  ```

Modifying the PATH statement

Define a path to the debugger directory. The general format for doing this is:

```
PATH=C:\C2XHLL
```

This allows you to invoke the debugger without specifying the name of the directory that contains the debugger executable file.

- ❑ If you are modifying an autoexec that already contains a PATH statement, simply include ;C:\c2xhll at the end of the statement as shown in Figure 1–3 *(a)*.

- ❑ If you are creating an initdb.bat file, use a different format for the PATH statement:

  ```
  PATH=C:\C2XHLL;%PATH%
  ```

  The addition of ;%path% ensures that this PATH statement won't undo PATH statements in any other batch files (including the autoexec.bat file).

Setting up the environment variables

An environment variable is a special system symbol that the debugger uses for finding or obtaining certain types of information. The debugger uses three environment variables, named D_DIR, D_SRC, and D_OPTIONS. The rest of the subsection tells you how to set up these environment variables. The format for doing this is the same for both the autoexec.bat and initdb.bat files.

- ❑ Set up the D_DIR environment variable to identify the c2xhll directory:

  ```
  SET D_DIR=C:\C2XHLL
  ```

  (Be careful not to precede the equal sign with a space.)

  This directory contains auxiliary files (evmrst, evminit.cmd, etc.) that the debugger needs.

- ❑ Set up the D_SRC environment variable to identify any directories that contain program source files that you'll want to look at while you're debugging code. The general format for doing this is:

  SET D_SRC=*pathname*$_1$*;pathname*$_2$*...*

  For example, if your 'C2x programs were in a directory named *c2xsrc* on drive C, the D_SRC setup would be:

  ```
  SET D_SRC=C:\C2XSRC
  ```

- ❑ You can use several options when you invoke the debugger. If you use the same options over and over, it's convenient to specify them with D_OPTIONS. The general format for doing this is:

  SET D_OPTIONS= [*object filename*] [*debugger options*]

  This tells the debugger to load the specified object file and use the specified options each time you invoke the debugger. These are the options that you can identify with D_OPTIONS:

–b	–bb	–c	–i *pathname*
–p *port address*	–s	–t *filename*	–v

  For more information about options, see Section 1.5. Note that you can override D_OPTIONS by invoking the debugger with the –x option.

Identifying the correct I/O switches

Refer to your entries in Table 1–2 (page 1-5). If you didn't modify the I/O switches, skip this step.

If you modified the I/O switch settings, you must use the debugger's –p option to identify the I/O space that the EVM is using. You can do this each time you invoke the debugger, or you can specify this information by using the D_OPTIONS environment variable. Table 1–3 lists the nondefault I/O switch setting and the appropriate line that you can add to the autoexec.bat or initdb.bat file.

Table 1–3. Identifying Nondefault I/O Address Space

	switch #		Add this line to the
Address Range	**1**	**2**	**batch file**
0x0280–0x029F	on	off	`SET D_OPTIONS=–p 280`
0x0320–0x033F	off	on	`SET D_OPTIONS=–p 320`
0x0340–0x035F	off	off	`SET D_OPTIONS=–p 340`

Note: I/O Address Space

1) The ’C2x EVM uses 96 bytes of the PC I/O space.

2) If you didn’t note the I/O switch settings, you may use a trial-and-error approach to find the correct –p setting. **If you use the wrong setting, you’ll see this error message when you try to invoke the debugger:**

```
CANNOT INITIALIZE THE EVM ! !
– Check I/O configuration
```

If you plan to use the EVM for running other host applications (for example, a modem), you must first load a valid object file into the EVM. To do this, invoke the debugger and load the object file:

evm2x *filename*

Once you have entered the debugging environment and the object file has been loaded, exit the debugger:

QUIT ↵

At the DOS prompt, reset the EVM by entering the evmrst command:

EVMRST ↵

If you decide to change your I/O switch settings, you can specify a different I/O space by entering the –p option following evmrst. The evmrst command also reads the D_OPTIONS environment variable in your autoexec.bat or initdb.bat file. You can override D_OPTIONS by entering evmrst followed by the –x option.

Notes:

- ❑ Never reset the ’C2x EVM with evmrst unless you have first loaded a valid object file to the EVM.
- ❑ If you plan to use the debugger with the EVM, you don’t need to reset the EVM with evmrst before invoking the debugger.

1.5 Invoking the Debugger

Here's the basic format for the command that invokes the debugger:

```
evm2x    [filename]  [–options]
```

evm2x is the command that invokes the debugger.

filename is an optional parameter that names an object file that the debugger will load into memory during invocation. The debugger looks for the file in the current directory; if the file isn't in the current directory, you must supply the entire pathname. If you don't supply an extension for the filename, the debugger assumes that the extension is .out.

–options supply the debugger with additional information (Table 1–4 summarizes the available options).

In a DOS environment, you can also specify filename and option information with the D_OPTIONS environment variable (see *Setting up the environment variables*, page 1-9). Table 1–4 lists the debugger options and specifies which debugger tools use the options; the subsections following the table describe the options.

Table 1–4. Summary of Debugger Options

Option	Brief description
–b[b]	Select the screen size
–c	Clear memory
–i *pathname*	Identify additional directories
–p *port address*	Identify the port address
–s	Load the symbol table only
–t *filename*	Identify a new initialization file
–v	Load without the symbol table
–x	Ignore D_OPTIONS

Selecting the screen size (–b option)

By default, the debugger uses an 80-character-by-25-line screen. You can use one of the options below to specify a different screen size.

Option	Description	Display
none	80 characters by 25 lines	Default display
–b	80 characters by 43 lines	Any EGA or VGA display
–bb	80 characters by 50 lines	VGA only

Clearing memory (–c option)

The –c option sets the memory reserved for uninitialized data to all zeros.

Identifying additional directories (–i option)

The –i option identifies additional directories that contain your source files. Replace *pathname* with an appropriate directory name. You can specify several pathnames; use the –i option as many times as necessary. For example:

evm2x **–i** *pathname*$_1$ **–i** *pathname*$_2$ **–i** *pathname*$_3$ . . .

Using –i is similar to using the D_SRC environment variable (see *Setting up the environment variables*, page 1-9). If you name directories with both –i and D_SRC, the debugger first searches through directories named with –i. The debugger can track a cumulative total of 20 paths (including paths specified with –i, D_SRC, and the debugger USE command).

Identifying the port address (–p option)

The –p option identifies the I/O port address that the debugger uses for communicating with the EVM. If you used the default switch settings, you don't need to use the –p option. **If you used nondefault switch settings, you must use –p**. Refer to your entries in the *Your Settings* table, page 1-5; depending on your switch settings, replace *port address* with one of these values:

Switch 1	Switch 2	Option
on	on	–p 240 (optional)
on	off	–p 280
off	on	–p 320
off	off	–p 340

If you didn't note the I/O switch settings, you can use a trial-and-error approach to find the correct –p setting. If you use the wrong setting, you will see an error message when you invoke the debugger.

Loading the symbol table only (–s option)

If you supply a *filename* when you invoke the debugger, you can use the –s option to tell the debugger to load only the file's symbol table (without the file's object code). This is similar to the debugger's SLOAD command.

Identifying a new initialization file (–t option)

The –t option allows you to specify an initialization command file that will be used instead of evminit.cmd. If –t is present on the command line, the file specified by *filename* will be invoked as the command file instead of evminit.cmd.

Loading without the symbol table (–v option)

The –v option prevents the debugger from loading the entire symbol table when you load an object file. The debugger loads only the global symbols and later loads local symbols as it needs them. This speeds up the loading time and consumes less memory space.

The –v option affects all loads, including those performed when you invoke the debugger and those performed with the LOAD command within the debugger environment.

Ignoring D_OPTIONS (–x option)

The –x option tells the debugger to ignore any information supplied with D_OPTIONS. For more information about D_OPTIONS, refer to page 1-10.

1.6 Exiting the Debugger

To exit the debugger and return to the operating system, enter this command:

```
quit ↵
```

You don't need to worry about where the cursor is or which window is active—just type. If a program is running, press ESC to halt program execution before you quit the debugger.

If you are running the debugger under Microsoft Windows, you can also exit the debugger by selecting the exit option from the Microsoft Windows menu bar.

1.7 Installation Error Messages

While invoking the debugger, you may see the following message:

```
CANNOT INITIALIZE THE EVM ! !
– Check I/O configuration
```

To determine the problem, follow these actions:

- ❑ Check the EVM board to be sure it is installed snugly.
- ❑ Ensure your port address is set correctly:
  - ■ Check to be sure the –p option used with the D_OPTIONS environment variable matches the I/O address defined by your switch settings (refer to *Your Switch Settings*, Table 1–2, and *Identifying Nondefault I/O Address Space*, Table 1–3).
  - ■ Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings in Table 1–1. Modify the –p option of the D_OPTIONS environment variable to reflect the change in your switch settings.

1.8 Using the Debugger With Microsoft Windows

If you're using Microsoft Windows, you can freely move or resize the debugger display on the screen. If the resized display is bigger than the debugger requires, the extra space is not used. If the resized display is smaller than required, the display is clipped. Note that when the display is clipped, it can't be scrolled.

You may want to create an icon to make it easier to invoke the debugger from within the Microsoft Windows environment. Refer to your Microsoft Windows manual for details.

You should run Microsoft Windows in either the standard mode or the 386 enhanced mode to get the best results.

Chapter 2

New Features of the TMS320C2x C Source Debugger

This chapter describes new debugger features that are not documented in the *TMS320C2x C Source Debugger User's Guide.*

Topic **Page**

2.1 Zooming the Active Window

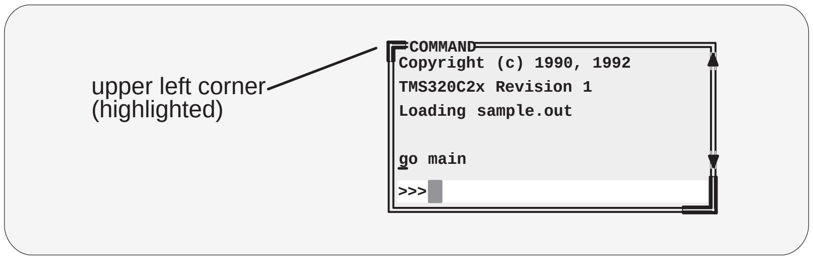
The easiest way to resize the active window is to zoom it. Zooming a window makes it as large as possible so that it takes up the entire display (except for the menu bar) and hides all the other windows. Unlike the SIZE command, zooming is not affected by the window's position in the display.

To "unzoom" a window, repeat the same steps you used to zoom it. This will return the window to its prezoom size and position.

There are two basic ways to zoom or unzoom a window:

- ❑ By using the mouse
- ❑ By using the ZOOM command

1) Point to the upper left corner of the window. This corner is highlighted—here's what it looks like:



```
Copyright (c) 1990, 1992
TMS320C2x Revision 1
Loading sample.out

go main
>>>
```

2) Click the left mouse button.

zoom You can also use the ZOOM command to zoom/unzoom the window. The format for this command is:

zoom

For more information about the active window or manipulating windows, see *The Debugger Display* chapter in the *TMS320C2x C Source Debugger User's Guide.*

2.2 Defining Your Own Command Strings

The debugger provides a shorthand method of entering often-used commands or command sequences. This processing is called *aliasing*. Aliasing enables you to define an alias name for the command(s) and then enter the alias name as if it were a debugger command.

To do this, use the ALIAS command. The syntax for this command is:

alias [*alias name* [, “*command string*”]]

The primary purpose of the ALIAS command is to associate the *alias name* with the debugger command you’ve supplied as the *command string*. However, the ALIAS command is versatile and can be used in several ways:

- **Aliasing several commands.** The *command string* can contain more than one debugger command—just separate the commands with semicolons.

  For example, suppose you always began a debugging session by loading the same object file, displaying the same C source file, and running to a certain point in the code. You could define an alias to do all these tasks at once:

  ```
  alias init,”load test.out;file source.c;go main”
  ```

  Now you could enter init instead of the three commands listed within the quote marks.

- **Supplying parameters to the command string.** The *command string* can define parameters that you’ll supply later. To do this, use a percent sign and a number (%1) to represent the parameter that will be filled in later. The numbers should be consecutive (%1, %2, %3) unless you plan to reuse the same parameter value for multiple commands.

  For example, suppose that every time you filled an area of memory, you also wanted to display that block in the MEMORY window:

  ```
  alias mfil,”fill %1, %2, %3, %4;mem %1”
  ```

  Then you could enter:

  ```
  mfil 0xff80,1,0x18,0x1122
  ```

  The first value (0xff80) would be substituted for the first FILL parameter and the MEM parameter (%1). The second, third, and fourth values would be substituted for the second, third, and fourth FILL parameters (%2, %3, and %4).

- **Listing all aliases.** To display a list of all the defined aliases, enter the ALIAS command with no parameters. The debugger will list the aliases and their definitions in the COMMAND window.

  For example, assume that the init and mfil aliases had been defined as shown in the previous two examples. If you entered:

  `alias`

  you'd see:

  ```
    Alias       Command
  -----------------------------------------
  INIT    -->   load test.out;file source.c;go main
  MFIL    -->   fill %1,%2,%3,%4;mem %1
  ```

- **Finding the definition of an alias.** If you know an alias name but are not sure of its current definition, enter the ALIAS command with just an alias name. The debugger will display the definition in the COMMAND window.

  For example, if you had defined the init alias as shown in the first example above, you could enter:

  `alias init`

  Then you'd see:

  ```
  ”INIT” aliased as ”load test.out; file source.c;go main”
  ```

- **Nesting alias definitions.** You can include a defined alias name in the *command string* of another alias definition. This is especially useful when the command string would be longer than the debugger command line.

- **Redefining an alias.** To redefine an alias, re-enter the ALIAS command with the same alias name and a new command string.

- **Deleting aliases.** To delete a single alias, use the UNALIAS command:

  unalias *alias name*

  To delete *all* aliases, enter the UNALIAS command with an asterisk instead of an alias name:

  `unalias *`

  Note that the * symbol *does not* work as a wildcard.

Note: Limitations of Alias Definitions

- ❑ Alias definitions are lost when you exit the debugger. If you want to reuse aliases, define them in a batch file.
- ❑ Individual commands within a command string are limited to an expanded length of 132 characters. The expanded length of the command includes the length of any substituted parameter values.

2.3 Entering Operating-System Commands (DOS Only)

The debugger provides a simple method of entering DOS commands without explicitly exiting the debugger environment. To do this, use the SYSTEM command. The format for this command is:

system [*DOS command* [, *flag*]]

The SYSTEM command behaves in one of two ways, depending on whether or not you supply an operating-system command as a parameter:

- ❑ If you enter the SYSTEM command with a DOS command as a parameter, then you stay within the debugger environment.
- ❑ If you enter the SYSTEM command without parameters, the debugger opens a *system shell*. This means that the debugger will blank the debugger display and temporarily exit to the operating-system prompt.

Use the first method when you have only one command to enter; use the second method when you have several commands to enter.

Entering a single command from the debugger command line

If you need to enter only a single DOS command, supply it as a parameter to the SYSTEM command. For example, if you want to copy a file from another directory into the current directory, enter:

```
system copy a:\backup\sample.c sample.c ↵
```

If the DOS command produces a display of some sort (such as a message), the debugger will blank the upper portion of the debugger display to show the information. In this situation, you can use the *flag* parameter to tell the debugger whether or not it should hesitate after displaying the results of the DOS command. *Flag* may be a 0 or a 1:

0	The debugger immediately returns to the debugger environment after the last item of information is displayed.
1	The debugger does not return to the debugger environment until you press ↵. (This is the default.)

In the preceding example, the debugger would open a system shell to display the following message:

```
         1 File(s) copied
Type Carriage Return To Return To Debugger
```

The message would be displayed until you pressed ↵.

If you wanted the debugger to display the message and then return immediately to the debugger environment, you could enter the command in this way:

```
system copy a:\backup\sample.c sample.c,0 ↵
```

Entering several commands from a system shell

If you need to enter several commands, enter the SYSTEM command without parameters. The debugger will open a system shell and display the DOS prompt. At this point, you can enter any DOS command.

When you are finished entering commands and are ready to return to the debugger environment, enter:

```
exit ↵
```

Note: Memory Limitation When Using a System Shell

Available memory may limit the DOS commands that you can enter from a system shell. For example, you would not be able to invoke another version of the debugger.

2.4 Recording Information From the Display Area

The information shown in the display area of the COMMAND window can be written to a log file. The log file is a system file that contains commands you've entered, their results, and error or progress messages. To record this information in a log file, use the DLOG command.

Log files can be executed by using the TAKE command. When you use DLOG to record the information from the COMMAND window display area, the debugger automatically precedes all error or progress messages and command results with a semicolon to turn them into comments. This way, you can easily re-execute the commands in your log file by using the TAKE command.

❑ To begin recording the information shown in the COMMAND window display area, use:

  dlog *filename*

  This command opens a log file called *filename* that the information is recorded into.

- To end the recording session, enter:

  dlog close ↵

If necessary, you can write over existing log files or append additional information to existing files. The extended format for the DLOG command is:

dlog *filename* [**,**{**a** | **w**}]

The optional parameters of the DLOG command control how the log file is created and/or used:

- **Creating a new log file.** If you use the DLOG command without one of the optional parameters, the debugger creates a new file that it records the information into. If you are recording to a log file already, entering a new DLOG command and filename closes the previous log file and opens a new one.
- **Appending to an existing file.** Use the **a** parameter to open an existing file to which to append the information in the display area.
- **Writing over an existing file.** Use the **w** parameter to open an existing file to write over the current contents of the file. Note that this is the default action if you specify an existing filename without using either the **a** or **w** options; you will lose the contents of an existing file if you don't use the append (a) option.

2.5 Additional Batch File Features

Echoing strings in a batch file

When executing a batch file, you can display a string to the COMMAND window by using the ECHO command. The syntax for the command is:

echo *string*

This displays the *string* in the COMMAND window display area.

For example, you may want to document what is happening during the execution of a certain batch file. To do this, you could use the following line in your batch file to indicate that you are creating a new memory map for your device:

```
echo Creating new memory map
```

(Notice that the string should not be in quotes.)

When you execute the batch file, the following message appears:

```
.
.
Creating new memory map
.
.
```

Note that any leading blanks in your string are removed when the ECHO command is executed.

Controlling command execution in a batch file

In batch files, you can control the flow of debugger commands. You can choose to conditionally execute debugger commands or set up a looping situation by using IF/ELSE/ENDIF or LOOP/ENDLOOP, respectively.

- To conditionally execute debugger commands in a batch file, use the IF/ELSE/ENDIF commands. The syntax is:

  if *Boolean expression*
  debugger command
  debugger command
  .
  .
  [**else**
  debugger command
  debugger command
  .
  .]
  endif

  The debugger includes some predefined constants for use with IF. These constants evaluate to 0 (false) or 1 (true). Table 2–1 shows the constants and their corresponding tools.

Table 2–1. Predefined Constants for Use With Conditional Commands

Constant	Debugger Tool
$$SWDS$$	software development system
$$SIM$$	simulator
$$EVM$$	evaluation module

If the Boolean expression evaluates to true (1), the debugger executes all commands between the IF and ELSE or ENDIF. Note that the ELSE portion of the command is optional. (See the *Basic Information About C Expressions* chapter in the *TMS320C2x C Source Debugger User's Guide* for more information about expressions and expression analysis.)

One way you can use these predefined constants is to create an initialization batch file that works for any debugger tool. This is useful if you are using, for example, both the SWDS and the EVM. To do this, you can set up the following batch file:

```
if $$SWDS$$
echo Invoking initialization batch file for SWDS.
use \c2xhll
take dbinit.cmd
.
.
endif

if $$EVM$$
echo Invoking initialization batch file for EVM.
use \c2xhll
take evminit.cmd
.
.
endif
.
.
```

In this example, the debugger will execute only the initialization commands that apply to the debugger tool that you invoke.

❑ To set up a looping situation to execute debugger commands in a batch file, use the LOOP/ENDLOOP commands. The syntax is:

loop *expression*
debugger command
debugger command
.
.
endloop

These looping commands evaluate in the same method as in the run conditional command expression. (See the *Basic Information About C Expressions* chapter in the *TMS320C2x C Source Debugger User's Guide* for more information about expressions and expression analysis.)

- If you use an *expression* that is not Boolean, the debugger evaluates the expression as a loop count. For example, if you wanted to execute a sequence of debugger commands ten times, you would use the following:

```
loop 10
step
.
.
endloop
```

The debugger treats the 10 as a counter and executes the debugger commands ten times.

- If you use a Boolean *expression*, the debugger executes the commands repeatedly as long as the expression is true. This type of expression has one of the following operators as the highest precedence operator in the expression:

>	>=	<
<=	==	!=
&&	||	!

For example, if you want to continuously trace some register values, you can set up a looping expression like the following:

```
loop !0
step
? PC
? AR0
endloop
```

The IF/ELSE/ENDIF and LOOP/ENDLOOP commands work with the following conditions:

- ❑ You can use conditional and looping commands only in a batch file.
- ❑ You must enter each debugger command on a separate line in the batch file.
- ❑ You can't nest conditional and looping commands within the same batch file.

2.6 Patch Assembly

You can modify the code in the disassembly window on a statement-by-statement basis. The method for doing this is called *patch assembly.* Patch assembly provides a simple way to temporarily correct minor problems by allowing you to change individual statements and instruction words.

You can patch-assemble code by using a command or by using the mouse.

patch Use the PATCH command to identify the address of the statement you want to change and the new statement you want to use at that address. The format for this command is:

patch *address, assembly language statement*

For patch assembly, use the **right** mouse button instead of the left. (Clicking the left mouse button sets a software breakpoint.)

1) Point to the statement that you want to modify.

2) Click the right button. The debugger will open a dialog box so that you can enter the new statement. The address field will already be filled in; clicking on the statement defines the address. The statement field will already be filled in with the current statement at that address (this is useful when only minor edits are necessary).

Patch assembly may, at times, cause undesirable side effects:

- ❑ Patching a multiple-word instruction with an instruction of lesser length will leave “garbage” or an unwanted new instruction in the remaining old instruction fragment. This fragment must be patched with either a valid instruction or a NOP, or else unpredictable results may occur when running code.
- ❑ Substituting a larger instruction for a smaller one will partially or entirely overwrite the following instruction; you will lose the instruction and may be left with another fragment.

If you want to insert a large amount of new code or if you want to skip over a section of code, you can use a different patch assembly technique:

- ❑ To insert a large section of new code, patch a branch instruction to go to an area of memory not currently in use. Using the patch assembler, add new code to this area of memory, and branch back to the statement following the initial branch.
- ❑ To skip over a portion of code, patch a branch instruction to go beyond that section of code.

CAUTION

Effects of Patch Assembly

The patch assembler changes only the disassembled assembly language code—it does not change your source code. After determining the correct solution to problems in the disassembly, edit your source file, recompile or reassemble it, and reload the new object file into the debugger.

Additional information about modifying assembly language code

When using patch assembly to modify code in the disassembly window, keep these things in mind:

- **Directives.** You cannot use directives (such as .global or .word).
- **Expressions.** You can use constants, but you cannot use arithmetic expressions. For example, an expression like 12 + 33 is not valid in patch assembly, but a constant such as 12 is allowed.
- **Labels.** You cannot define labels. For example, a statement such as the following is not allowed:

  ```
  LOOP: B LOOP
  ```

  However, an instruction can refer to a label as long as the label is defined in a COFF file that is already loaded.
- **Constants.** You can use hexadecimal, octal, decimal, and binary constants. The syntax to input constants is the same as that for the DSP assembler. (Refer to the *TMS320 Fixed-Point DSP Assembly Language Tools User's Guide.*)
- **Error messages.** The error messages for the patch assembler are the same as the corresponding DSP assembler error messages. Refer to the *TMS320 Fixed-Point DSP Assembly Language Tools User's Guide* for a detailed list of these messages.

2.7 Using Additional MEMORY Windows

The main way to observe memory contents is to view the display in a MEMORY window. Four MEMORY windows are available. The default window, labeled MEMORY, is described in *The Debugger Display* chapter. Three additional windows are called MEMORY1, MEMORY2, and MEMORY3. Notice that the default window does not have an extension number in its name; this is because MEMORY1, MEMORY2, and MEMORY3 are pop-up windows that can be opened and closed throughout your debugging session. Having four windows allows you to view four different memory ranges.

The amount of memory that you can display is limited by the size of the individual MEMORY windows (which is limited only by the screen size). During a debugging session, you may need to display different areas of memory within a window or create an additional MEMORY window.

To create an additional MEMORY window or to display another range of memory in the current window, use the MEM command.

- **Creating a new MEMORY window.**

  If the default MEMORY window is the only MEMORY window open and you want to open another MEMORY window, enter the MEM command with the appropriate extension number:

  mem[#] *address*

  For example, if you want to create a new memory window starting at address 0x8000, you would enter:

  ```
  mem1 0x8000 ↵
  ```

  This displays a new window, MEMORY1, showing the contents of memory starting at the address 0x8000.

  The ’C2x has separate data, program, and I/O spaces. By default, the MEMORY window shows data memory. If you want to display program memory, you can enter the MEM command like this:

  mem[#] *address***@prog**

  The **@prog** suffix identifies the *address* as a program memory address. To display I/O space, use the **@io** suffix. You can also use **@data** to display data memory. However, if you are displaying data memory, the @data is unnecessary since data memory is the default.

- **Displaying a new memory range in the current MEMORY window.**

  Displaying another block of memory identifies a new starting address for the memory range shown in the current MEMORY window. The debugger displays the contents of memory at *address* in the first data position in your MEMORY window. The end of the range is defined by the size of the window.

  If the only MEMORY window open is the default MEMORY window, you can view different memory locations by entering:

  mem *address*

  To view different memory locations in the optional MEMORY windows, use the MEM command with the appropriate extension number on the end. For example:

To do this. . .	Enter this. . .
View the block of memory starting at address 0x8000 in the MEMORY1 window	`mem1 0x8000`
View another block of memory starting at address 0x002f in the MEMORY2 window	`mem2 0x002f`

You can close and reopen additional MEMORY windows as often as you like.

- ❑ **Closing an additional MEMORY window.**

  Closing a window is a two-step process:

  1) Make the appropriate MEMORY window the active window.
  2) Press F4.

  Remember, you cannot close the default MEMORY window.

- ❑ **Reopening an additional MEMORY window.**

  To reopen an additional MEMORY window after you've closed it, enter the MEM command with its appropriate extension number.

2.8 Displaying Data in Alternative Formats

By default, all data is displayed in its natural format. This means that:

- ❑ Integer values are displayed as decimal numbers.
- ❑ Floating-point values are displayed in floating-point format.
- ❑ Pointers are displayed as hexadecimal addresses (with a 0x prefix).
- ❑ Enumerated types are displayed symbolically.

However, any data displayed in the COMMAND, MEMORY, WATCH, or DISP window can be displayed in a variety of formats.

Changing the default format for specific data types

To display specific types of data in a different format, use the SETF command. The syntax for this command is:

setf [*data type*, *display format*]

The *display format* parameter identifies the new display format for any data of type *data type*. Table 2–2 lists the available formats and the corresponding characters that can be used as the *display format* parameter.

Table 2–2. Display Formats for Debugger Data

Display Format	Parameter	Display Format	Parameter
Default for the data type	*****	Hexadecimal	**x**
ASCII character (bytes)	**c**	Octal	**o**
Decimal	**d**	Valid address	**p**
Exponential floating point	**e**	ASCII string	**s**
Decimal floating point	**f**	Unsigned decimal	**u**

Table 2–3 lists the C data types that can be used for the *data type* parameter. Only a subset of the display formats applies to each data type, so Table 2–3 also shows valid combinations of data types and display formats.

Table 2–3. Data Types for Displaying Debugger Data

	Valid Display Formats									
Data Type	**c**	**d**	**o**	**x**	**e**	**f**	**p**	**s**	**u**	**Default Display Format**
char	√	√	√	√					√	ASCII (c)
uchar	√	√	√	√					√	Decimal (d)
short	√	√	√	√					√	Decimal (d)
int	√	√	√	√					√	Decimal (d)
uint	√	√	√	√					√	Decimal (d)
long	√	√	√	√					√	Decimal (d)
ulong	√	√	√	√					√	Decimal (d)
float			√	√	√	√				Exponential floating point (e)
double			√	√	√	√				Exponential floating point (e)
ptr			√	√			√	√		Address (p)

Here are some examples:

- ❑ To display all data of type short as an unsigned decimal, enter:

  `setf short, u`

- ❑ To return all data of type short to its default display format, enter:

  `setf short, *`

- ❑ To list the current display formats for each data type, enter the SETF command with no parameters:

  `setf`

  You'll see a display that looks something like this:

```
           Display Format Defaults
     Type char:           ASCII
     Type unsigned char:  Decimal
     Type int:            Decimal
     Type unsigned int:   Decimal
     Type short:          Decimal
     Type unsigned short: Decimal
     Type long:           Decimal
     Type unsigned long:  Decimal
     Type float:          Exponential floating point
     Type double:         Exponential floating point
     Type pointer:        Address
```

- ❑ To reset all data types back to their default display formats, enter:

  `setf *`

Changing the default format with ?, MEM, DISP, and WA

You can also use the ?, MEM, DISP, and WA commands to show data in alternative display formats. (The ? and DISP commands can use alternative formats only for scalar types, arrays of scalar types, and individual members of aggregate types.)

Each of these commands has an optional *display format* parameter that works in the same way as the *display format* parameter of the SETF command.

When you don't use a *display format* parameter, data is shown in its natural format (unless you have changed the format for the data type with SETF).

Here are some examples:

- To watch the PC in decimal, enter:

  ```
  wa pc,,d
  ```

- To display memory contents in octal, enter:

  ```
  mem 0x0,o
  ```

- To display an array of integers as characters, enter:

  ```
  disp ai,c
  ```

The valid combinations of data types and display formats listed for SETF also apply to the data displayed with DISP, ?, WA, and MEM. For example, if you want to use display format **e** or **f**, the data that you are displaying must be of type float or type double. Additionally, you cannot use the **s** display format parameter with the MEM command.

For more information about using the DISP, ?, WA, and MEM commands, refer to the *Managing Data* chapter of the *TMS320C2x C Source Debugger User's Guide.*

2.9 Associating Sound With Error Messages

You can associate a beeping sound with the display of error messages. To do this, use the SOUND command. The format for this command is:

sound on | off

By default, no beep is associated with error messages (SOUND OFF). The beep is helpful if the COMMAND window is hidden behind other windows.

Chapter 3

Documentation Errors in the *TMS320C2x C Source Debugger User’s Guide*

This chapter identifies and corrects the documentation errors or modifications in the *TMS320C2x C Source Debugger User’s Guide.*

3.1 Entering Text in a Dialog Box

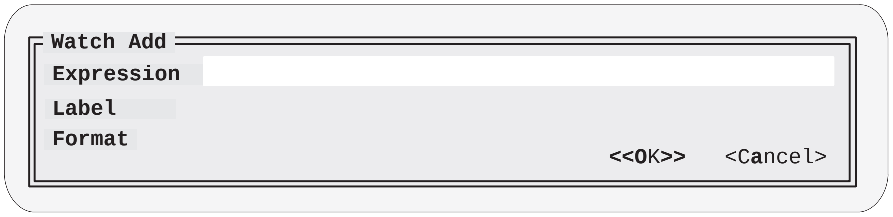
The *Entering parameters in a dialog box* subsection has changed. The following information replaces it.

Many of the debugger commands have parameters. When you execute these commands from pulldown menus, you must have some way of providing parameter information. The debugger allows you to do this by displaying a **dialog box** that asks for this information.

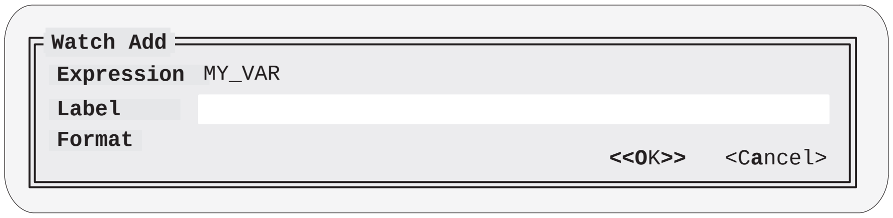
Entering text in a dialog box is much like entering commands on the command line. For example, the Add entry on the Watch menu is equivalent to entering the WA command. This command has three parameters:

wa *expression* [,[*label*] [, *display format*]]

When you select Add from the Watch menu, the debugger displays a dialog box that asks you for this parameter information. The dialog box looks like this:



You can enter an *expression* just as you would if you were to type the WA command; then press TAB or ↓. The cursor moves down to the next parameter:



When the dialog box displays more than one parameter, you can use the arrow keys to move from parameter to parameter. You can omit entries for optional parameters, but the debugger won't allow you to skip required parameters.

In the case of the WA command, the two parameters, *label* and *format*, are optional. If you want to enter a parameter, you may do so; if you don't want to use these optional parameters, don't type anything in their fields—just continue to the next parameter.

Modifying text in a dialog box is similar to editing text on the command line:

- When you display a dialog box for the first time during a debugging session, the parameter fields are empty. When you bring up the same dialog box again, though, the box displays the last values that you entered. (This is similar to having a command history.) If you want to use the same value, just press TAB or ↓ to move to the next parameter.
- You can edit what you type (or values that remain from a previous entry) in the same way that you can edit text on the command line. (See the *How to type in and enter commands* subsection in the *TMS320C2x C Source Debugger User's Guide*.)

When you've entered a value for the final parameter, point and click on <OK> to save your changes, or <Cancel> to discard your changes; the debugger closes the dialog box and executes the command with the parameter values you supplied.

3.2 Filling a Block of Memory

On page 9-10 of the *TMS320C2x C Source Debugger User's Guide*, the basic syntax for the FILL command is shown incorrectly; the correct syntax is as follows:

fill *address, page, length, data*

- The *address* parameter identifies the first address in the block.
- The *page* is a 1-digit number that identifies the type of memory (program, data, or I/O) to fill:

To fill this type of memory	**Use this value as the *page* parameter**
Program memory	**0**
Data memory	**1**
I/O space	**2**

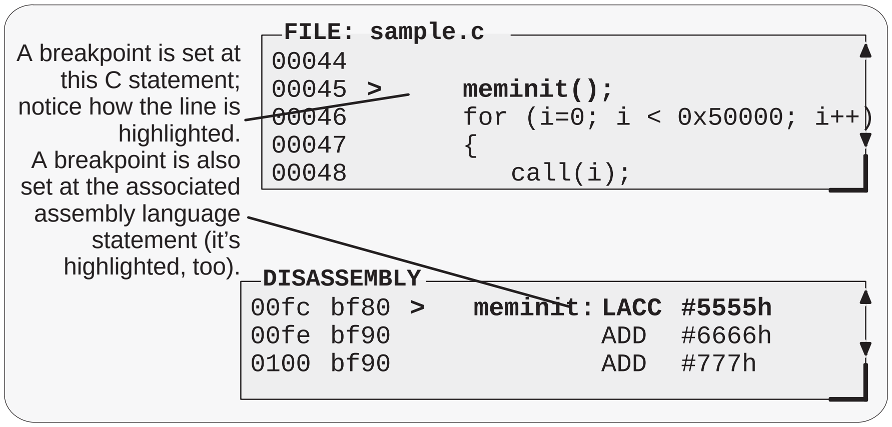
- The *length* parameter defines the number of words to fill.
- The *data* parameter is the value that is placed in each word in the block.

For example, to fill program memory locations 0x10FF–0x110D with the value 0xABCD, you would enter:

```
fill 0x10ff,0,0xf,0xabcd
```

3.3 Identifying Breakpoints That Are Set

On page 10-2 of the *TMS320C2x C Source Debugger User's Guide*, it was documented that breakpointed statements are highlighted with a BP> label and are shown in a brighter, heavier font. Breakpointed statements are now highlighted with a > character, along with the bright, heavy font:


A breakpoint is set at this C statement; notice how the line is highlighted.
A breakpoint is also set at the associated assembly language statement (it's highlighted, too).
FILE: sample.c
DISASSEMBLY

```
00044
00045 >     meminit();
00046       for (i=0; i < 0x50000; i++)
00047       {
00048          call(i);
```

```
00fc bf80 >   meminit: LACC  #5555h
00fe bf90              ADD   #6666h
0100 bf90              ADD   #777h
```

3.4 Resizing a Window

On page 5-21 of the *TMS320C2x C Source Debugger User's Guide*, the information about using the keyboard to size a window has changed. You have only two debugger options for specifying the screen size (see page 1-13). As a result, the following information replaces the **SIZE, method 1** section in the user's guide:

SIZE, method 1: Use the *width* and *length* parameters. Valid values for the width and length depend on the screen size and the window position on the screen. If the window is in the upper left corner of the screen, the maximum size of the window is the same as the screen size minus one line. (The extra line is needed for the menu bar.) For example, if the screen size is 80 characters by 25 lines, the largest window size is 80 characters by 24 lines.

If a window is in the middle of the display, you can't size it to the maximum height and width—you can size it only to the right and bottom screen borders. The easiest way to make a window as large as possible is to zoom it, as described on page 2-2.

3.5 Moving a Window

On page 5-23 of the *TMS320C2x C Source Debugger User's Guide*, the information about using the keyboard to move a window has changed. You have only two debugger options for specifying the screen size (see page 1-13). As a result, the following information replaces the **MOVE, method 1** section in the user's guide:

MOVE, method 1: Use the *X position* and *Y position* parameters. You can move a window by defining a new XY position for the window's upper left corner. Valid X and Y positions depend on the screen size and the window size. X positions are valid if the X position plus the window width in characters is less than or equal to the screen width in characters. Y positions are valid if the Y position plus the widow height is less than or equal to the screen height in lines.

For example, if the window is 10 characters wide and 5 lines high and the screen size is 80 x 25, the command **move 70, 20** would put the lower right-hand corner of the window in the lower right-hand corner of the screen. No X value greater than 70 or Y value greater than 20 would be valid in this example.

3.6 Using SCONFIG Files

The file created by the SSAVE command in this version of the debugger saves positional, screen size, and video mode information that was not saved by SSAVE in previous versions of the debugger. The format of this new information is not compatible with the old format. Do not use the SCONFIG files from previous debugger versions; if you attempt to load an earlier version's SCONFIG file, the debugger will issue an error message and stop the load.

3.7 Setting the Characteristics of a Memory Range

With the EVM, SWDS, or simulator, you can set the memory specifications for program memory, data memory, or I/O space. The following information corrects areas of the *TMS320C2x C Source Debugger User's Guide*:

❏ On page 7-6, the MA command has a parameter called *type*. The *type* can be any of the keywords listed in the table; types IPORT, OPORT, and IOPORT are *not* restricted to the simulator.

- ❑ In the syntax description for the MS command (page 9-9), the *page* parameter information should be as follows:
  - ❑ The *page* is a 1-digit number that identifies the type of memory (program, data, or I/O) to save:

To save this type of memory	Use this value as the *page* parameter
Program memory	**0**
Data memory	**1**
I/O space	**2**

- ❑ The syntax for the ADDR command (page 12-7), has changed:

  addr *address*[@**prog** | @**data** | @**io**]
  addr *function name*

  By default, the *address* parameter is treated as a program-memory address. However, you can follow it with @**prog** to identify program memory, @**data** to identify data memory, or @**io** to identify I/O space.

- ❑ The syntax for the EVAL command (page 12-15), has changed:

  eval *expression*[@**prog** | @**data** | @**io**]
  e *expression*[@**prog** | @**data** | @**io**]

  If the *expression* identifies an address, you can follow it with @**prog** to identify program memory, @**data** to identify data memory, or @**io** to identify I/O space. Without the suffix, the debugger treats an address expression as a data-memory location.

- ❑ In the syntax description for the FILL command (page 12-16), the *page* parameter information should be as follows:
  - ❑ The *page* is a 1-digit number that identifies the type of memory (program, data, or I/O) to fill:

To fill this type of memory	Use this value as the *page* parameter
Program memory	**0**
Data memory	**1**
I/O space	**2**

3.8 Using the ?, DISP, MEM, and WA Commands

The descriptions for the ?, DISP, MEM, and WA commands have changed. The following information replaces the information in the *TMS320C2x C Source Debugger User's Guide.*

?

Evaluate Expression

Syntax **?** *expression*[**@prog** | **@data** | **@io**] [*, display format*]

Menu selection none

Description The ? (evaluate expression) command evaluates an expression and shows the result in the COMMAND window display area. The *expression* can be any C expression, including an expression with side effects; however, you cannot use a string constant or function call in the *expression.*

If the *expression* identifies an address, you can follow it with **@prog** to identify program memory, **@data** to identify data memory, or **@io** to identify I/O space. Without the suffix, the debugger treats an address expression as a data-memory location.

If the result of *expression* is not an array or structure, then the debugger displays the results in the COMMAND window. If *expression* is a structure or array, ? displays the entire contents of the structure or array; you can halt long listings by pressing ESC.

When you use the optional *display format* parameter, data will be displayed in one of the following formats:

Parameter	Result	Parameter	Result
*****	Default for the data type	**x**	Hexadecimal
c	ASCII character (bytes)	**o**	Octal
d	Decimal	**p**	Valid address
e	Exponential floating point	**s**	ASCII string
f	Decimal floating point	**u**	Unsigned decimal

disp

Open DISP Window

Syntax **disp** *expression*[**@prog** | **@data** | **@io**] [*, display format*]

Menu selection none

Description The DISP command opens a DISP window to display the contents of an array, structure, or pointer expressions to a scalar type (of the form **pointer*). If the *expression* is not one of these types, then DISP acts like a ? command.

If the *expression* identifies an address, you can follow it with **@prog** to identify program memory, **@data** to identify data memory, or **@io** to identify I/O space. Without the suffix, the debugger treats an address expression as a data-memory location.

Once you open a DISP window, you may find that a displayed member is itself an array, structure, or pointer:

A member that is an array looks like this	[. . .]
A member that is a structure looks like this	{. . .}
A member that is a pointer looks like an address	0x0000

You can display the additional data (the data pointed to or the members of the array or structure) in another DISP window by using the DISP command again, using the arrow keys to select the field and then pressing F9, or pointing the mouse cursor to the field and pressing the left mouse button. You can have up to 120 DISP windows open at the same time.

When you use the optional *display format* parameter, data will be displayed in one of the following formats:

Parameter	Result	Parameter	Result
*****	Default for the data type	**x**	Hexadecimal
c	ASCII character (bytes)	**o**	Octal
d	Decimal	**p**	Valid address
e	Exponential floating point	**s**	ASCII string
f	Decimal floating point	**u**	Unsigned decimal

The *display format* parameter can be used only when you are displaying a scalar type, an array of scalar type, or an individual member of an aggregate type.

You can also use the DISP command with a typecast expression to display memory contents in any format. Here are some examples:

```
disp *0
disp *(float *)123
disp *(char *)0x111
```

This shows memory in the DISP window as an array of locations; the location that you specify with the *expression* parameter is member [0], and all other locations are offset from that location.

mem — *Modify MEMORY Window Display*

Syntax

mem[#]   *expression* [, *display format*]

Menu selection

none

Description

The MEM command identifies a new starting address for the block of memory displayed in the MEMORY window. The optional extension number (#) opens an additional MEMORY window allowing you to view a separate block of memory. The debugger displays the contents of memory at *expression* in the first data position in the MEMORY window. The end of the range is defined by the size of the window. The *expression* can be an absolute address, a symbolic address, or any C expression.

You can display program, data, or I/O memory:

❑ By default, the MEMORY window displays data memory. Although it is not necessary, you can explicitly specify data memory by following the *expression* parameter with a suffix of **@data**.

❑ You can display the contents of program memory by following the *expression* parameter with a suffix of **@prog**. When you do this, the MEMORY window's label changes to MEMORY [PROG] so that there is no confusion about the type of memory being displayed.

❑ You can display the contents of the I/O space by following the *expression* parameter with a suffix of **@io**. When you do this, the MEMORY window's label changes to MEMORY [IO] so that there is no confusion about the type of memory being displayed.

When you use the optional *display format* parameter, memory will be displayed in one of the following formats:

Parameter	Result	Parameter	Result
*****	Default for the data type	**x**	Hexadecimal
c	ASCII character (bytes)	**o**	Octal
d	Decimal	**p**	Valid address
e	Exponential floating point	**u**	Unsigned decimal
f	Decimal floating point		

wa — *Add Item to WATCH Window*

Syntax **wa** *expression*[**@prog** | **@data** | **@io**] [,[*label*] [, *display format*]]

Menu selection **W**atch→**A**dd

Description The WA command displays the value of *expression* in the WATCH window. If the WATCH window isn't open, executing WA opens the WATCH window. The *expression* parameter can be any C expression, including an expression that has side effects.

If the *expression* identifies an address, you can follow it with **@prog** to identify program memory, **@data** to identify data memory, or **@io** to identify I/O space. Without the suffix, the debugger treats an address expression as a data-memory location.

WA is most useful for watching an expression whose value changes over time; constant expressions provide no useful function in the watch window. The *label* parameter is optional. When used, it provides a label for the watched entry. If you don't use a *label*, the debugger displays the *expression* in the label field.

When you use the optional *display format* parameter, data will be displayed in one of the following formats:

Parameter	Result	Parameter	Result
*	Default for the data type	x	Hexadecimal
c	ASCII character (bytes)	o	Octal
d	Decimal	p	Valid address
e	Exponential floating point	s	ASCII string
f	Decimal floating point	u	Unsigned decimal

If you want to use a *display format* parameter without a *label* parameter, just insert an extra comma. For example:

```
wa PC,,d
```