

Book 4—Software Tools

Part C:

TriMedia Debugger

Table of Contents

Chapter 15 Introduction to Debugging

Overview of the TriMedia Debugger (tmdbg)	10
Command-Line Version	10
Graphical User Interface Version	10
Backward Compatibility	10
TriMedia Debugger Architecture	12
Front End	12
Target-Driver	12
Debug Monitor	13
Installation Requirements	13
Getting Started Quickly	14
Other Debugging Tools: printf and DP	14

Chapter 16 Debugging Standard TriMedia Applications

Preparing Programs for Debugging.....	16
Generating Symbolic Debugging Information	17
Linking the Debug Monitor Library	17
Using the -g Option	17
Using the TriMedia Linker.....	17
Using the Debugger	18
Starting the Command-Line Version of tmdbg	18
Listing tmdbg Commands	18
Getting Help about a Specific Command.....	19
Listing the Source Code	19
Source Code Line Components.....	20
Starting the GUI Version of tmdbg	20
Controlling Program Execution.....	23
Single Stepping	23
Stepping Across Functions	23
Stepping Out of Functions	24

Using Breakpoints.....	24
Setting Breakpoints	24
Setting Software Instruction Breakpoints	24
Setting Hardware Data Breakpoints.....	26
Setting Hardware Instruction Breakpoints	29
Removing Breakpoints	30
tmdbg Breakpoint Anomalies	31
Software Workaround.....	33
Examining and Changing Data, Memory, and Registers	33
Symbolic Access	33
Expression Evaluation	34
Assignment	35
Nonsymbolic Access	35
Tracing the Call Stack.....	39
Moving Up and Down the Call Stack	39
Disassembling Instructions.....	41
Using tmdbg with tmsim on Windows 95	42

Chapter 17 Debugging C++ Code with tmdbg

Introduction	44
Accessing C++ Definitions.....	44
Accessing Class Members.....	45
Accessing C++ Variable Declarations	47
Setting Breakpoints.....	47
Accessing Virtual Functions	48
Debugging Templates.....	49
Unsupported Features	50
Non-ANSI Compliant Names.....	52
Notes	53

Chapter 18 Debugging TriMedia Applications Using JTAG

Introduction	56
Stand-Alone Debugging Modes	57
No-Host Mode	57
Host-Assisted	58
System Requirements	59
Hardware	59
Software	60
Setting Up the System for Stand-Alone Debugging	61
Testing the JTAG Connection.....	62
Testing the Corelis Board JTAG Connection	63
Sample Session	63
Testing the Turbo Board JTAG Connection	64
Sample Session	64
Compiling a Program for Stand-Alone Debugging	64
Debugging Stand-Alone TriMedia Applications	65
Debugging with No-Host	65
JTAG Debugging in Host-Assisted Mode	67
Sample Host-Assisted JTAG Debugging Session	67
Multiple Debugging Sessions.....	68

Chapter 19 Debugging TriMedia pSOS+™ Applications

Introduction	70
pROBE Functionality in tmdbg —The pSOS+ Monitor	70
Setting Up a pSOS+ Application for Use with tmdbg	70
Inspecting pSOS+ Objects	70
Query pSOS+ Configuration	71
Query Date	72
Query Object	72
Query Partition	72
Query Queue	73
Query Region	73
Query Semaphore	74
Query Task	74

Getting Profile Information of a pSOS+ Application	74
pSOS+ Breakpoints.....	76
Breakpoint on pSOS+ System Calls.....	76
Breakpoints on Task Scheduling	76
Deleting pSOS+ Breakpoints.....	77
Callout Functions	77
Print	78
Pitfalls.....	79
Chapter 20 Debugging TriMedia Applications Using printf and DP	
Introduction.....	82
Debugging Using printf.....	82
Comparing DP and printf	82
Using DP	83
Description of DP Macros	84
Chapter 21 Debugging Multiprocessor and Multitasking Applications	
Introduction.....	86
Loading a Multiprocessor Application.....	86
Switching Focus	87
The procs Command	87
Debugging Tasks.....	87
Debugging Tasks in System Mode	88
Debugging Tasks in the Task Mode	89
Chapter 22 Other Debugging Information	
Debugging Interrupt Handlers	92
Interrupt Handler Characteristics and Problems	92
Interrupt Handler Solutions	93
Setting Breakpoints	93
Debugging the Host Call Interface and Device Library	93
Debugging at Optimization Levels Higher Than -O1	94
Diminished Visibility of Local Variables and Parameters	94
Variables May Have Been Optimized Away	95

Code May Have Been Moved	95
Code May Have Been Unrolled	95
Code May Have Been Inlined	95

Chapter 23 tmdbg Command Reference

Overview of Debugger Commands.....	98
Debugger Expressions.....	98
Debugger Commands.....	98
Execution Control Commands	99
Data and Stack Commands	100
Source File Commands	103
pSOS+ Commands	104
Task-Level Debugging Commands	105
Multi-Processor Debugging Commands	105
Miscellaneous Commands	105

Chapter 24 Code Listings

x.c	110
foo.c	110
d.h	111
d1.c	112
d2.c	113

Chapter 15

Introduction to Debugging

Topic	Page
Overview of the TriMedia Debugger (tmdbg)	10
TriMedia Debugger Architecture	12
Installation Requirements	13
Getting Started Quickly	14
Other Debugging Tools: printf and DP	14

Overview of the TriMedia Debugger (**tmdbg**)

The TriMedia Source-Level Debugger (**tmdbg**) is the main tool for debugging TriMedia applications. **tmdbg** is an interactive debugging tool that enables you to run a program with interactive user control and to inspect and modify the state of a stopped program.

tmdbg provides complete control over the execution of a program. It enables you to view the values of variables and expressions, set breakpoints in the code, and run and trace a program. It is available in the following two flavors:

- Command-line
- GUI

Command-Line Version

The command-line version of **tmdbg** runs on all supported platforms (Windows 95, Windows NT, and UNIX). It provides a set of commands that enable you to perform the different debugging tasks. A complete debugger command reference is available in Chapter 23, *tmdbg Command Reference*.

Graphical User Interface Version

The current graphical user interface (GUI) version of **tmdbg** runs on all supported platforms in the same way that the command-line version and provides an easy-to-use GUI that allows you perform most of the debugging actions with the mouse. This reduces your learning time and enables you to focus on debugging, rather than having to memorize and type complex commands.

Note

The Messages window in the GUI version of tmdbg allows you to enter commands as if you were using tmdbg's command-line version.

Backward Compatibility

Object code, trees code, or assembly code created by compiling with the **-g** option on the 1.1 or 2.0a compilers should not be used with the 2.0b debugger. This restriction applies even if the code is scheduled, assembled, or linked by the 2.0b compiler with other 2.0b object.

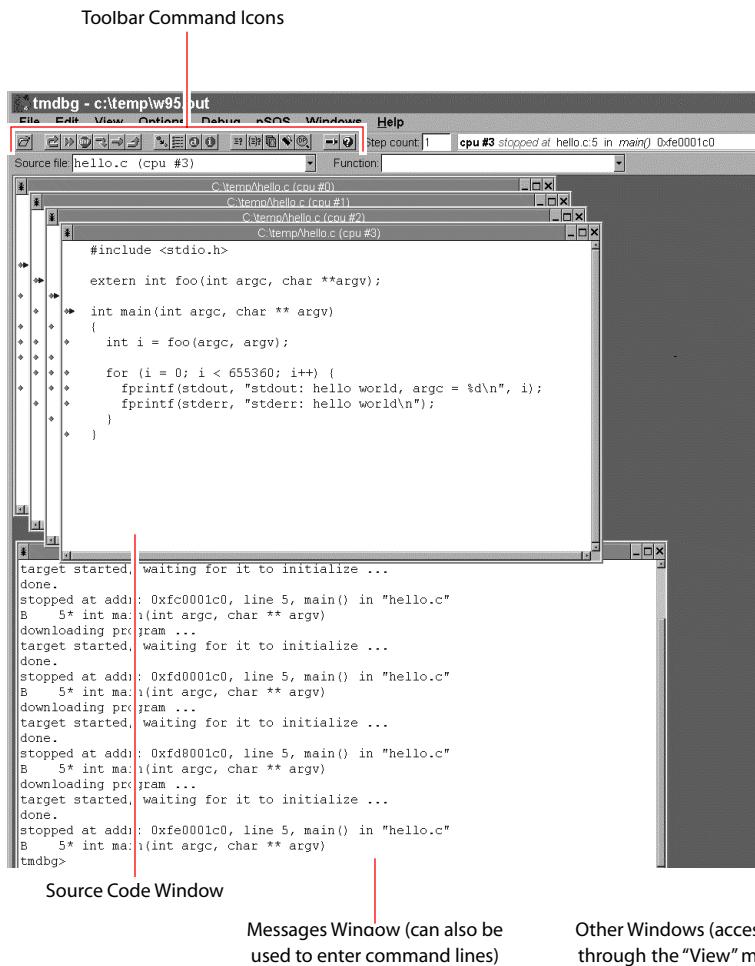


Figure 1 tmdbg's user interface

Note

The look-and-feel of the user interface may differ depending on the platform you are using. The screen captures in this chapter were taken on a Windows 95 platform.

TriMedia Debugger Architecture

The TriMedia Debugger (**tmdbg**) works with different host drivers and dynamic loaders. It can be used on a stand-alone system or on a PC-based system. Figure 2 shows the three major components of the TriMedia Debugger Architecture: front-end, target-driver, and debug monitor.

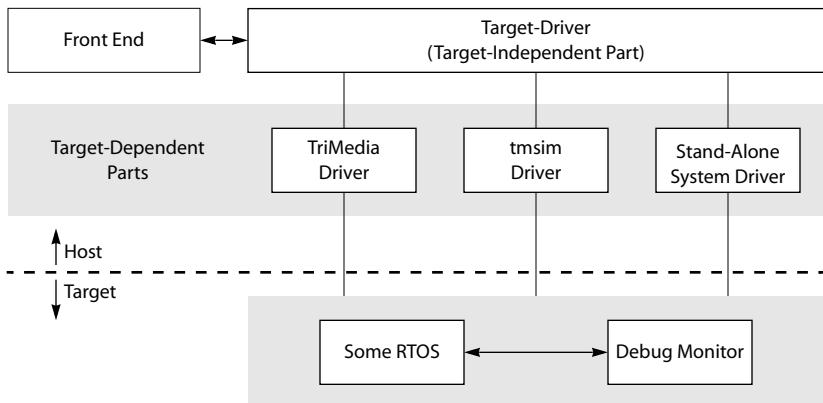


Figure 2 TriMedia debugger architecture

Front End

The front end and the target-driver components run on the host, while the debug monitor runs on the target. The front end includes the following:

- Command-line parser
- GUI
- Expression evaluator
- Symbol table module
- Object file reader

Target-Driver

The target-driver is the back end of **tmdbg** and runs on the host. A target system includes a TriMedia board, possibly a RTOS, and some host-based drivers used for loading, communications, and so forth. The target-driver has two parts: a target-dependent part and a target-system-independent part. The target-dependent part (also called the *device driver*) is provided by TriMedia, or another vendor developing a system that includes a TriMedia microprocessor.

Because many target-dependent drivers may exist on a system, **tmdbg** provides the following command-line options to choose the target system:

```
tmdbg -target {tmsim | tm1} a.out
cuv] library [ object ... ]
o abctmld -eb a.o b.o libc.a c.o
.o a.o b.o c.o
```

Debug Monitor

The debug monitor runs on the target on which an RTOS might or might not be running. The monitor sends messages to the host-resident debugger in different ways:

- When you use **tmdbg** with **tmsim**, the debug monitor uses sockets to send messages.
- When you use **tmdbg** to debug programs on TriMedia reference boards (plugged into a PCI slot on a PC or a Mac), the debug monitor uses the shared memory on the host.
- For passing messages to the stand-alone systems, **tmdbg** communicates with the debug monitor through the Joint Test Action Group (JTAG) port. The TriMedia architecture provides two MMIO registers to be used as input and output buffers, and one MMIO control register to be used for handshake. The JTAG registers on the chip are used solely as a communication mechanism for the debugger.

A common message-passing layer is implemented on top of the low-level data-transfer mechanisms such as JTAG, PCI, and sockets. **tmdbg** provides the same functionality (namely source-level debugging and low-level access to registers, memory, and so on) whether the target is **tmsim** or a TriMedia board connected via PCI or JTAG.

Installation Requirements

To use **tmdbg**, you must perform a *complete* installation of the TriMedia SDE as described in [Book 1, Getting Started with Philips TriMedia](#). If you perform only a light installation and try to use **tmdbg**, you may get error messages.

Note

tmdbg will report a **gethostbyname** failure if it cannot find an IP address for the host machine. See [A Note on Installation Requirements For the TriMedia Debugger](#) in Chapter 2 of Book 1, *Getting Started with Philips TriMedia*.

Getting Started Quickly

If you want to start using the debugger without having to read the debugger documentation, do the following:

1. Compile the program you want to debug with the **-g** option.
2. Start the debugger by performing one of the following steps:
 - Enter **tmdbg -cli** at the command line or DOS prompt to launch the command-line version of **tmdbg**.
 - Double-click the tmdbg.exe icon (c:\<Installation_Folder_Name>\bin\) in Windows 95 and Windows NT platforms to launch the GUI version of **tmdbg**.
 - Enter **tmdbg** at the command line or DOS prompt to launch the GUI version of the debugger. See Chapter 23 for a **tmdbg** command reference.
3. If you are using the command-line version of **tmdbg**, enter **help** at **tmdbg**'s command-line prompt to learn how to use **tmdbg**'s commands.

Other Debugging Tools: printf and DP

Chapter 20, *Debugging TriMedia Applications Using printf and DP*, describes two important techniques, *printf* and *Debug Print (DP)*, that you can use to debug TriMedia programs when not using **tmdbg**.

Chapter 16

Debugging Standard TriMedia Applications

Topic	Page
Preparing Programs for Debugging	16
Using the Debugger	18
Controlling Program Execution	23
Using Breakpoints	24
Examining and Changing Data, Memory, and Registers	33
Tracing the Call Stack	39
Disassembling Instructions	41
Using tmdbg with tmsim on Windows 95	42

Preparing Programs for Debugging

To be able to use **tmdbg** to debug applications, you must do the following:

1. Generate the symbolic debugging information.
 2. Link with the debug monitor library (libmon.o).
 3. Specify the debugging target environment using the **-target** option:
 - **-target tm1000** for the TM-1000 chip
 4. Specify the host using the **-host** option:
 - **-host Win95** for Windows 95 platforms
 - **-host WinNT** for Windows NT platforms
 - **-host nohost** for stand-alone systems
 - **-host tmsim** for the simulator

By default, the execution host is the simulator

- #### 5. Specify the optimization level.

By default, the compiler performs level 1 optimization. The only other level of optimization that you can specify when using the `-g` option is level 2. You do this by using `tmcc`'s `-O2` option.

This is all done automatically when you compile your program using the `-g`, `-target`, and `-O2` options, as shown in Figure 3.

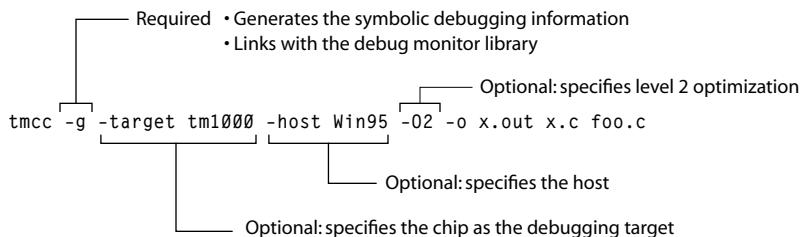


Figure 3 Compiler options required to generate the debug information

Note

tmdbg provides no debugging support for any part of a program not compiled with the **-g** option (for example, routines linked in from the standard C library).

The following sections describe the debugging preparation steps.

Generating Symbolic Debugging Information

Using the **-g** option, the compiler driver (**tmcc**) automatically generates symbolic debugging information. The **tmdbg** program uses this information to inspect and modify the state of the debugged program.

The symbolic debugging information contains source file paths and a wide range of source code-related information such as types and scope information of variables.

Linking the Debug Monitor Library

You can link the debug monitor library in one of the following two ways:

- Using **tmcc**'s **-g** option
- Using the TriMedia linker **tmld**

IMPORTANT

tmld should generally not be used to link applications directly. **tmcc** specifies information to **tmld** that is essential for the application to work. This includes the boot type, endianness, runtime startup, and libraries. If there is a user need to use **tmld** directly, **tmcc -v** should be used first to find out the required elements.

Using the **-g** Option

Using the **-g** option, the compiler driver (**tmcc**) automatically links the program with the appropriate version of the debug monitor library (*libmon.o*), depending on whether it is operating in little-endian or big-endian mode.

Using the TriMedia Linker

Using the TriMedia linker (**tmld**) directly, you must link the appropriate version of *libmon.o* (little-endian or big-endian). Little-endian and big endian versions of *libmon.o* are located in the following directories:

Debug Monitor Library Version	Directory
Little-endian	\$INSTALL_DIR/lib/el
Big-endian	\$INSTALL_DIR/lib/eb

Using the Debugger

The TriMedia debugger (**tmdbg**) can be used with either its command-line version, or its GUI version.

Starting the Command-Line Version of tmdbg

To debug a program (x.out for example) using the command-line version of the debugger, do the following:

1. Enter the following at the shell prompt:

```
tmdbg -cli
```

This starts the TriMedia debugger (**tmdbg**) and the following message appears:

```
TriMedia C debugger tmdbg
v0.69 of SunOS (Jul 2 1997 16:21:21)

no target program loaded
```

2. Load the x.out program using the **load** command. **tmdbg** automatically determines the debugging target (the simulator or the TriMedia chip) from the executable and downloads the program to **tmsim** or the chip.

```
tmdbg> load x.out
downloading program ...
target started, waiting for it to initialize ...
done.
Stopped at addr: 0x001000cd line: 31, main() in "x.c"
B 31* int main(int argc, char *argv[])
tmdbg>
```

3. After it loads the program, the debugger is in a ready state, at the beginning of the **main** function of the x.c.

Note

If tmdbg encounters problems while loading the program, it reports the problem by issuing an appropriate message.

IMPORTANT

The current version of tmdbg generates error messages when it attempts to demangle C identifier names that start with “_0”. Such names are non-ANSI compliant, but they are valid, which means that tmcc does not flag them as errors. Therefore, it is highly recommended *not* to use the “_0” prefix in C identifier names (C++ identifier names can have this prefix).

Listing tmdbg Commands

Use the **help** command, or refer to Chapter 23, to see a listing of **tmdbg**'s commands.

Getting Help about a Specific Command

To list the syntax for the **br** command, for example, enter the following:

```
tmdbg> help br
BREAK - Set a source-level breakpoint (same as the stop command).

break <line>                      Stop execution at the line
break <line> <file>                Stop execution at the line of <file>
break <func>                       Stop when <func> of current file is called
break addr <addr>                 Stop execution at the given address

break <line> when <cond>          Stop at line when <cond> is true
break <line> <file> when <cond>  Stop at line of <file> when <cond> is true
break <func> when <cond>           Stop when <func> is called and <cond> is true
break addr <addr> when <cond>   Stop at <addr> when <cond> is true
```

Note: conditions are evaluated in the context of the breakpoint location.

Listing the Source Code

One of the commands that you will use very often is the **list** command. This command lists the code of the source file that is currently visited.

For example, to list the code of the x.c program, enter the following:

```
tmdbg> list
```

tmdbg displays the following:

```
32  {
33      noot y;
34      static int z = 42;
35      fun f = &main;
36
B 37      y.x = foo(f, bar(x.x));
38      printf ("%d %d\n", x.x, x.y);
B 39      printf ("%x %x %x %x %x\n", y.x, y.y[0], y.y[1], y.y[2], y.y[3]);
B 40      return z;
41  }
42
```

In this example, the **list** command (you can also use **l**) lists the source code of the x.c file starting at the line 32 (the line following the line where **tmdbg** stopped the execution of x.out).

To list the code in a specified range, enter the following:

```
tmdbg> l 30 35
```

tmdbg displays the following:

```
30
B 31*  int main(int argc, char *argv[])
32  {
33      noot y;
34      static int z = 42;
35      fun f = &main;
```

Source Code Line Components

tmdbg adds the following indicators to the code lines of the source file (see Figure 4):

- Line number indicating the sequential order of program lines (starting at 1).
- Decision tree marker (B) indicating the beginning of a decision tree.
- Asterisk (*) indicating the current Program Counter (PC). The PC points to the line where the program is stopped.

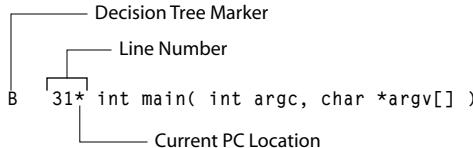


Figure 4 Source code line components

Starting the GUI Version of **tmdbg**

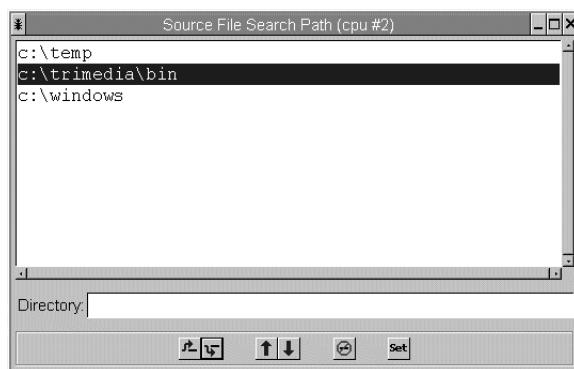
To debug the a TriMedia program using the GUI version of the debugger, do the following:

1. Enter **tmdbg** at the shell prompt (or doubleclick the **tmdbg** icon).



The **tmdbg** window appears, as shown below.

2. Choose Set Search Path from the Options menu.



Enter the paths that you want **tmdbg** to use when searching for source files. Use the Add button to add new paths and the Delete button to delete existing paths. Click Set when finished and close the Source File Search Path dialog box.

3. Choose Load to Target from the File menu.

The Load Target window appears, as shown below.

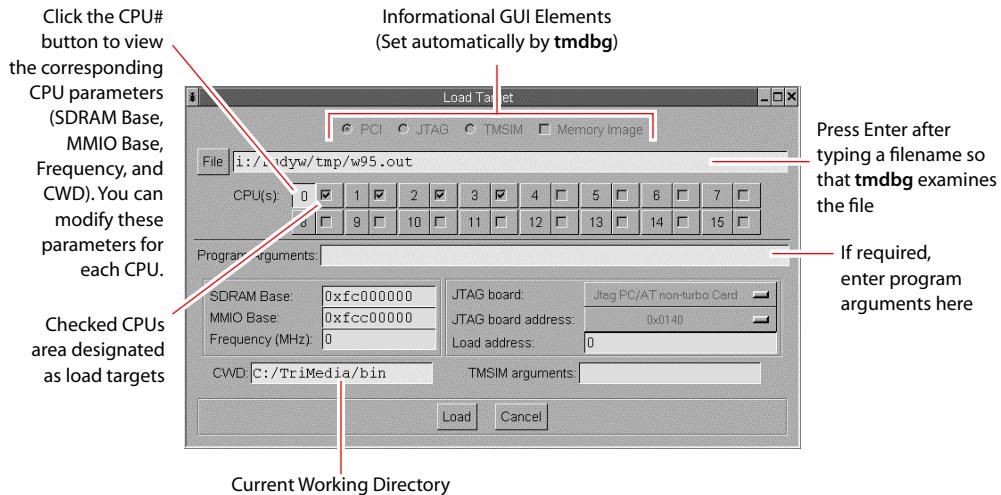


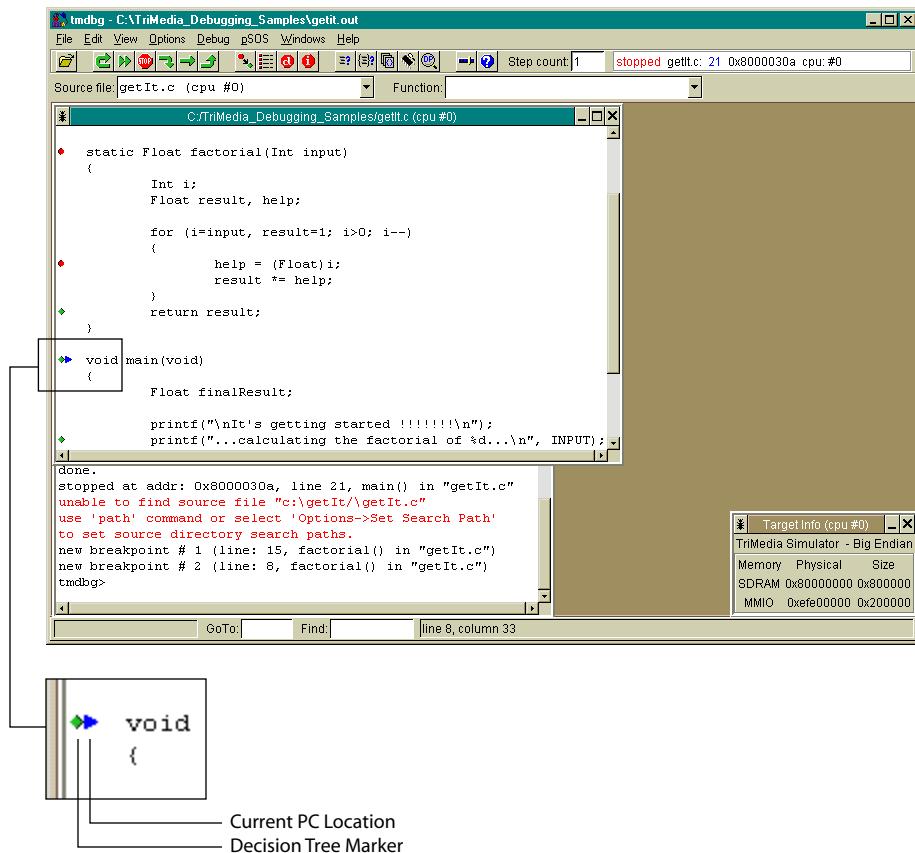
Figure 5 Load To Target Window

4. Click Load.

tmdbg examines the file to be loaded.

5. Click on the File button and locate the TriMedia executable (.out file).

The tmrun or tmsim window appears (depending on the compilation target) and **tmdbg** loads the source files of target program.



- A source code window appears containing the code where **tmdbg** stopped. A blue right-arrow points to the line where execution stopped (current PC location). The green dot to its left is a decision tree marker (where you can place breakpoints).
- The Source Files window contains a list of all source files. Clicking an item in the window opens a new source code window containing the code of the selected source file.
- The Messages window displays the status of the program. A summary of the status appears in the status bar.

Controlling Program Execution

tmdbg allows you to control your program's execution with the following commands:

Command-line	GUI	
run		Although the information in this section is based on the command-line version of tmdbg , the same applies to the GUI version of tmdbg . Instead of typing commands, you click or choose the corresponding buttons or menu options.
cont		
stop		
step		
next		
finish		

This section describes only the **step**, **next**, and **finish** commands.

Single Stepping

To single-step through the source code, but not across functions, use the **step** command (also **s**). Each time you use this command, **tmdbg** resumes execution until it reaches the next decision tree that may be in the same function or in another function. Following is an example:

```
tmdbg> s
Stopped at addr: 0x001002c3 line: 5, foo() in "foo.c"
B 5* int
tmdbg> s
Stopped at addr: 0x00100192 line: 37, main() in "x.c"
B 37*     y.x = foo(f, bar(x.x));
```

Stepping Across Functions

To step across functions, use the **next** command (also **n**). Each time you use this command, **tmdbg** steps over intervening function calls (if any), as shown here.

```
tmdbg d.out
      TriMedia C debugger tmdbg
      v0.61 of SunOS (Apr 30 1997 12:10:32)

downloading program ...
Target started. Waiting for it to initialize ...
done.
Stopped at addr: 0x001001c0 line: 11, main() in "d1.c"
B 11* main()
tmdbg> s
```

```

Stopped at addr: 0x00100219 line: 19, main() in "d1.c"
B 19* printf ("Sum (%d) = %d \n", i, sum(i));
tmdbg> s
Stopped at addr: 0x00100c00 line: 10, sum() in "d2.c"
B 10* int
tmdbg> l
 11  sum (int i)
 12  {
 13      int j, sum;
 14
 15      for (j = 0, sum = 0; j < i+1; j++)
B 16          sum += j;
 17
B 18      return (sum);
 19 }
 20
 21
tmdbg> s
Stopped at addr: 0x00100c60 line: 16, sum() in "d2.c"
B 16*     sum += j;
tmdbg> finish
Target is running. Type ctrl-c to stop the target
Stopped at addr: 0x00100280 line: 19, main() in "d1.c"
B 19*     printf ("Sum (%d) = %d \n", i, sum(i));

```

Stepping Out of Functions

Use the **finish** command to step out of functions.

tmdbg respects previously set breakpoints when stepping out of a function. That is, if the current function calls another which contains a previously set breakpoint, **tmdbg** will stop on that breakpoint. The same holds for breakpoints inside the function as well.

Using Breakpoints

This section describes how to set and remove breakpoints.

Setting Breakpoints

Setting breakpoints allows you to trace the execution of your programs. **tmdbg** provides the following three types of breakpoints:

- Software instruction breakpoints
- Hardware data breakpoints
- Hardware instruction breakpoints

Setting Software Instruction Breakpoints

Setting a software breakpoint involves applying a software patch to the target code and consequently incurs more overhead than hardware breakpoints (see *Setting Hardware Data Breakpoints*) but are much more flexible.

Instruction breakpoints specify the lines in the source code at which execution stops. To set a software breakpoint:

Command-line	GUI
Use the break (b or br)	Click the decision tree marker.

For example, to set a breakpoint at line 37 in x.c, enter the following:

```
tmdbg> b 37
```

tmdbg displays a message confirming the setting of the breakpoint:

```
New breakpoint # 1 (line: 37, main() in "x.c")
```

Important

You can only set breakpoints at lines marked with decision tree markers (B).

Setting a Breakpoint Inside a Function

To set a breakpoint inside a function (for example, **bar**) enter the following:

```
tmdbg> br bar
```

tmdbg displays a message confirming the setting of the breakpoint:

```
New breakpoint # 2 (line: 23, bar() in "x.c")
```

In this example, execution stops at line 23, before line 23 is executed.

Setting Conditional Breakpoints

To set a conditional breakpoint, simply use the standard ‘break’ syntax and extend it with ‘when <expression>’. For example, to reset the breakpoint at line 37, delete it, and then reset it as follows:

```
tmdbg> d 1
cleared bp, line 37 in "x.c" (addr: 0x00100140)
tmdbg> b 37 when y.x == 0
New breakpoint # 3 (line: 37, main() in "x.c")
```

The expression used as the condition is evaluated using the same context as the breakpoint location. For example, it is not possible to set a breakpoint at line 37 using the **static int y** declared in the file scope. The remaining uses of break are analogous.

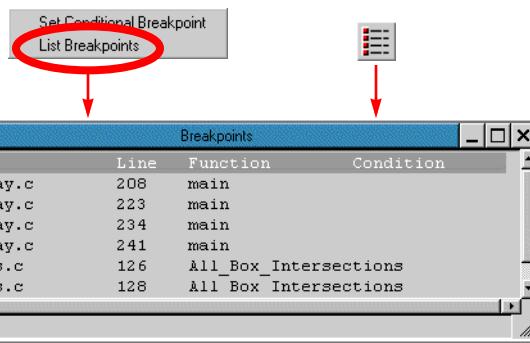
Note

It is necessary to remove an existing breakpoint to set a conditional one in the same place.

For more information about the different formats of the **break** command, use either the **help** command or refer to the manual page. In addition, refer to the section *tmdbg Breakpoint Anomalies* on page 31 for more information.

Listing Currently Active Breakpoints

To list currently active breakpoints, do the following:

Command-line	GUI																												
Use the status command as in the following example.	<p>Right-click a decision tree marker and choose List Breakpoints. (Alternatively, you can click the List Breakpoints button or choose List Breakpoints from the View menu.)</p>  <table border="1" data-bbox="495 489 1071 701"> <thead> <tr> <th>File</th> <th>Line</th> <th>Function</th> <th>Condition</th> </tr> </thead> <tbody> <tr> <td>povray.c</td> <td>208</td> <td>main</td> <td></td> </tr> <tr> <td>povray.c</td> <td>223</td> <td>main</td> <td></td> </tr> <tr> <td>povray.c</td> <td>234</td> <td>main</td> <td></td> </tr> <tr> <td>povray.c</td> <td>241</td> <td>main</td> <td></td> </tr> <tr> <td>boxes.c</td> <td>126</td> <td>All_Box_Intersections</td> <td></td> </tr> <tr> <td>boxes.c</td> <td>128</td> <td>All_Box_Intersections</td> <td></td> </tr> </tbody> </table>	File	Line	Function	Condition	povray.c	208	main		povray.c	223	main		povray.c	234	main		povray.c	241	main		boxes.c	126	All_Box_Intersections		boxes.c	128	All_Box_Intersections	
File	Line	Function	Condition																										
povray.c	208	main																											
povray.c	223	main																											
povray.c	234	main																											
povray.c	241	main																											
boxes.c	126	All_Box_Intersections																											
boxes.c	128	All_Box_Intersections																											

```
tmdbg> status
Stopped at addr: 0x001000cd line: 31, main() in "x.c"
B 31* int main(int argc, char *argv[])
List of breakpoints:
[1] line: 37 addr: 0x00100140, main() in "x.c" when y.x == 0
[2] line: 23 addr: 0x00100080, bar() in "x.c"
```

Setting Hardware Data Breakpoints

Hardware breakpoints are implemented by the TriMedia chip, so they require no software overhead. Data breakpoints allow you to trace changes to data. This is especially

important when you suspect that a memory location is being accessed or updated by an unknown entity. To set a data breakpoint, do the following:

Command-line	GUI
Use the watch command. This command places a data watchpoint on the address range you specify.	<p>Click the Set Data Breakpoint button or choose Hardware Data Breakpoint from the Debug menu.</p> <p>The screenshot shows the TMDBG interface. At the top, there's a menu bar with 'File', 'Edit', 'Run', 'Break', 'Registers', 'Stack', 'Watch', 'Breakpoints', 'Hardware', 'Help'. Below the menu bar is a toolbar with icons for Run, Stop, Step, Next, Finish, and Refresh target status. On the left, there's a stack viewer and a registers viewer. In the center, there's a 'Data Breakpoint (cpu #0)' dialog box. The dialog has fields for 'Address range (low)' (set to 100) and 'Address range (high)' (set to 200). It has radio buttons for 'Inside range' (selected) and 'Outside range'. There are checkboxes for 'Break on:' (Load, Store, Break on specific value), 'Not Value' (with a field set to 0xffffffff), and 'Pass count' (set to 1). At the bottom are 'Set' and 'Clear' buttons. To the left of the dialog, the 'Debug' menu is open, showing options like Run, Continue, Stop, Step, Next, Finish, and Refresh target status, with 'Hardware Data Breakpoint' highlighted. A red arrow points from the 'Hardware Data Breakpoint' menu item to a red circle with a question mark icon on the right. Another red arrow points from the 'Set' button in the dialog to the same red circle.</p>

For example, to set a data breakpoint for the variable **y** in **x.c**, do the following:

1. Use the **print** command to find **y**'s starting address.
2. Use the **whatis** command to get **y**'s size.
3. Use the **watch load inrange** command to set a data breakpoint.

This is illustrated in the following example:

```
tmdbg> print &y
evaluates to addr: 0x008fff44
tmdbg> whatis noot
union noot {
    signed int x;
    signed char y[4];
};
tmdbg> watch load inrange 0x008fff44 0x008fff48
Data breakpoint placed for range:0x008fff44 0x008fff48
```

Note

The **what is** command accepts expressions.

Following is another example using d.out (compiled from d1.c, d2.c, and d.h). After starting **tmdbg** and using the **step** command, the program is stopped at line 19.

```
tmdbg d.out
      TriMedia C debugger tmdbg
      v0.61 of SunOS (Apr 30 1997 12:10:32)

downloading program ...
Target started. Waiting for it to initialize ...
done.
Stopped at addr: 0x001001c0 line: 11, main() in "d1.c"
B 11* main()
tmdbg> s
Stopped at addr: 0x00100219 line: 19, main() in "d1.c"
B 19*     printf ("Sum ( %d ) = %d \n", i, sum(i));
```

Place a data watchpoint on the address range **0x00117b4c** to **0x00117ba8** and continue execution.

```
tmdbg> watch store inrange 0x00117b4c 0x00117ba8
Data breakpoint placed for range: 0x00117b4c - 0x00117ba8
tmdbg> c
Target is running. Type ctrl-c to stop the target
Sum ( 0 ) = 0
dbpc = 0
bogus( 0 ) = 2.111516
Stopped at addr: 0x00100b00 line: 40, main() in "d1.c"
B 40*   fool (john, i);
HW data break event in tree at addr: 0x00101040 line: 51, foo() in "d2.c"
B 51     p2->age = sum;
```

When a store operation is performed in the specified range, a data break event occurs and execution stops at the end of the decision tree in which the store occurred. The debugger displays the source line where execution will continue (in this example, line 40) and the source line corresponding to the tree in which the store operation occurred (in this example, line 51).

Setting More than One Data Breakpoint

You can only set one data breakpoint at a time, as opposed to software instruction breakpoints, where there is no limit on the number of instruction breakpoints.

```
tmdbg> watch store inrange 0x008fff4c 0x008ffff50
Please delete the existing data breakpoint using 'del hwbp'.
```

Watch Command Types

tmdbg provides four types of the **watch** command (each has four variations) that allows you to set data breakpoints for memory load and store operations inside or outside the

address range you specify. Figure 6 illustrates the difference among these types. For more information about the **watch** command, use the **help** command.

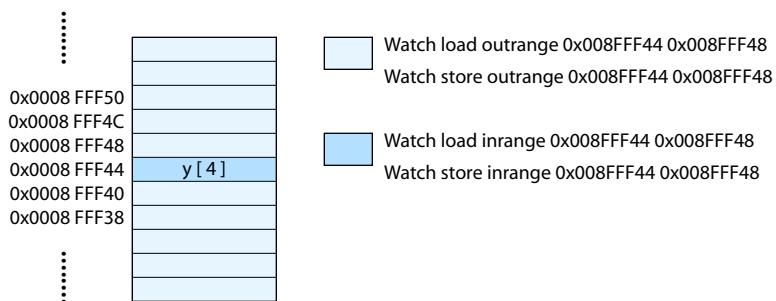


Figure 6 watch command types

Setting Hardware Instruction Breakpoints

You can set a hardware instruction breakpoint at any instruction address or address range. To set a data breakpoint, do the following:

Command-line	GUI
Use the hb command.	<p>Click the Instruction Breakpoint button or choose Hardware Instruction Breakpoint from the Debug menu.</p> <p>The 'Hardware Instruction Breakpoint (cpu #0)' dialog box is displayed. It contains fields for 'Address range (low)' and 'Address range (high)', a 'Break if:' radio button group ('Inside range' is selected), a 'Pass count' input field (set to 1), and 'Set' and 'Clear' buttons. A red arrow points from the 'Hardware Instruction Breakpoint' option in the menu to this dialog box. A red 'i' icon is also present in the dialog box area.</p>

Since line numbers and functions can be uniquely associated with an address, the **hwbreak** command supports these forms as well. For example, to set a hardware breakpoint at function **main()**:

```
tmdbg>hb main()
hardware instruction breakpoint placed for range: 0x00100080 - 0x00100080
```

The **status** command will report the existence of the hardware instruction breakpoint.

```
tmdbg> status
stopped at addr: 0x001000cd, line 31, main() in "x.c"
B 31* int main(int argc, char *argv[])
list of breakpoints:

HW Instruction breakpoint: addr range: 0x00100080 - 0x00100080 passcount: 1
x.c main()
```

Notice that the instruction breakpoint, like the data breakpoint, is defined by an address range. It is possible to specify a larger range than the single instruction in the previous example. However, there can only be a single hardware breakpoint (including data and instruction breakpoints) defined at any time. So, first you must delete the breakpoint.

Furthermore, the **passcount** field allows the breakpoint to be skipped *n* times before causing an interrupt.

```
tmdbg> del hwbp
hardware instruction breakpoint deleted
tmdbg> hb inrange 0x00100080 0x00100090 after 7
hardware instruction breakpoint placed for range: 0x800001c0 - 0x800001d0, pa
sscount: 7
```

And finally, the **inrange** and **outrange** parameters are analogous to the hardware data breakpoint

Removing Breakpoints

To remove an instruction breakpoint:

Command-line	GUI
Use the delete command (also d and del).	Click the red decision tree marker (represents an instruction breakpoint). It becomes green.

The following example shows you how to delete the first breakpoint:

```
tmdbg> del 1
cleared bp, line 37 in x.c (addr: 0x00100140)
```

To remove a data breakpoint:

Command-line	GUI
Use the del hwbp command.	Click the Clear button in the Data Breakpoint window. 

To remove all breakpoints (instruction and data):

Command-line	GUI
Use the del all command.	Click the Clear All Breakpoints button. 

tmdbg Breakpoint Anomalies

The TriMedia C compiler (**tmccom**) transforms the source code for a function into a set of decision trees. A decision tree may consist of a series of straight-line code followed by a function call. If a line containing a function call performs an assignment, the assignment may become part of the *next* decision tree. For example, consider the following program:

```
#include <iostream.h>

int foo ( int n ){
    cout << "foo has been invoked with: " << n << endl;
    return ++n;
}

int a, b;

int main (){
    a = foo(1);
    b = foo(foo(a));
}
```

When you compile and load the program into **tmdbg**, the listing for **main** appears as follows:

```
tmdbg> list
      15  {
B   16      a = foo(1);
B   17      b = foo(foo(a));
      18 }
```

Although the decision tree marker (**B**) marks the beginning of a decision tree, the decision tree boundary is *not* necessarily at the *beginning* of the line.

For example, place a breakpoint in line 16 and enter a **continue (c)** command:

```
tmdbg> b 16
new breakpoint # 1 (line: 16, main() in "g.cc")
tmdbg> c
target is running, type ctrl-c to stop the target
foo has been invoked with: 1
stopped at addr: 0x00100380, line 16, main() in "g.cc"
B 16*
```

You'll notice that the function **foo** has already been executed. However, the assignment (**a = foo(1);**) has not been performed yet:

```
tmdbg> p a
a = 0
```

This means that the assignment is part of the next decision tree and the decision tree marker in line 16 corresponds to the return of function **foo**.

The same principle holds for embedded function calls. There is a decision tree boundary after each series of straight-line code, followed by a function call. So, in line 17, there are actually *two* decision tree boundaries.

If you set a breakpoint in line 17 and continue execution, **tmdbg** stops the program execution after the inner call to **foo**, but before executing the outer call and the assignment to **b**.

```
tmdbg> b 17
new breakpoint # 2 (line: 17, main() in "g.cc")
tmdbg> c
target is running, type ctrl-c to stop the target
foo has been invoked with: 2
stopped at addr: 0x001003d2, line 17 main() in "g.cc"
B 17*
```

If you issue a **next** command, **tmdbg** steps over the function call and *not* the next line. You actually end up on the same line.

```
tmdbg> n
target is running, type ctrl-c to stop the target
foo has been invoked with: 3
stopped at addr: 0x00100440, line 17, main() in "g.cc"
B 17*
tmdbg> p b
b = 0
```

The variable assignment has still not yet been made. If you process the next decision tree (the last one in **main**), the variable **b** gets assigned. You must issue a **step** command (instead of **next**) here or the program will complete and no variables will be active.

```
tmdbg> s
stopped at addr: 0x0011f402, line 21, exit() in "exit.c"
source file has no debug info
```

The **step** command lands you in the ANSI C library function **exit()**, which is the implicit return point for the **main()** function. However, you can still access variable **b**.

```
tmdbg> p b
b = 4
```

Software Workaround

If you require the behavior of a traditional debugger, the best thing to do is to set a breakpoint on the *first* function called, so that once the breakpoint has been reached, you can go up the call stack one level and perform whatever inquiries need to be made.

For example, to stop before any work on line 16 is finished; you would use the following commands:

```
break foo
continue
up
```

Then, perform any inquiries:

```
down
continue
```

And resume as normal.

Examining and Changing Data, Memory, and Registers

This section briefly describes the mechanisms and commands that **tmdbg** provides for examining and changing data in the memory and registers. For more detailed information, use the **help** command.

Symbolic Access

Symbolic access allows you to do the following:

- Display the value of expressions and registers using the **print** command:

```
tmdbg> print x
x = struct {
    x = 305419896
    y = 2
}
tmdbg> print f
f = (nil)
```

- Display the address of a symbol using **print** command and preceding the variable name with the address operator (&):

```
tmdbg> p &y
evaluates to addr: 0x008fff44
```

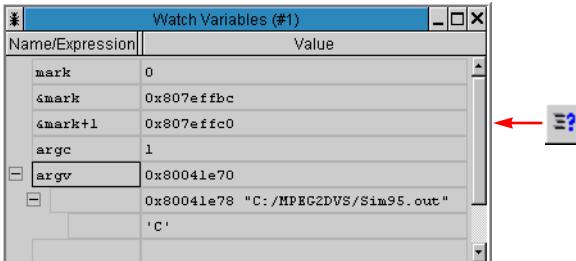
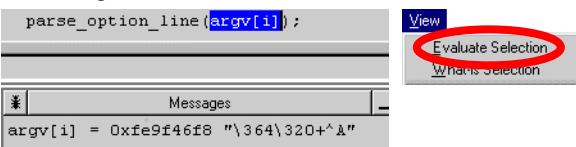
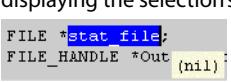
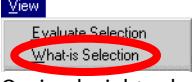
When examining data structures, take into account that the scope of these variables affects the outcome.

For example, **y** (the local variable in the function **main** where execution stopped) is displayed when you use the command **print y**, as long as you are inside the function **main**, even if there is a global variable **y**. Local variables always take precedence.

Expression Evaluation

tmdbg supports all standard C operators for evaluating expressions using the C promotion rules and supports type casting. In addition, register names, when used in expressions, represent 4-byte integer values.

Expression can be used with the following **tmdbg** commands:

Command-line	GUI
assign	<p>Choose Watch Variables from the View menu (or click the Watch Variables icon), enter the variable name in the Name/Expression field, and enter the value of the expression in the Value field.</p> 
print	<p>Select the expression you want to evaluate and choose Evaluate Selection from the View menu. The value of the selection appears in the Messages window.</p>  <p>Or, simply select the expression to evaluate and a balloon opens up displaying the selection's value.</p> 
whatis	<p>Select the expression and choose Whatis Selection from the View menu. The definition of the selection appears in the Messages window.</p>  <p>Or, simply right-select the expression and a balloon opens up displaying the selection's definition.</p>
examine	<p>Refer to the section <i>Nonsymbolic Access</i> on page 35.</p>

Assignment

Assignment allows you to assign (using the **assign** command) the value of an expression to a variable or a register, as shown in the following examples:

```
tmdbg> assign func =(int)main
tmdbg> print func
    func = 1049731
tmdbg> assign $3 = $14 + 2
tmdbg> print $3
    $3 = 2

tmdbg> a x.x = 21

tmdbg> a $SPC = 100000
about to alter a system register, go ahead? [y/n] default = n:
system register not altered
```

Note

In the third example, tmdbg displays an alert message before changing the value of a special register because modifying special registers is not required in general. Doing so without knowing the register usage conventions and internals of the run-time system may cause the system to crash.

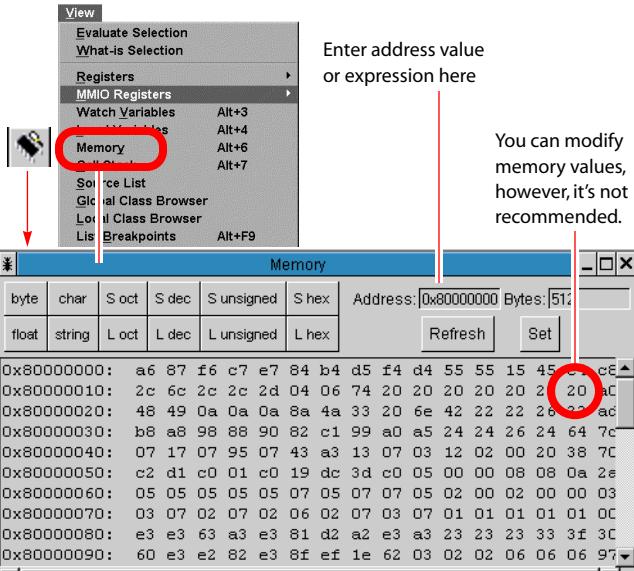
The **assign** command is very useful in debugging because it allows you to change the values of variables and registers while running the program with **tmdbg**. For example, if you suspect that a certain variable is responsible for crashing the program you're debugging, you can use the **assign** command to change the value of the suspected variable and test your theory after resuming program execution.

In addition, the **assign** command allows you to take advantage of free registers (not used by the compiler and debugger) for temporary storing of expressions.

Nonsymbolic Access

Non-symbolic access allows you to display the values of the specified memory locations or registers.

To display the values memory locations:

Command-line	GUI
Use the examine (x) command, as shown in the following example.	<p>To examine memory locations, use the Memory button or choose Memory from the View menu.</p>  <p>Enter address value or expression here</p> <p>You can modify memory values, however, it's not recommended.</p>

```
tmdbg> x 0x008fff44 X 16
0x008fff44 : 0x00000000 0x008fff80 0x00116240 0x00000000
0x008fff54 : 0x00000000 0x0010f100 0x0011bf20 0x00100040
0x008fff64 : 0x00000000 0x0010f0c0 0x00000000 0x00000000
0x008fff74 : 0x00100040 0x00000000 0x00000000 0x00000001
```

Note

The **examine** command accepts expressions.

To display the values of registers, use the **regs** command, as shown in the following example:

```
tmdbg> regs
$0 = 0
$2 = 0x00106b80 [1.507932e-39]
$4 = 0x008efe80 [1.313194e-38]
$6 = 0x008efcc8 [1.313132e-38]
$8 = 0
$10 = 0x00164904 [2.046574e-39]
$12 = 0x00100140 [1.469816e-39]
$14 = 0
$16 = 0
$18 = 0
$20 = 0
$22 = 0
$24 = 0
$26 = 0
$28 = 0
$30 = 0
$32 = 0
$34 = 0xffffffffd [-NaN]
$36 = 0
$38 = 0
$40 = 0
$42 = 0x00118574 [1.609077e-39]
$44 = 0x00000001 [1.401298e-45]
$46 = 0x0000000a [1.401298e-44]
$48 = 0x00000003 [4.203895e-45]
$50 = 0x00000003 [4.203895e-45]
$52 = 0
$54 = 0x00118564 [1.609055e-39]
$56 = 0
$58 = 0
$60 = 0x00115263 [1.590758e-39]
$62 = 0x001682e8 [2.067341e-39]

$RP ($2) = 0x00106b80   $SPC  = 0x0010054c   $CC_COUNT_LO = 0x00000000
$FP ($3) = 0x008effbc   $DPC  = 0x0010054c   $CC_COUNT_HI = 0x00000000
$SP ($4) = 0x008efe80   $PCSW = 0x000000c00
arg1($5) = 0x0000002d [6.305843e-44]
arg2($6) = 0x008efcc8 [1.313132e-38]
arg3($7) = 0
arg4($8) = 0
```

The **regs** command displays the registers of the current (stack) context. This includes the global and system registers.

To display the value of high registers (64 to 129), choose High Registers from the View menu. The High Registers window appears.

	Registers 64 - 128						-	□	X
64	0x00000000	65	0x00000000	66	0x00000000	67	0x00000000		
68	0x00000000	69	0x00000000	70	0x00000000	71	0x00000000		
72	0x00000000	73	0x00000000	74	0x00000000	75	0x00000000		
76	0x00000000	77	0x00000000	78	0x00000000	79	0x00000000		
80	0x00000000	81	0x00000000	82	0x00000000	83	0x00000000		
84	0x00000000	85	0x00000000	86	0x00000000	87	0x00000000		
88	0x00000000	89	0x00000000	90	0x00000000	91	0x00000000		
92	0x00000000	93	0x00000000	94	0x00000000	95	0x00000000		
96	0x00000000	97	0x00000000	98	0x00000000	99	0x00000000		
100	0x00000000	101	0x00000000	102	0x00000000	103	0x00000000		
104	0x00000000	105	0x00000000	106	0x00000000	107	0x00000000		
108	0x00000000	109	0x00000000	110	0x00000000	111	0x00000000		
112	0x00000000	113	0x00000000	114	0x00000000	115	0x00000000		
116	0x00000000	117	0x00000000	118	0x00000000	119	0x00000000		
120	0x00000000	121	0x00000000	122	0x00000000	123	0x00000000		
124	0x00000000	125	0x00000000	126	0x00000000	127	0x00000000		

Set **Undo** **Refresh**

WARNING

Although the GUI version of tmdbg enables you to change the values, you are highly discouraged from doing so, because unspecified behavior may result.

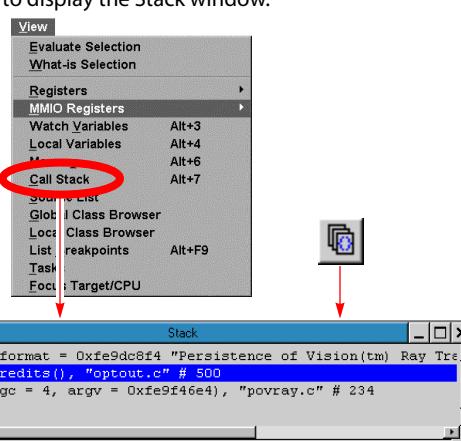
To display the value of a single register, use the **print** command, as shown in the following example:

```
print $1
```

Tracing the Call Stack

tmdbg's call stack tracks the function-calling sequence in ascending order based on the stack pointer. Each entry in the stack is preceded with a number indicating its stack order.

To display the stack entries:

Command-Line	GUI
Use the trace (also t) command, as shown in the following example.	<p>Choose Call Stack from the View menu, or click the Stack Trace button to display the Stack window.</p>  <p>Clicking a stack entry displays its corresponding source file in a new source code window.</p>

```
tmdbg> t
=>[1] bar(x = 305419896), line 23 in "x.c"
[2] main(argc = 1, argv = 0x11a2e4), line 37 in "x.c"
```

In this case, the stack pointer (`=>`) points to **bar** where execution has stopped.

Note

The function that is on top of the stack (deepest callee function) is always indicated by [1].

Moving Up and Down the Call Stack

To move up and down the Call Stack:

Command-line	GUI
Use the up and down commands, as shown in the following example.	Click the desired entry.

```
tmdbg> up
Current function: main() in file "x.c"
B 37 y.x = foo(f, bar(x.x));
[1] bar(x = 305419896), line 23 in "x.c"
=>[2] main(argc = 1, argv = 0x11a2e4), line 37 in "x.c"
tmdbg> down
Current function: bar() in file "x.c"
B 23* static int
=>[1] bar(x = 305419896), line 23 in "x.c"
[2] main(argc = 1, argv = 0x11a2e4), line 37 in "x.c"
```

Going up means that the stack pointer moves from entry n to entry n+1, and vice versa.

Disassembling Instructions

The **Disassemble** command disassembles machine instructions. In the following example, only a portion of the disassembled code is shown:

```
tmdbg> help dis
DIS - Disassemble machine instructions.

dis                                Disassemble 10 instructions from current pc
dis for <n>                      Disassemble <n> instructions from current pc

dis <line>                         Disassemble 10 instructions from <line>
dis <line> for <n>                  Disassemble <n> instructions from <line>

dis <line> "file"                   Disassemble 10 instructions from <line> of "file"
dis <line> "file" for <n>           Disassemble <n> instructions from <line> of "file"

dis addr <adr>                   Disassemble 10 instructions from address <adr>
dis addr <adr> for <n>             Disassemble <n> instructions from address <adr>
dis addr <adr1> to <adr2>         Disassemble from address <adr1> to <adr2>

dis <func>                          Disassemble 10 instructions, from <func>
dis <func> for <n>                 Disassemble <n> instructions, from <func>

tmdbg> s
Stopped at addr: 0x00106516 line: 31, rand() in "rand.c"
source file has no debug info.
tmdbg> fin
Target is running. Type ctrl-c to stop the target
Stopped at addr: 0x00100386 line: 22, main() in "d1.c"
B 22*      f1 = rand();  fs[1] = f1;
tmdbg> dis
(* cycle 0 *)
    IF r1  iimm(1024) -> r2 ,
    IF r1  nop ,
    IF r1  nop ,
    IF r1  h_st32d(-128) r5 r3 ,
    IF r1  nop ;

(* cycle 1 *)
    IF r1  nop ,
    IF r1  nop ,
    IF r1  nop ,
    IF r1  ld32d(-128) r3 -> r127 ,
    IF r1  nop ;
```

Using tmdbg with tmsim on Windows 95

Because **tmdbg** uses sockets for interprocess communications when used with the simulator **tmsim**, the host operating system must support sockets.

On Windows 95, you must install TCP/IP networking (even if the host is not attached to a network) and make sure the machine has an assigned IP address. Do the following:

1. Choose Network from the Control Panels menu.
2. If TCP/IP is not installed, click the Add button and follow installation instructions.
3. Choose IP Address from the Control Panels >> Network >> TCP/IP >> Properties menu.
4. Make sure the machine has an assigned IP address. If no IP address is assigned, enter any IP address.

Note

tmdbg reports a **gethostbyname** failure if it cannot find an IP address for the host machine.

Chapter 17

Debugging C++ Code with tmdbg

Topic	Page
Introduction	44
Accessing C++ Definitions	44
Accessing C++ Definitions	44
Accessing Class Members	45
Accessing C++ Variable Declarations	47
Setting Breakpoints	47
Accessing Virtual Functions	48
Debugging Templates	49
Debugging Templates	49
Unsupported Features	50
Non-ANSI Compliant Names	52
Notes	53

Introduction

This version of **tmdbg** contains many improvements over version 1.1. Mainly, changes in the compiler give the debugger more information about C++ types. Much C++ information (such as member access control, which was missing in the 1.1 debugger) is now supported. As a result, this version of the C++ debugger is easier to use. Apart from a few restrictions that are discussed in the following pages, most functions that debug C also debug C++.

Accessing C++ Definitions

If the class appeared in the source code as

```
class derived : public base {
    float data;
    static int instance_count;
public:
    derived (float f = 0.0);
    ~derived ();
    float foo (float f);
    void not_used() { ; }
    derived &operator + (const derived &d);
    derived &operator + (int n);
    operator float ();
    operator farray ();
};
```

use the **what is** command to display the class definition:

```
tmdbg> whatis derived
typedef const class derived derived;
public:
    class derived& operator +(const class derived& d);
    class derived& operator +(signed int n);
    float operator float(void);
    class farray operator farray(void);
private:
    static signed int instance_count;
    float data;
public:
    derived(float f);
    ~derived(signed int);
    float foo(float f);
    unknown <int assumed> not_used(void);
    derived(const derived&);
};
```

Class members might not appear in their original order. The compiler also generates internal types and functions that were not declared in the C++ code. For example, the compiler generates the following type definition even though this type was not in the source.

```
typedef const class derived derived;
```

It will also generate, for every class, a default copy constructor (whether the original class contained one or not):

```
derived(const derived&);
```

By default, when using the print function, base class information will not be printed, but it can be printed if the verbose option is set.

```
tmdbg> wi d1
class derived d1;
tmdbg> p d1
d1 = class {
    instance_count = 0
    data           = 0.0
}
tmdbg> set verbose on
tmdbg> p d1
d1 = struct {
    instance_count = 0
    base = class {
        data = 0
    }
    data = 0.0
}
```

Note:

The compiler does not pass any information to the debugger regarding friends, and access adjustments. This information will not be printed.

Accessing Class Members

Non-static data members and function members can be accessed by their qualified names.

```
tmdbg> whatis derived::instance_count
static signed int instance_count;

tmdbg> whatis derived::derived
derived::derived(float f);
addr: 0x00101a83
derived::derived(const derived&);
addr: failed to get address, function may have been removed by compiler
tmdbg> whatis derived::~derived
void derived::~derived(signed int);
addr: 0x00101b83

tmdbg> whatis derived::base::data
signed int base::data;
```

In this example, the debugger sees two overloaded symbols for the constructor of **derived**. One of them is the constructor specified in the C++ code. The other has been added by the compiler (the copy constructor), but **tmdbg** is unable to retrieve its address. It has been removed by the compiler because this constructor was not used.

Similarly, to display static data, you can use qualified names to access memory contents.

```
tmdbg> p derived::instance_count
instance_count = 0
```

Note

This is not possible with non-static data. Non-static data requires that an instance of the class be specified.

To make assignments to static data, you can use the qualified name.

```
tmdbg> a derived::instance_count = 600
```

Note

Partially qualified names relative to a scope are not currently being supported in **tmdbg**. For example, if you use the variable **instance_count** inside **derived**'s function member **foo**, the compiler resolves the name to **derived::instance_count**. However, **tmdbg** does not handle this. For example, if you set a breakpoint in **foo**, stop there, and try to access **instance_count**, you get an error message.

```
Stopped at addr: 0x00101a83, line 36, derived () in "cppsample.cc"
B 36* derived (float f = 0.0) {instance_count++;
tmdbg> p instance_count
unknown symbol: instance_count
tmdbg> p derived::instance_count
instance_count = 0
```

To get around this problem, always use fully qualified names.

Base classes can be seen by specifying their names in any of the following ways: **base**, **derived::base**, or **::base**. However, **::base** and **base** reduce to the same scope, because of the lack of context information used in the debugger, as previously specified.

```
tmdbg> whatis ::base
class base {
    signed int data;
public:
    base(void);
    base(signed int d);
    void set_data(signed int d);
    void foo(void);
    base(const base &);
};
typedef const class base base;
tmdbg> wi derived::base
class base {
    signed int data;
public:
    base(void);
    base(signed int d)
    void set_data(signed int d);
    void foo(void);
    base(const base&);
};
```

Accessing C++ Variable Declarations

Assignments to class members are also supported. Static members can either be specified by class type, or through an instance of the class. For example, if **d1** is an instance of class derived:

```
tmdbg> p derived::instance_count
instance_count = 1
tmdbg> a derived::instance_count = 600
tmdbg> p d1.instance_count
evaluates to: 600
tmdbg> p d1.instance_count + derived::instance_count * 2
1800
```

Volatile members must be specified through an instance.

```
tmdbg> a d1.data = 15.5
tmdbg> p d1.data
evaluates to: 15.5
```

Variable declarations in other scopes are generally visible in C++ programs. The debugger also reports all visible variables starting with the current scope and proceeding to the outermost. For example, from the context of the function **main**, several symbols have the name **x**:

```
tmdbg> wi x
signed int main`x;
signed int x;
tmdbg> wi ::x
signed int x;
tmdbg> p x
main`x = 0
x = 15
```

As illustrated, you can use the global scope modifier ‘**::**’.

Setting Breakpoints

To place a breakpoint in a C++ function, you must specify the complete qualified function name:

```
tmdbg> b derived::foo
new breakpoint # 2 (line: 42, derived::foo() in "cppsample.cc")

tmdbg> b derived::base::foo
new breakpoint # 3 (line: 29, base::foo() in "cppsample.cc")
```

You can also place breakpoints in C++ operators and type-conversion routines. For example, if you define the class **farray** this way:

```
class farray{
    float data[5];

public:
```

```
void set_data(float f) { data[0] = data[1] = data[2] = data[3]
                           = data[4] = f; }
```

and redefine the class **derived** this way:

```
class derived : public base {
    float data;
    static int instance_count;
public:
    derived (float f = 0.0) { instance_count++;
                             data = f;
                             base::set_data(0.0); }
    ~derived ()           { instance_count--; }
    float foo (float f)   { cout << "derived, foo: " << data << endl;
                           return data + f; }

    derived &operator + (const derived &d);
    derived &operator + (int n);

    operator float ();
    operator farray ();
};
```

then you can set breakpoints in the type-conversion routines as follows:

```
tmdbg> b derived::operator float
new breakpoint # 4 (line: 79, derived::operator float() in "cppsample.cc")

tmdbg> b derived::operator farray
new breakpoint # 5 (line: 72, derived::operator farray() in "cppsample.cc")
```

If a function is overloaded, you are prompted to choose one or more functions.

```
tmdbg> b derived::operator +
Please choose from options 0 to 3
[0] /* cancel break command */
[1] /* set breakpoint in all */
[2] class derived *operator +(const derived &d)
[3] class derived *operator +(signed int n)
> 1
new breakpoint # 6 (line: 57, derived::operator +() in "cppsample.cc")
new breakpoint # 7 (line: 65, derived::operator +() in "cppsample.cc")
```

If there are many functions listed, you can enter multiple options at the '>' prompt.

Note

The keyword **operator** is always necessary for specifying C++ operators and type-conversion routines.

Accessing Virtual Functions

In the C++ examples in the previous sections, any of the functions could have been virtual. Virtual functions are only an issue when functions are identified via their instance declarations, which is currently not supported.

Debugging Templates

tmdbg now supports class and function template debugging. It supports parsing of class template arguments on the command line as well as accessing templates without arguments. To specify a class template without arguments, simply give the name without any arguments and **tmdbg** will provide each instance.

For example, given the declaration for **vector**,

```
template<class T> class vector {
    T* v;
    int sz;
public:
    vector (int n) { v = (T *)malloc(sizeof(T) * n); sz = n; }
    T& operator[](int i) { return v[i]; }
    T& elem(int i) { return v[i]; }
};

typedef vector<complex> cvec;

vector<int>      v1(20);
vector<complex>  v2(30);
cvec              v3(40);
vector<cvec>     m(10); /* matrix of complex's */
```

you can access the type definition of each template instance as follows:

```
tmdbg> wi vector
struct vector<signed int> {
    signed int *v;
    signed int sz;
    struct vector<signed int> *vector(signed int n);
    signed int *operator [](signed int i);
};
struct vector<complex> {
    struct complex *v;
    signed int sz;
    struct vector<complex> *vector(signed int n);
    struct complex *operator [](signed int i);
    struct complex *elem(signed int i);
};
struct vector<vector<complex>> {
    struct vector<complex> *v;
    signed int sz;
    struct vector<vector<complex>> *vector(signed int n);
    struct vector<complex> *operator [](signed int i);
};
```

Likewise, to set a breakpoint in one of the template members, specify the template name with no arguments:

```
tmdbg> b vector::operator []
Please choose from options 0 to 4
[0] /* cancel break command */
[1] /* set breakpoint in all */
[2] signed int *vector<signed int>::operator [](signed int i)
[3] struct complex *vector<complex>::operator [](signed int i)
[4] struct vector<complex>
```

```
*vector<vector<complex>>::operator [](signed int i)
> 2
new breakpoint # 2 (line: 11, vector::operator []() in "cpp023.cc")
Access to class template data is also straightforward:
```

```
tmdbg> p v2.v[3].re
      evaluates to: 7.0
tmdbg> p m.v[5].v[3].re
      evaluates to: 7.0
```

Access to static data through the instance is analogous, but you may not access static data using the class template type:

```
tmdbg> p vector::max
error in parsing: syntax error at token: .
tmdbg> p vector<int>::max
error in parsing: syntax error at token: <
```

Function templates are handled similarly. If there is a function template **sort**,

```
template<class T> void
sort(vector<T> v)
{
    ...
}
```

then specify the function name as normal:

```
tmdbg> wi sort
void sort(class vector<signed int> v);
addr: 0x00000000
void sort(class vector<float> v);
addr: 0x00000000
tmdbg> b sort
Please choose from options 0 to 3
[0] /* cancel break command */
[1] /* set breakpoint in all */
[2] void sort(class vector<signed int> v)
[3] void sort(class vector<float> v)
> 2
```

Unsupported Features

The current version of **tmdbg** has the following limitations:

- **tmdbg** cannot access C or C++ include files in general because the compiler does not generate information about included files. What makes C++ include files even more difficult to access is the possibility that they may contain executable code. The compiler does not generate any debugging information about included files at present.
- **tmdbg** does not allow partially qualified names. All names must be fully qualified (as discussed previously).
- Because of a bug in the TriMedia compiler front end, **tmdbg** has trouble understanding class template names with the characters '7' and '_'. When a class template identifier contains one of these characters, **tmdbg** might fail to identify the correct

arguments for the template. This might not happen for every name with a '7' or a '_', but will definitely happen for some names such as `vec7junk_<int>`.

Moreover, if a struct or class name in a C++ source file has the characters '7' and '_', **tmdbg** may incorrectly determine the name to be a class template. For example, the name "`xxx7i_`" will be reported as "`struct xxx<signed int>`".

If you are having these sorts of problems, the temporary work-around is to remove the characters '7' and '_' from the class template or struct name.

- Currently, there is no support for function calls within expressions (either C or C++ functions), so functions might not be accessed through their instances.
- Currently, **tmdbg** does not support function calls from within the debugger. Since many C++ type casts involve a function call, it does not support C++ type casts either:
 - No casts involving classes (function calls not implemented in expressions)
 - C++ style casts (`static_cast`, `dynamic_cast`, and so on)
- C++ multitasking mechanisms are not supported.
- In the case where a class receives, by inheritance, two (or more) instances of the same class, **tmdbg** cannot distinguish between them by accessing the members directly. However, accessing the most derived class as a whole will show the different instances of the base classes.
- **tmdbg** does not fully understand unnamed types:

```
union { int x; char y; } foo;
```

A bug in the compiler makes the type `foo` to be named `__Cn` where *n* is the type number in the file.

`__Cn` is an unnamed class (or struct, or union),

`__Nn` is an unnamed namespace,

`__En` is an unnamed enum,

`__Vn` is an unnamed member variable.

```
tmdbg> wi foo
union foo;
tmdbg> wi __C1
class __C1 {
public:
    signed int x;
    signed char y;
};
```

This code shows the temporary work-around.

- The compiler does not support debugging information for anonymous unions.

- The compiler generates namespaces as classes with all members static. Since **tmdbg** has no knowledge of the **using** keyword, types and functions that are usually visible from the global scope are not visible inside the debugger:

```
tmdbg> wi size_t  
unknown symbol: size_t  
tmdbg> wi std::size_t  
typedef unsigned int std::size_t;
```

Most types and functions that are defined in the standard headers are within the **std** namespace.

Non-ANSI Compliant Names

The TriMedia Compiler and the TriMedia Debugger use the string **_0** (two underbars and a zero) as a special prefix to denote mangled C++ names. Mangled C++ names are valid C identifiers, which contain valid C++ information necessary for debugging. Consequently, the debugger attempts to “demangle” anything beginning with **_0**.

Names from a source program starting with **_0** will most likely *not* be in a valid mangled format (for example, **_0_fooled_you**). Names such as this will cause the debugger to emit an internal warning, and continue, but a name such as **_0dDfooBx** will cause the debugger to mistakenly generate a class named **foo** containing a static member **x**.

We highly recommend that you follow the ANSI C standard, or at least *not* prefix any identifiers with **_0**.

Notes

The following information concerns some idiosyncrasies of the **tmdbg** debugger:

- If a function within a class is never called, the compiler does not generate information for it. Hence, the debugger is unaware it ever existed. When you attempt to access unused functions, you get an “unknown symbol” error message.
- Some type names are converted to a *base* name by the compiler. For example, type-defs are converted to their base types.

```
tmdbg> wi atype
typedef signed int atype;
tmdbg> break Dclass::operator atype
new breakpoint #23 (line: 144, Dclass::operator signed int()
in "cpp003.cc")
```

- Types defined within classes are denoted with their fully qualified names. In the following example, the class **inner** defined within **outer** has the name **outer::inner** and this name must be used to specify the type.

```
tmdbg> wi outer
struct outer {
    struct inner operator outer::inner();
    struct inner {
        float data;
        struct inner *inner (floatf);
    };
}
tmdbg> wi outer::operator outer::inner
struct inner operator outer::outer::inner();
addr: 0x00102400
```


Chapter 18

Debugging TriMedia Applications Using JTAG

Topic	Page
Introduction	56
Stand-Alone Debugging Modes	57
System Requirements	59
Setting Up the System for Stand-Alone Debugging	61
Testing the JTAG Connection	62
Compiling a Program for Stand-Alone Debugging	64
Debugging Stand-Alone TriMedia Applications	65

Introduction

The TriMedia debugger (**tmdbg**) may be used to debug software on TriMedia-based stand-alone systems through a JTAG board (which could be a Corelis board or the new TriMedia JTAG turbo board as described in *System Requirements* on page 59) that connects to a TriMedia board via its JTAG port. The current release of the TCS toolset includes support for stand-alone debugging via JTAG. Detailed specifications of the JTAG block can be found in Chapter 17 of the *TM-1000 Data Book*.

The JTAG access port on a TriMedia processor provides access to three special MMIO registers that are used for communication between the debug monitor running on the TriMedia chip and **tmdbg** running on a PC host. Two of the registers are used as one-word input/output buffers and the third as a handshake register.

The commands and functionality of the source-level debugger **tmdbg** remain the same whether it is used with the machine-level simulator **tmsim**, the actual TriMedia chip in a PC host, or in a stand-alone system.

Note

JTAG debugging is not supported when the TriMedia board is in a JTAG chain.

Stand-Alone Debugging Modes

This chapter describes how to debug TriMedia stand-alone applications in the following two modes:

- No-Host
- Host-Assisted

No-Host Mode

In this mode, TriMedia runs in a stand-alone setup. For example, the TriMedia IREF board may be plugged into a passive PCI backplane that provides power. The TriMedia board is connected to a JTAG board in the PC through a JTAG connector cable. The JTAG board which could be a Corelis board or the new TriMedia JTAG turbo board as described in *System Requirements* on page 59, allows **tmdbg** to debug stand-alone applications running on the TriMedia board connected to it.

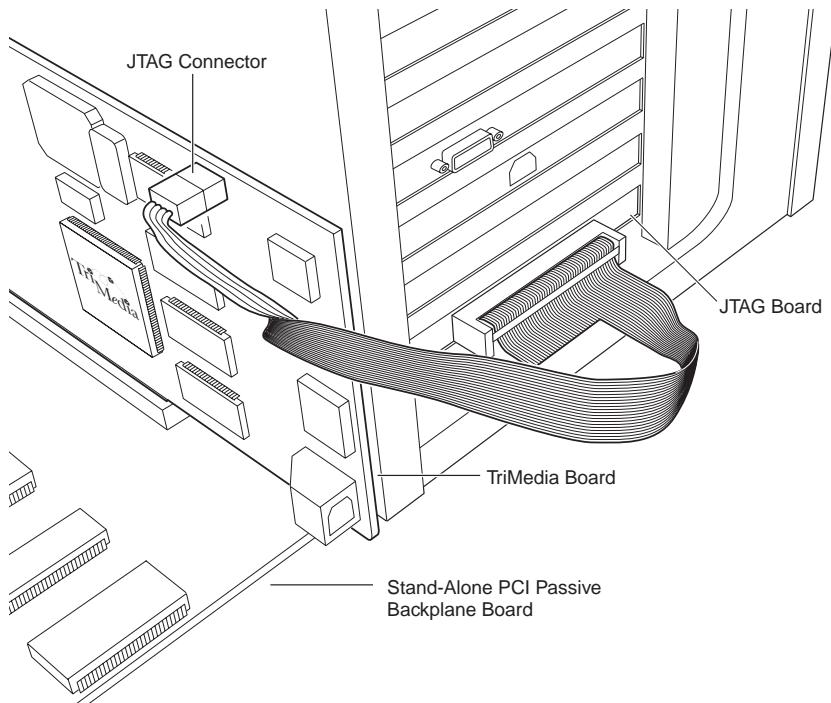


Figure 7 No-Host Mode

Host-Assisted

In this mode, the TriMedia board runs in a standard PCI slot in a PC and is connected (via a JTAG connector cable) to a JTAG board that can be in the same PC or a different PC.

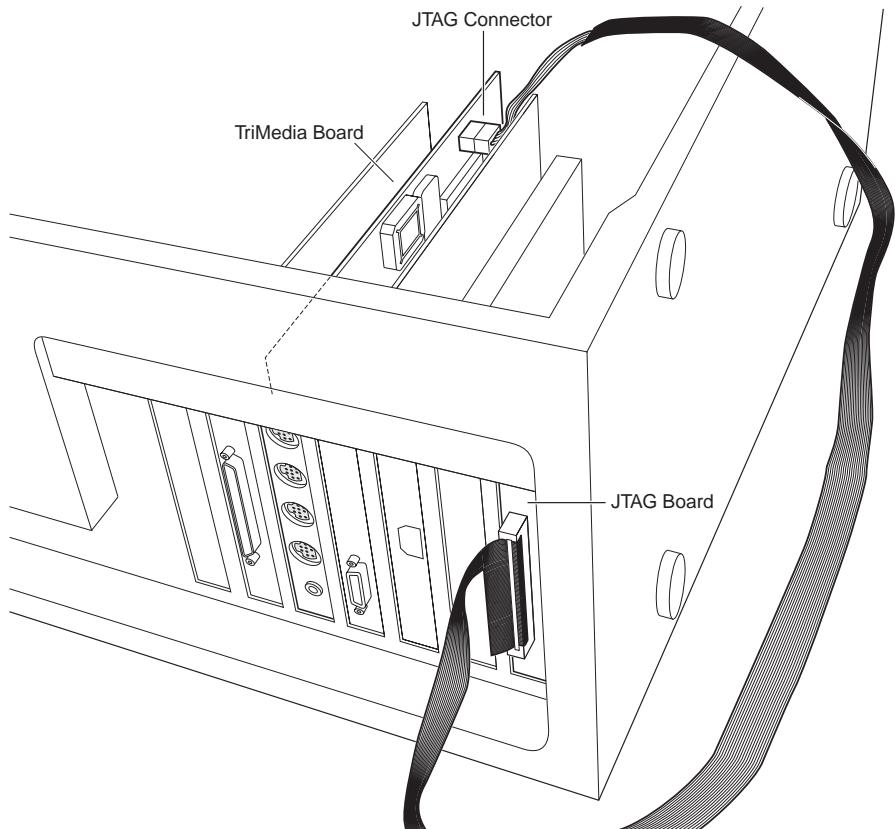


Figure 8 Host-Assisted mode

System Requirements

This section describes the hardware and software requirements for using **tmdbg** to debug stand-alone systems.

Hardware

The following hardware components are necessary to run **tmdbg**:

- A Pentium PC with a spare ISA slot and a free I/O address at 0x100, 0x140, 0x240 or 0x300. I/O address 0x140 is usually free and is the default address used by **tmdbg** to talk to the JTAG controller card.
- A Corelis PC-1149.1/100F JTAG controller card

The Corelis PC-1149.1/100F is designed to control the operation of an IEEE Standard 1149.1 (JTAG) scan test path by generating the proper signals under software control to interface with the target devices.

- *Instead* of the Corelis PC-1149.1/100F board, you can use the TriMedia JTAG turbo board designed specifically to work with **tmdbg**.

This board (which comes with a JTAG cable) has a single JTAG TAP and its download speed is about 15 times faster than that of the Corelis board, even though the TriMedia board also runs at a nominal clock speed of 15 Mhz.

For more information, please contact Associated Technologies, Inc. at the following address:

Associated Technologies, Inc.
Santa Clara, CA 95054
Tel: +1-408-727-3904
Email: astech@worldnet.att.net

- A TriMedia IREF board such as version 2.2 which includes a JTAG TAP and ground pins (Figure 10).
- An 8-pin flat ribbon cable with a 2×4 connector header at one end and a 40-pin female flat-cable connector at the other end (the TriMedia turbo board comes with such a cable).

If you don't have this type of cable, you can make your own from the following parts:

- A 40-pin female flat-cable connector (40-pin IDC, 3M part number 3432-5203 or equivalent) to attach to the JTAG board
 - A standard 40-pin flat ribbon cable
- This cable *should not* be too long. A 12-inch long cable is the recommended maximum length. In addition, you only need to use 8 pins (pins 1 to 8) of the cable.

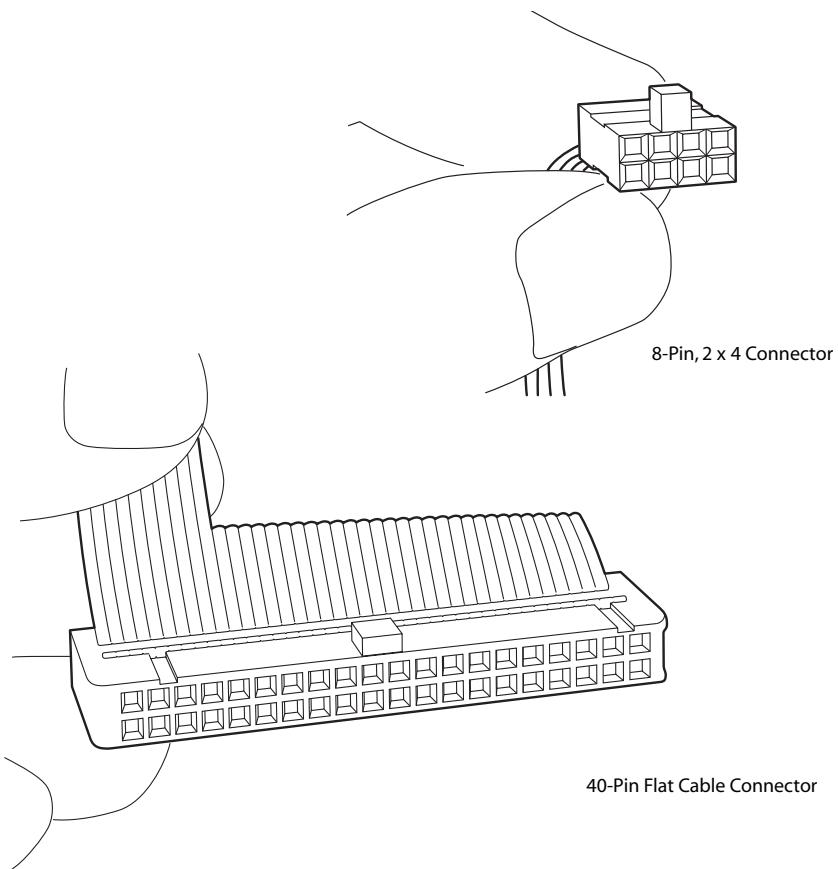


Figure 9 Connector Types

- A 2x4 Connector Header (SAMTEC part number TSW-104-07-L-D) to attach to the TriMedia IREF Board 0.025" square pins, 0.1" centers.

Software

The following software components are necessary to run **tmdbg**:

- Windows 95TM or Windows NT 4.0
- L1 boot program found in *TCS_INSTALL/examples/autoboot/jtag/l1.eeprom*

This directory also contains a README file, the L1 JTAG ROM sources, a Makefile, jtagtester.exe, jtagturbo.exe, and more.

Setting Up the System for Stand-Alone Debugging

To set up your system for JTAG-based debugging, do the following:

1. Install the Corelis card or the new TriMedia JTAG board on a PC following the manufacturer's instructions. There is no need to use any IRQs, and I/O address 0x140 is usually free on most PCs.
2. Run the self-test program that comes with the card and ensure that it passes the test.
3. If debugging with no host, burn an EEPROM with the L1 JTAG EEPROM image (included in the release) and mount it (the EPROM) on your stand-alone board as the boot PROM (see Figure 10 for location of EEPROM).
4. [Optional] Attach the 40-pin female flat-cable connector to the flat ribbon cable according to the following table.

Table 1 The wiring connection for the flat cable connecting the JTAG controller board to a TriMedia IREF board

JTAG Board Signal	JTAG Board	TriMedia IREF Board	TriMedia Signal
TDO ^A	Corelis Pin 3	JTAG Pin 1	TDI
TDI ^B	Corelis Pin 5	JTAG Pin 3	TDO
TCK	Corelis Pin 9	JTAG Pin 5	TCK
TMS	Corelis Pin 7	JTAG Pin 7	TMS
GROUND	Corelis Pin 2	JTAG Pin 2	GROUND
GROUND	Corelis Pin 4	JTAG Pin 4	GROUND
GROUND	Corelis Pin 6	JTAG Pin 6	GROUND
GROUND	Corelis Pin 8	JTAG Pin 8	GROUND

A. The JTAG board's TDO must be connected to TriMedia's TDI.

B. The JTAG board's TDI must be connected to TriMedia's TDO.

5. [Optional] Attach the 2×4 Connector Header to the other end of the flat ribbon cable.

The cable should be connected to the Header according to Table 1 above.

Refer to Figure 10 to ensure that the Header is plugged into the TriMedia IREF Board correctly.

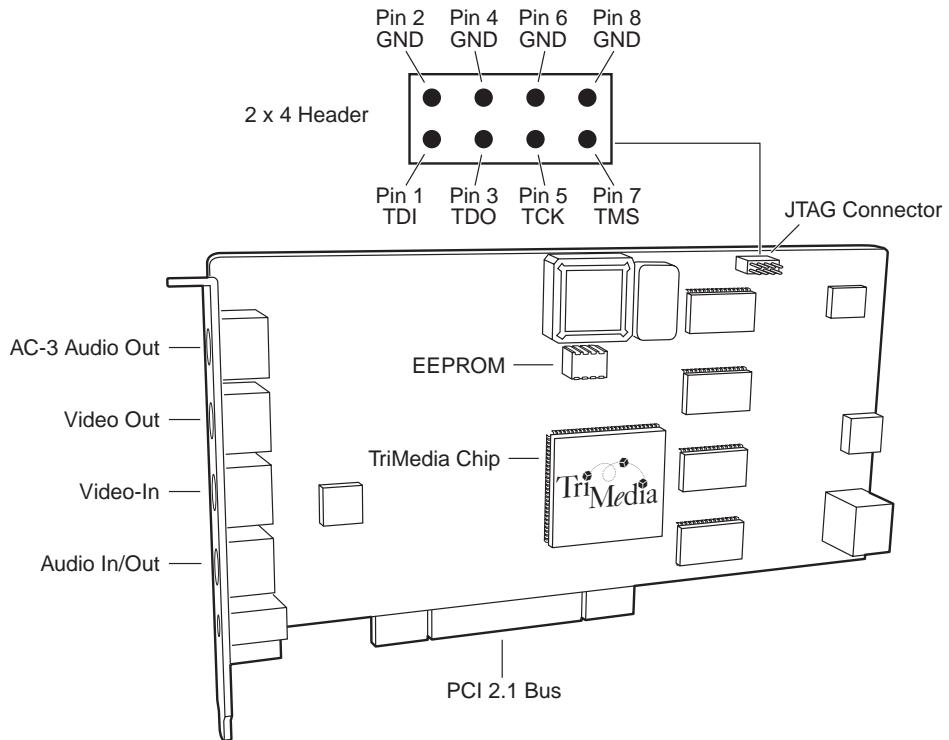


Figure 10 TriMedia IREF Board Version 2.2

6. Connect the 40-pin female flat-cable connector of the connector cable to the Corelis Controller Board.
7. Connect the 2x4 Connector Header of the connector cable to the TriMedia IREF board.

Testing the JTAG Connection

There are two methods to test JTAG connection, depending on which board you are using.

Previously, **tmdbg** assumed that sram base starts at **0** and mmio base starts at **0xe0000000** for stand-alone systems. You could override this via **-memorybase** and **-mmiobase** command line options in stand-alone mode, but that was error prone. The new L1 monitor-**tmdbg** handshake eliminates that possibility.

tmdbg recognizes the new version of the monitor and gets mmio base, sdram base, sdram limit and cacheable limit from the L1 monitor. It uses these values to relocate the L2 code appropriately. If the L1 boot completes successfully, it will download that code (from the L1 monitor) to target via JTAG.

WARNING

Do not use `jtagtester.exe` with turbo boards. Likewise, do not use `jtagturbo.exe` with Corelis JTAG boards.

Testing the Corelis Board JTAG Connection

For the Corelis board, do the following to test the JTAG connection:

1. Power-up the stand-alone system.

If the L1 boot completes successfully, it will wait for application download to start via JTAG. It signals this by writing a magic number (0x12340002) to `JTAG_DATA_OUT` and setting `JTAG_CTRL` to 0 full bit.

2. Run the program `jtagtester.exe`
3. Enter the command `o` (which scans `JTAG_DATA_OUT` register) a few times and you should see the output 0x12340002.

4. Quit the `jtagtester.exe` program by typing `q`.

Sample Session

```
[D:/JTAG/tm1/SMON/Debug] jtagtester
Enter a base address for the jtag board
(0x100, 0x140, 0x240, or 0x300) Default: 0x140:

Base address for the board: 0x140
Using JTAG bus           : 0
Initialized jtag controller
Type h for help
smon>h
  i <value>      -- scan in <value> to In register
  I <In> <Ifull> -- scan in In and Ifull registers
  o              -- scan out Out register
  O <Ofull>     -- scan out Out and Ofull registers
  c <value>      -- write <value> to Ctrl register
  r              -- to reset the chip
  q              -- quit
smon>o
jtag_data_out: 0x12340002
smon>c 0
Read Jtag ctrl: 0x05
smon>q
quiting
```

Testing the Turbo Board JTAG Connection

If you are using the TriMedia JTAG turbo board, do the following to test the JTAG connection:

1. Power-up the stand-alone system.
- If the L1 boot completes successfully, it will wait for application download to start via JTAG. It signals this by writing a magic number (0x12340002) to **JTAG_DATA_OUT** and setting **JTAG_CTRL** to 0 full bit.
2. Run the program `jtagturbo.exe`.
 3. Enter the command **o** (which scans **JTAG_DATA_OUT** register) a few times and you should see the output 0x12340002.
 4. Quit the `jtagturbo.exe` program by typing **q**.

Sample Session

```
C:\TriMedia\bin\jtagturbo.exe
Enter a base address (in hex) for the jtag board
(0x100, 0x140, 0x240, or 0x300) Default: 0x140:

base address for the board: 0x140
Initialized jtag controller
Type h for help
turbo> h
i <value>      -- scan in <value> to In register
I <In> <Ifull> -- scan in In and Ifull registers
o                -- scan out Out register
O <Ofull>      -- scan out Out and Ofull registers
c <value>      -- write <value> to Ctrl register
r                -- to reset the chip
q                -- quit
turbo> r
turbo> o
jtag_data_out: 0x12340002
turbo> q
quiting
```

Compiling a Program for Stand-Alone Debugging

The TCS tool kit contains a monitor library `libmon.o`. This library is automatically linked in to the user program when the program is compiled with **-g** option. To prepare a program for stand-alone debugging, the programs must be compiled as in the following example:

```
tmcc -host nohost -g a1.c a2.c a3.c -o nohost.out
```

The **-host nohost** and **-g** options are very important. The latter option links in a debug monitor. The **-host nohost** parameter links in stubs (which do nothing) for host calls such as `printf`.

Debugging Stand-Alone TriMedia Applications

This section describes how to debug TriMedia stand-alone applications using JTAG in the No-Host and Host-Assisted modes.

IMPORTANT

When debugging stand-alone applications, **tmdbg** expects the target program to have been compiled using the **-g** and **-host nohost** options as outlined in *Compiling a Program for Stand-Alone Debugging* on page 64. If this is not the case, **tmdbg** generates an error message and downloads the program anyway via JTAG, but you will not be able to debug it.

Debugging with No-Host

To debug a TriMedia stand-alone executable (*nhost.out*), do the following:

1. Reset your stand-alone board (on which your TriMedia board is installed) by pressing its reset button.

When the stand-alone board is ready (after reset), **tmdbg** (running on a PC host) can talk to the L1 monitor program on the stand-alone board's EPROM via JTAG.

By default, the L1 monitor program provides **tmdbg** with the following base SDRAM and MMIO values:

- Base-SDRAM: 0
- Base-MMIO: 0x0efe00000

These values can be modified by changing the corresponding values in the files Makefile and L1rom.c that are located in the directory TCS/examples/autoboot/jtag/.

2. Use **tmdbg** to download your target program (*nhost.out*) onto the TriMedia chip.
3. Start debugging.

As previously mentioned, the source-level debugging functionality of **tmdbg** remains the same with the use of either the simulator, or a TriMedia chip in a PC or in a stand-alone system. Following is a sample session:

```

TriMedia C/C++ debugger tmdbg
v1.08 of tcsWin95 (Mar 25 1998 17:34:34)

program probably compiled with -host nohost
using TriMedia Jtag Turbo Card,
please ensure that it is properly installed and tested

target ready for application download
L1 monitor version: 1
receiving target info
MMIO_BASE:          0x0efe00000
DRAM_BASE:          0x0
DRAM_LIMIT:         0x800000
DRAM_CACHEABLE_LIMIT: 0x800000
load addr:          0x840
downloading program ...

```

```

preparing downloadable memory image... done
sent load addr: 0x840
sent code size: 0xa358
one dot = 4K bytes
..... done
target started, waiting for it to initialize ...
done.
stopped at addr: 0x00000a00, line 13, main() in "d1.c"
B 13* main()
tmdbg> b 32
new breakpoint # 1 (line: 32, main() in "d1.c")
tmdbg> b foo
new breakpoint # 2 (line: 43, foo() in "d2.c")
tmdbg> c
target is running, type ctrl-c to stop the target
stopped at addr: 0x00000f80, line 32, main() in "d1.c"
B 32* f8 = rand(); fs[8] = f8;tmdbg> p f7
    f7 = 7419.0
tmdbg> assign f7 = 1111
tmdbg> p f7
    f7 = 1111.0
tmdbg> c
c
target is running, type ctrl-c to stop the target
stopped at addr: 0x00001780, line 43, foo() in "d2.c"
B 43* int
tmdbg> t
t
=>[1] foo(i1 = 12767, f = 2.75, c = 'g', i2 = 12767,
           p2 = 0x1774c, p1 = <struct>), line 43 in "d2.c"
[2] main(), line 43 in "d1.c"
tmdbg> s
stopped at addr: 0x00001880, line 52, foo() in "d2.c"
B 52* p2->age = sum;
tmdbg> p p2
p2 = 0x1774c
tmdbg> p *p2
    evaluates to: struct {
        name = 0x17740 "John Doe"
        age = 25
        misc = array [20] of signed int {
            20, 19, 18, 17, 16,
            15, 14, 13, 12, 11,
            10, 9, 8, 7, 6,
            5, 4, 3, 2, 1
        }
        kids = (nil) <'nh1.out`d2.c`struct person *>
    }
tmdbg> c
target is running, type ctrl-c to stop the target
stopped at addr: 0x00000f80, line 32, main() in "d1.c"
B 32* f8 = rand(); fs[8] = f8;
tmdbg> c
target is running, type ctrl-c to stop the target
stopped at addr: 0x00001780, line 43, foo() in "d2.c"
B 43* int
tmdbg> c
target is running, type ctrl-c to stop the target

```

JTAG Debugging in Host-Assisted Mode

To debug host-assisted stand-alone TriMedia applications, do the following:

1. Use **tmgmon** or **tmon** to download and start the L1 boot program (located in the examples directory: TCS_INSTALL/examples/autoboot/jtag/l1.out) into the TriMedia board in the test PC.
2. Click OK to dismiss the following error message issued by **tmgmon** (Window95 version):
_TMMANSharedPatch:Variable:UNDEFINED.
3. Use **tmdbg** to download a TriMedia stand-alone executable (a program compiled with the **-host nohost** option).
4. Click OK to dismiss the dialog box that asks you to reset the stand-alone board manually because the TriMedia board is not actually installed in a stand-alone chassis.

tmdbg does the following:

- Gets the MMIO and SDRAM base address from the L1 boot program.
 - Relocates the target program's load address to SDRAM's base address plus the 0x840 bytes that represent the 2K (0x800) bytes used by the L1 boot program and a 64-byte (0x40) pad between the L1 and L2 code
 - Downloads the target, via JTAG, as in the no-host case.
5. Start debugging.

Sample Host-Assisted JTAG Debugging Session

This is what you will see in a typical host-assisted JTAG debugging session:

```

TriMedia C/C++ debugger tmdbg
v1.10 of tcsWin95 (Apr 9 1998 13:21:07)

program probably compiled with -host nohost
using TriMedia JTAG Turbo Card,
please ensure that it is properly installed and tested

target ready for application download
L1 monitor version: 1
receiving target info
MMIO_BASE:          0xffffc00000
DRAM_BASE:          0xfffff00000
DRAM_LIMIT:         0xffff800000
DRAM_CACHEABLE_LIMIT: 0xffff800000
load address: 0xffff000000 < sram base: 0xffff000000 + L1 monitor size: 0x840
using load address: 0xffff000840
load addr:          0xffff000840
downloading program ...
preparing downloadable memory image... done
sent load addr: 0xffff000840
sent code size: 0x185a0
one dot = 4K bytes
..... done
target started, waiting for it to initialize ...
done.

```

Notice that the load address is based on the DRAM_BASE (a host assigned PCI address).

Multiple Debugging Sessions

For multiple debugging sessions, do the following after the first session is done and before the downloading of a second TriMedia target program:

1. In **tmdbg**, choose Unload from the File menu, but do not dismiss the dialog box that asks you to reset the stand-alone board manually just yet.
2. Bring up the **tmgmon** window and click the Stop button.
3. Click OK to dismiss the warning message that says:

Program Execution may not have started on TriMedia.

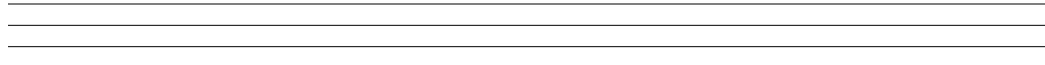
This should stop the L1 program and reset the TriMedia board.

4. Go back to the **tmdbg** window and click OK to dismiss the Board Reset dialog box, mentioned in step 1.
5. Go back to the **tmgmon** window and click Download and Go to start the L1 program again.

tmdbg is now ready to load your next target program.

Chapter 19

Debugging TriMedia pSOS+™ Applications



Topic	Page
Introduction	70
pROBE Functionality in tmdbg —The pSOS+ Monitor	70
Callout Functions	77
Print	78
Pitfalls	79

Introduction

pSOS+ applications differ from standard TriMedia applications in that the pSOS operating system is embedded in them. To debug pSOS+ applications, you can use **tmdbg** in the same way that you use in debug standard TriMedia applications.

However, when debugging pSOS+ applications, first break at the **root** function and then continue because user pSOS+ calls start at **root** (equivalent to **main** in other applications). You can also break in any task function to debug a multi-task application.

Note that **tmdbg** does not yet contain the functionality to support multiprocessors. To debug a pSOS+m application, use methods such as callout functions, DP and Print, which are described later in this chapter.

pROBE Functionality in tmdbg —The pSOS+ Monitor

Several commands have been added to **tmdbg** (both standard and GUI versions) to help you debug pSOS+ applications. These commands are based on the commands as found in ISI's pROBE+ and enable to do the following:

- Inspect the state of pSOS+ objects (such as tasks and queues).
- Get profile information on the application.
- Place breakpoints at pSOS+ events such as service calls and task scheduling. In the future, task-level debugging support will be supported.

Setting Up a pSOS+ Application for Use with tmdbg

To make use of the added functionality, the pSOS+ application must be compiled with **-g** and linked with **libmon.o** and **psosmon.o**. **libmon.o** is linked in automatically when the **-g** flag is used during linking. **psosmon.o** can be found in **\$(TCS)/lib/{el,eb}**. **psosmon.o** contains the functions called by the pSOS+ kernel to keep the profile up to date, and the functions called by **tmdbg** to get the information on the pSOS+ objects.

Inspecting pSOS+ Objects

To inspect the state of pSOS+ objects or the kernel's global state, the following functions are provided:

qc	Query pSOS+ configuration
qd	Querydate
qo	Query objects
qp	Querypartitions

qq	Queryqueues
qr	Queryregions
qs	Querysemaphores
qt	Querytasks

Each of these functions are only operative when the pSOS+ kernel is initialized. This means that the user must execute a program up to the **root** function to use them. When started, **tmdbg** will halt at **main**, so normally the user will first do a “**b root**” to set a breakpoint at function **root** and then a **c** command to continue up to that point.

Query pSOS+ Configuration

The **qc** command will list the pSOS+ configuration table as specified by the user in **sys_conf.h**.

```
tmdbg> qc
pSOS Configuration
-----
Table Address      8004c6d0
pSOS code         8004d22c
I/O jump table   8004d2c0
System Stack Size 00008000
----- Max number of -----
Tasks      Queues Semaphores I/O Devs   Objects
----- 5       6        7      2      20
----- Callout Functions -----
Fatal     Start    Delete   Switch
00000000 8003218a 00000000 00000000
----- Region 0 -----
Address      Size   Unitsize
80059d58 00795e70 00000100
- Root Task Initial Parameters -
Address Prio Mode Stack Size
800003c0 c8 2000 00008000
----- Ticks per      Ticks per -----
Timing      second     Slice
----- 500          10
```

Query Date

The **qd** command will show the time as set in the pSOS+ application in date, time, and ticks.

```
tmdbg> qd
      Date      Time      Ticks
-----
29-OCT-1997  1:31:23        7
```

Query Object

The **qo** command will show the user the list of objects currently available in the application. An argument can be passed to **qo** to only show objects of a specific type.

```
tmdbg> qo
Object      ID      Type
-----
'SEM'      #00030000  Semaphore
'RN#0'     #00000000  Region
'ROOT'     #00020000  Task
'TSK1'     #00070000  Task
'QUE3'     #00060000  Queue
'QUE1'     #00040000  Queue
'TSK2'     #00080000  Task
'QUE2'     #00050000  Queue
'IDLE'     #00010000  Task

tmdbg> qo queue
Object      ID      Type
-----
'QUE3'     #00060000  Queue
'QUE1'     #00040000  Queue
'QUE2'     #00050000  Queue
```

Query Partition

The **qp** command will provide the user with information about the partitions currently available in the pSOS+ application. An argument can be passed to **qp** that identifies a specific partition, in which case, only information on this partition will be shown.

```
tmdbg> qp
          Buffer  Number of   Free    Delete
Partition      Address  Size    Buffers  Buffers  Override
-----
'PT_1'  #00120000 80094200 000100        16      15    yes
```

Query Queue

The **qq** command can be used to get information on the queues. It accepts an argument to specify one queue, in which case, it will print more detailed information.

```
tmdbg> qq
      Waiting Queue  Max. Buffer Queue Message
Queue   Tasks Length Length Mgt. Type Size
-----
'QUE3' #00060000    0     1     16 local fifo variable
'QUE1' #00040000    0     1 no max sys-pool fifo 16 bytes
'QUE2' #00050000    0     0     6 sys-pool fifo 16 bytes

tmdbg> qq 0x40000
      Waiting Queue  Max. Buffer Queue Message
Queue   Tasks Length Length Mgt. Type Size
-----
'QUE1' #00040000    0     1 no max sys-pool fifo 16 bytes
-----
Waiting Tasks: NONE
-----
Messages:      Address  Size
(hex)          ----- 
               800599e0  0010
```

Query Region

The **qr** command will inform the user about the usage of the regions at that point by the pSOS+ application. Again, a specific region can be specified, resulting in more detailed information.

```
tmdbg> qr
      Free  Largest Wait. Delete Queue
Region  Address  Size Unitsize Bytes Segment Tasks Overr. Type
-----
'RN#0' #00000000 80059d58 00769600 000100 0074a200 00745200 0 fifo

tmdbg> qr RN#
      Free  Largest Wait. Delete Queue
Region  Address  Size Unitsize Bytes Segment Tasks Overr. Type
-----
'RN#0' #00000000 80059d58 00769600 000100 0074a200 00745200 0 fifo
-----
Waiting Tasks: NONE
-----
Segments:      Address  Size  Status
(hex)          ----- 
               80050800 00009558 pSOS
               80059d58 0002c800 header
               80086558 00745200 free
               807cb758 00000100 in use
               ...
               807e7a58 00008100 in use
               807efb58 00000070 unusable
```

Query Semaphore

The **qs** command, with optional argument to specify a definite semaphore, shows the user information on the current state of the semaphores in the pSOS+ application.

```
tmdbg> qs
          Waiting
Semaphore   Count   Tasks   Type
-----
'SEM ' #00030000 00000000      1   fifo

tmdbg> qs "SEM "
          Waiting
Semaphore   Count   Tasks   Type
-----
'SEM ' #00030000 00000000      1   fifo
-----
Waiting Tasks:
#00070000
```

Query Task

The **qt** command is offered to get information on the current tasks in the system.

```
tmdbg> qt
          Prio
Task       rity Mode Status           Ticks to
          Timeout Suspended
-----
'ROOT' #00020000 c8 0000 Running
'TSK1' #00070000 c8 0000 Smwait 'SEM ' #00030000    9   yes
'TSK2' #00080000 c8 0000 Wkafter
>IDLE' #00010000 64 0000 Ready

tmdbg> qt ROOT
          Prio
Task       rity Mode Status           Ticks to
          Timeout Suspended
-----
'ROOT' #00020000 c8 0000 Running
-----
Soft registers:
00000000 00000000 00000000 00001234 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
-----
Initial PC     800003c0 Pending events 00000000
Initial priority c8 ASR address 00000000
Initial mode    0000 ASR mode      0000
Stack bottom   807df558 Pending ASR  00000000
Stack top      807e7558
-----
Timers: NONE
```

Getting Profile Information of a pSOS+ Application

tmdbg automatically keeps track of several events during execution of a pSOS+ application. The information gathered by **tmdbg** can be accessed with the command **lp** (for list profile). Other commands are **cp** (for clear profile) and **prof on** and **prof off** (for profiling on/off). At the start-up of **tmdbg**, profiling is set “on.” As shown in the examples, **tmdbg**

keeps track of how many cycles are overhead (in pSOS+ start-up, *psosmon.o*, and *libmon.o*) and how many cycles are spent in each task. Furthermore, **tmdbg** keeps track of how many times a task has run, and how many times it has been preempted or blocked. Finally, the number of pSOS+ service calls done by every task is being counted. For every queue, **tmdbg** keeps track how often certain system calls are performed and what the maximum number of messages in the queue has been during the profile interval.

```
tmdbg> lp

Profile Interval: 2 ticks == 00072a3ba cycles (hex)

system startup      0006cd670
pSOS+ startup       00001c33d cycles (hex)
psosmon             000031507 cycles (hex)

-----Running----- -----Times----- ---Calls---
Task          Ticks    Cycles   Running Blocked Preemptd pSOS    I/O
-----
'TSK1' #00070000    0  00000007d6     2       1       1      3      0
'TSK2' #00080000    0  00000003f6     1       1       0      1      0
'ROOT' #00020000    2  00000e935      3       0       2      36      0
'IDLE' #00010000    0  0000000000     0       0       0      0      0

----- Number of Calls ----- Maximum
Queue          q_send w_urgent q_broadc q_receiv Length
-----
'QUE1' #00040000    1      0      0      0      1
'QUE2' #00050000    0      0      0      0      0
'QUE3' #00060000    1      0      0      0      1

----- Number of Calls ----- Maximum
Semaphore        sm_p    sm_v   Value
-----
'SEM ' #00030000    2      0      0

----- - Number of Calls - Minimum Largest
Region          rn_getseg rn_retseg Segment (hex)
-----
'RN#0' #00000000    13      2      00745200

----- - Number of Calls - Minimum
Partition        pt_getbuf pt_retbuf Free Buffers
-----
'PT_1' #00090000    1      0      249

tmdbg> lp task

Profile Interval: 2 ticks == 000111d45 cycles (hex)

pSOS startup      0000d133d cycles (hex)
psosmon             000031507 cycles (hex)

-----Running----- -----Times----- ---Calls---
Task          Ticks    Cycles   Running Blocked Preemptd pSOS    I/O
-----
'TSK1' #00070000    0  00000007d6     2       1       1      3      0
'TSK2' #00080000    0  00000003f6     1       1       0      1
'ROOT' #00020000    2  00000e935      3       0       2      36
'IDLE' #00010000    0  0000000000     0       0       0      0      0
```

pSOS+ Breakpoints

With psosmon.o linked in your application, it is now possible to set breakpoints at certain pSOS+ events. The new **tmdbg** commands involved are **bp**, **bt** and **dp**.

Command	Description
bp	Break when a specific system call is called
bt	Break when a task gets scheduled in or out
dp	Delete pSOS+ breakpoint

The **tmdbg** commands **status** and **delete all** also show and delete pSOS+ breakpoints, respectively.

Note that when a pSOS+ breakpoint is hit, it is not yet possible to look at the stack trace to see where exactly in the application this happened.

Breakpoint on pSOS+ System Calls

A breakpoint can be set on the event of a pSOS+ system call using **bp**. Most system calls identify the object on which it operates by name (for create) and by ID (for all other calls on the object). The name or ID, therefore, can be used as a argument to the **bp** command, to indicate a specific pSOS+ object.

Examples

The following are examples of breakpoints that may be set:

```
tmdbg> bp q_send
New pSOS+ breakpoint # 2 (call q_send)

tmdbg> bp q_create QUE1
New pSOS+ breakpoint # 3 (call q_create with argument 'QUE1')
```

Breakpoints on Task Scheduling

A breakpoint can be set on the event of a task being blocked or preempted using the command **bt**. **bt all** can be used to stop on any task scheduling, while **bt taskname** or **bt task_id** can be used to stop whenever a specific task gets blocked, preempted, or starts running.

Examples:

```
tmdbg> bt all
New pSOS+ breakpoint # 4 (on all task scheduling)

tmdbg> bt ROOT
New pSOS+ breakpoint # 5 (on scheduling task 'ROOT')
```

Deleting pSOS+ Breakpoints

dp breakpoint_id can be used to remove a breakpoint, while **dp all** will remove all pSOS+ breakpoints. The standard tmdbg command **delete all** will delete all pSOS+ breakpoints, as well as other breakpoints.

tmdbg does not yet contain the functionality to support multiprocessors. To debug a pSOS+m application, use methods such as callout functions, DP and Print, which are described later in this chapter.

Callout Functions

pSOS provides *callout functions*, a mechanism for notifying the application when certain events occur. The events include task start, task delete, task switch, and fatal. To use this mechanism for debugging, first declare the callout functions *in sys.conf.h*

```
extern void task_start ( unsigned long tid, void * tcb );
extern void task_delete( unsigned long tid, void * tcb );
extern void task_switch( unsigned long entering_tid,
                        void *entering_tcb,
                        unsigned long leaving_tid,
                        void *leaving_tcb );
extern void user_fatal ( unsigned long err, unsigned long flag );

#define KC_STARTCO ((void ()()) task_start)
                  /* callout at task activation */
#define KC_DELETECO ((void ()()) task_delete)
                  /* callout at task deletion */
#define KC_SWITCHCO ((void ()()) task_switch)
                  /* callout at task switch */
#define KC_FATAL     ((void ()()) user_fatal)
                  /* fatal error handler */
```

Next, add definitions to these functions in your code. For example,

```
void task_start ( unsigned long tid, void * tcb ){
    DP(("task_start: %d\n", tid));
}

void task_delete (unsigned long tid, void * tcb){
    DP(("task_delete: %d\n", tid));
}

void task_switch( unsigned long entering_tid, void *entering_tcb,
                 unsigned long leaving_tid, void * leaving_tcb ){
    DP(("task_switch: entering %d, leaving %d\n", entering_tid, leaving_tid));
}

void user_fatal( unsigned long err, unsigned long flag ){
    DP(("user_fatal: err = %d, flag = %d\n", err, flag));
}
```

If you want to print the name of the task instead of the task ID, keep a table of the task ID and corresponding name at the time of **t_create**, because the name of the task is not passed as an argument to these callout functions.

You can use **tmdbg** to break at these callout functions and step through them. Furthermore, you can use the **DP** command without having to use **tmdbg** to debug pSOS applications to track their real-time behavior.

The callout functions listed in this example use the **DP** command to print information prepared by the callout functions.

Print

For debugging without **tmdbg**, **printf** is recommended and can be used in most pSOS applications. However, because **printf** schedules a task switch, it can affect the execution order of the tasks. The user must provide a non-scheduling printing mechanism, if needed. An example of this mechanism, called **Print**, can be found in `$(PSOS_SYSTEM)/configs/print.c`. Please note that **Print** and **printf**, unlike **DP**, cannot be used from an ISR. The following code from `print.c` suspends scheduling during a call to **vprintf**.

```
#include <stdarg.h>
#include <tmlib/AppModel.h>

unsigned long Print( char *format, ... ){
    unsigned long result;
    AppModel_AppId self;

    va_list arg_pt;

    AppModel_suspend_scheduling();

    self= AppModel_switch(AppModel_root);
    va_start(arg_pt, format);
    result= vprintf(format, arg_pt);
    AppModel_switch(self);

    AppModel_resume_scheduling();

    return result;
}
```

Pitfalls

Keep the following in mind when debugging a pSOS application:

- Check the stack sizes in *sys_conf.h*. (See [sys_conf.h](#) in Chapter 6, *pSOS+™ Real-Time Operating System* in Book 3, *Software Architecture*, Part A.)
- Do not use **the following** from an ISR:
 - **printf**
 - **Print**, the user-defined non-blocking, non-scheduling **printf**, modeled after \$(PSOS_SYSTEM)/configs/print.c
 - Any pSOS system call that is mentioned in the *pSOS Reference document* is not to be called from an ISR

The reason for this precaution is that a file descriptor can still be locked when the interrupt handler attempts to print. This situation would result in a deadlock.

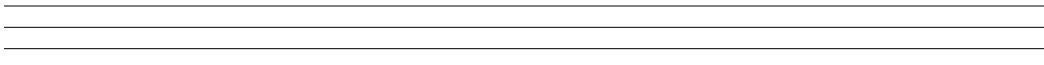
- Do not delete a task that is printing, as the file descriptor would remain locked forever.
- tmdbg does not check for single stepping into library code or the stripped code of pSOS. This happens, for example, when single stepping into sbrk.

Use the following workaround:

- Keep on stepping until you get back to the code on the screen.
- Set a breakpoint on the place to which to return.

Chapter 20

Debugging TriMedia Applications Using printf and DP



Topic	Page
Introduction	82
Debugging Using printf	82
Comparing DP and printf	82
Using DP	83

Introduction

The following sections describe how to use the **printf** and **DP** debugging techniques to debug your applications.

Debugging Using printf

On TriMedia, your secondary debugging tool is the **printf** function. The TriMedia **printf** function is included in the C RunTime library. It is implemented using a remote procedure call (RPC) mechanism, which is part of the host-support package.

Comparing DP and printf

printf as implemented on the TriMedia processor is a rather complicated procedure, involving communication with the host. That is why **printf** is not always useful for debugging. TriMedia provides another function (DP) that is much more appropriate. Table 2 compares the **DP** and **printf** techniques.

Table 2 DP and printf comparison

DP	printf
<i>Fast</i> Writes to memory, not screen.	<i>Slow</i> Writes to screen.
<i>Synchronous</i> Returns when completed.	<i>Asynchronous</i> In a multitasking environment, control could be transferred to another task while printing completes.
<i>Simple</i> Writes only to memory.	<i>Complex</i> Passes messages to the PC over the PCI bus using interrupts.
<i>Requires action to read output</i> You must dump the DP buffer.	<i>Does not require action</i> Output goes to the PC screen as soon as possible.
<i>Persistent</i> The contents of the DP buffer are preserved across a warm boot.	<i>Volatile</i> A crash to your PC destroys any debugging information you may have collected.
<i>Can be called from an Interrupt Service Routine (ISR)</i>	<i>Cannot be called from an ISR</i>

Note

Examine the audio test program (atest.c) to see some typical uses of **DP**. Details of errors that you don't want to show to users are reported by **DP**. You can call **DP** in an ISR to check whether an ISR is running. You can dump megabytes of trace information into the DP buffer. In this case, it is best to dump the DP buffer to a file.

Using DP

IMPORTANT

DP is a macro. The DP capability is always enabled unless **NO_DP** is defined.

DP is defined as a macro in tmlib/dprintf.h. It works like a simple form of **printf**. You must call **DP** with two sets of parentheses. For example, the command

```
DP(( "test %d\n", i));
```

allows **DP**, which is a varargs function, to be conditionally compiled out.

DP is mapped to the function **_dp** and works on all hosts (including **nohost**). **DP** has an associated initialization function that must be called before calling **DP**: **DP_START** will create a buffer where subsequent **DP** will write to.

DP can handle strings of up to 511 bytes long; your program will exit with value **0xFFFFD30F** if more than 511 characters have to be copied to the **DP** buffer in one call.

A powerful feature of this debugging subsystem is the persistent **DP** buffer that survives a warm reboot. Using this feature, you can analyze a long trace of the progress of a real-time system off-line.

You can dump the **DP** buffer using **tmmon** or **tmgmon** on Windows 95 and Windows NT.

The **DP** buffer can also be inspected through the GUI version of **tmdbg**.

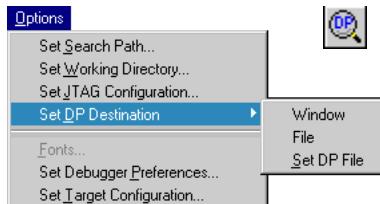


Figure 11 DP Options in tmdbg.

Description of DP Macros

`Address DP_START(UInt32 size, Address address)`

Synopsis	Initializes buffer and controlling struct for DP.				
Parameters	<table> <tr> <td><code>size</code></td> <td>Size of the buffer.</td> </tr> <tr> <td><code>address</code></td> <td>Address of user-provided buffer. Null if the buffer needs to be dynamically allocated.</td> </tr> </table>	<code>size</code>	Size of the buffer.	<code>address</code>	Address of user-provided buffer. Null if the buffer needs to be dynamically allocated.
<code>size</code>	Size of the buffer.				
<code>address</code>	Address of user-provided buffer. Null if the buffer needs to be dynamically allocated.				
Returns	Address of controlling struct.				

`DP_STOP(void)`

Synopsis	Frees allocated DP buffer memory. The DP control structs are overwritten. After <code>DP_STOP</code> , the contents of the DP buffer are irretrievably gone.
----------	---

`DP_ON(void)`

Synopsis	Switches DP printing back on after a <code>DP_OFF()</code> . Subsequent calls to DP will have effect again. A <code>DP_ON()</code> is not needed after a <code>dp_START</code>
----------	---

`DP_OFF(void)`

Synopsis	Switches DP printing off. Subsequent calls to DP will have no effect.
----------	--

IMPORTANT

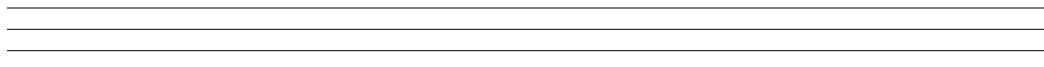
Always use the `DP_*` macros to do your “debug printf.” Do NOT call the underlying functions. The `DP_*` macros accept the same arguments as their corresponding functions. Once you have finished debugging, simply recompile your source with the `-DNO_DP` option. All of the `DP_*` macros will be compiled out, that is, they become comments and do not impact on the final code size or execution speed at all.

Note

For backward compatibility the macro `DPsize` is still defined: `DPsize(size)` is equivalent to `DP_START(size, Null)`.

Chapter 21

Debugging Multiprocessor and Multitasking Applications



Topic	Page
Introduction	86
Loading a Multiprocessor Application	86
Switching Focus	87
The procs Command	87
Debugging Tasks	87

Introduction

Debugging multiprocessor applications is similar to debugging other application using the GUI version of the TriMedia debugger. The main difference is in how you load the target and how you set the focus of execution.

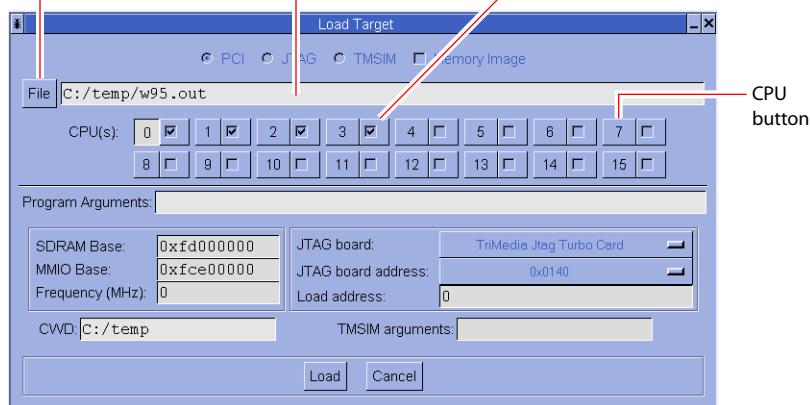
Loading a Multiprocessor Application

The Load Target dialog box enables you to load a single TriMedia executable (.out file) into multiple CPUs. It also enables you to load multiple .out files into multiple CPUs, but the host type should be the same for the multiple CPUs.

If you locate the executable using the File button, **tmdbg** checks the file automatically to find out the host on which it is designed to run

If you manually enter the filename, you must press Enter so that **tmdbg** checks the file automatically

Checkbox for selecting a CPU. The executable is downloaded onto the selected CPUs.



In the Load Target window, clicking a CPU button displays the CPU's corresponding SDRAM Base, MMIO Base, Frequency, and current working directory (CWD) values.

For PCI-hosted .out files, the SDRAM Base, MMIO Base, and Frequency fields are ready-only. This information is defined at PCI board installation time and cannot be changed.

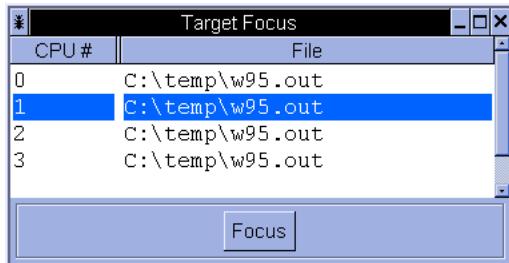
For TMSIM executable, a current working directory is not supported.

To load an executable, click the Load button. The .out file is loaded into the selected CPUs and a tmrun console opens for each CPU.

Switching Focus

To switch the execution focus, do the following:

1. Choose Focus from the View menu.
2. Select the desired CPU from the Target Focus window.
3. Click Focus.



Once in focus, all commands apply to the same CPU.

Note

Clicking on a source code window does not change execution focus.

The procs Command

There is no equivalent command for **procs** in the GUI version of the TriMedia debugger.

Debugging Tasks

To debug tasks using **tmdbg**, you must link the target program with the pSOS+ operating system, and the pSOS debug monitor. (Refer to *Setting Up a pSOS+ Application for Use with tmdbg* in Chapter 19.) The debugger supports debugging tasks in the **System** (default mode) and **Task** modes.

If the target is stopped in the System mode, all the tasks on the target stop. Interrupts are still enabled, but the debug monitor is entered, and does not yield to any other tasks which may be ready.

If the target is stopped in the Task mode, **tmdbg** supports debugging a single task while other tasks are running. Therefore, as long as one task is stopped, you can enter **tmdbg** commands. To switch to the Task mode, use the Debug Mode command in the options menu.

IMPORTANT

The Task mode can only be entered before the target program has started (that is, when the target is stopped at **main**).

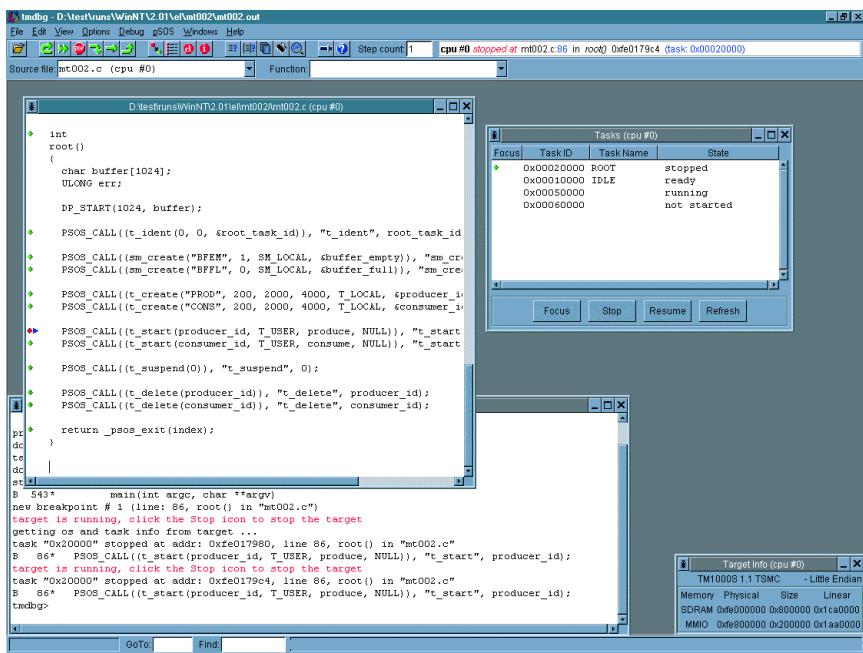


Figure 12 Debugging Multi-task Applications

The TriMedia debugger distinguishes between the “visited” tasks and the tasks that are “in focus” in whether their states are modifiable.

A task being visited can have its state examined but not changed (that is, the call stack, registers, and memory can be viewed, but the task cannot be explicitly restarted by the debugger). Only the usual operating system mechanism can restart that task.

When a task is in focus, however, its state is viewable as well as modifiable. Such a task can be restarted by the debugger without changing the state of any other task.

Debugging Tasks in System Mode

In the System mode, any task can be visited, but only one task is in focus at any time, and the focus cannot be changed, except by continuing execution and letting the operating system perform a task switch.

In the Tasks window, a diamond in the Focus column marks the task that is currently in focus.

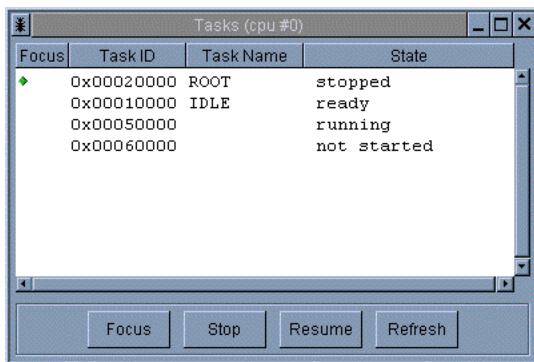


Figure 13 The Tasks window

When the target program stops on a breakpoint, the task currently in focus is the task being visited. To visit a different task, highlight the task, and click the Visit button. You can then view the call stack, registers, and memory information for the newly visited task. But remember that this task is not the one in focus. Clicking the continue button, for example, applies to the task in focus and not to the task being visited.

Debugging Tasks in the Task Mode

In the Task mode, you can visit or focus on any task that is stopped. For example, although task A may have hit a breakpoint, you can focus on task B and let it continue before A restarts.

If a task is not stopped, you can stop it by clicking the Stop Task button in the Tasks window¹. In the same way, you can restart tasks using the Resume Task button.

The Focus Task and Visit Task buttons are both active in the Task mode. As in system mode, however, only the focused task can be restarted.

1. The act of stopping a task is not precise. In some cases, you cannot stop a task until it makes a system call.

Chapter 22

Other Debugging Information

Topic	Page
Debugging Interrupt Handlers	92
Debugging the Host Call Interface and Device Library	93
Debugging at Optimization Levels Higher Than -O1	94

Debugging Interrupt Handlers

The TriMedia register set is divided into global and local registers. Global registers are used for values that are alive across decision trees. For more information regarding calling conventions, see [Chapter 21, TriMedia Interrupts API](#), in Book 5, *System Utilities*, Part C. Generally, when a function is called, it must save the global registers that it will use to carry values across decision trees.

The debugger, **tmdbg** (running on the host machine), interacts with the target program (running on TriMedia) via a debug monitor (`libmon.o`, linked in with user code when `-g` is used for compiling and linking). The debug monitor runs on TriMedia with the target program. Breakpoints can be set only at decision tree boundaries. When a breakpoint is reached, the debug monitor is invoked and it immediately saves all the global registers. Local registers need not be saved because they are not live across decision tree boundaries. The debug monitor then sends a message to **tmdbg** and awaits a reply.

Interrupt Handler Characteristics and Problems

The local registers are used only by the instruction scheduler for values that are alive within a decision tree, but dead across trees.

tmdbg does not support breakpoints at or inside typical interrupt handlers. This is because of two distinct characteristics of the manner in which interrupt handlers are scheduled and breakpoints are processed.

- Interrupt handlers that use local registers.

The first problem arises because of the way interrupt handlers (those that are compiled with `#pragma __TCS_handler__`) are compiled. Interrupt handlers are scheduled to run quickly, so they do not save any registers.

Instead, they use the local registers, none of which should be live when the handler is invoked. All decision trees in an interrupt handler use local registers to pass information from one tree to another, contrary to the usual convention for non-handler code. Since local registers are not saved by the debug monitor, breaking within an interrupt handler causes local register contents to be lost.

- Breakpoint triggering with unpredictable results.

The second problem occurs when a breakpoint in an interrupt handler X is triggered. The debug monitor tries to send a message to **tmdbg** and waits for a reply. However, during this time, another interrupt may occur and trigger the handler X again, re-entering the monitor. This scenario may be repeated and an infinite loop may occur. Since the monitor is not re-entrant, breaking in an interrupt handler in general will cause unpredictable results.

Interrupt Handler Solutions

The solution to the problems caused by the characteristics of the interrupt handlers is to debug interrupt handlers by using the following approach:

- When the handler is invoked, it must first disable the interrupt that triggered it. Then, to force the local registers to be saved, the handler should call another function. Following is an example.

```
#include <tm1/tmInterruptions.h>
void safe_handler_bp(void) { ; }

void int_handler(void){
    #pragma __TCS_handler__
    safe_handler_bp();

    /* process interrupt */
}
```

Setting Breakpoints

To debug the handler, you can now set a breakpoint in the function `safe_handler_bp`, run the program until the breakpoint is reached, do a `fin` and proceed to debug the handler. You should let the execution continue when it reaches the line containing `intSET_IEN`. If you try to step through the code after the interrupts have been re-enabled, the second problem mentioned may occur. It should also be noted that if you are running `tmdbg` with `tmsim` as the target, you should not step into the Device Library disable and enable interrupt calls (as described in the following section).

Debugging the Host Call Interface and Device Library

Since the monitor sends a message to `tmdbg` each time it is invoked by a breakpoint, it is not possible to set breakpoints inside of the message-passing routines. Setting a breakpoint will cause the monitor to be entered, which will cause a message to be sent to `tmdbg`, which will invoke the message-passing routines, which will cause a breakpoint, which will cause the monitor to be re-entered, and so on. The results are either an infinite loop or unpredictable.

This is only an issue when debugging using `tmsim` because, in this case, the message-passing routines are part of the TriMedia C library. When `tmsim` is the target, `tmdbg` uses the socket routines `sock_recv` and `sock_send` to communicate with the monitor. These routines are implemented by the TriMedia C library on top of `HostCall_send` and `HostCall_host_send`.

Furthermore, these calls are implemented on top of the Device Library. Breakpoints may not be set inside any of this code. If you happen to interrupt execution by typing CTRL-C and land within any of this code, you should not use the `step`, `next`, or `finish` commands. Inquiries can be made, but a breakpoint set will trigger the problem

described previously. When **tmdbg** runs on the TriMedia processor, the monitor does not rely on any of the TriMedia library, so this problem will not occur.

Debugging at Optimization Levels Higher Than -O1

Although the compiler defaults to -O1 when you specify **-g**, it is possible to compile with **-g** at a higher optimization level. To do this you must state the optimization level you want along with the **-g** command (**-g -O3**, for example). While debugging at higher optimization levels is not explicitly prohibited, it is not supported at optimization levels greater than **-O2**.

There are several differences that you should expect when attempting to debug at **-O2** or higher. These are described in the sections below.

Diminished Visibility of Local Variables and Parameters

At **-O2** or higher, the compiler uses registers extensively to hold temporary or intermediate values. In many cases, the original discrete local variable or parameter does not exist after the first dtree of the code and is only kept in a form that aids in optimization. For example, consider the following loop, where **i** is an integer and **a** and **b** are arrays of integers.:

```
for( i = 6; i < m ; i++ ) a[i] = b[i] - m;
```

In this case, **i** is used only as an index. Array indexing is an expensive calculation because **a[i]** implies that **a+(i*sizeof(int))** and multiplication are more expensive than addition. The compiler may choose to keep only the value of **i** (as **i*4**) and just add 4 at the end of the loop, conceptually changing the code to the following.

```
for( i = 24; i < m*4; i += 4 ){
    t1 = a + i;
    t2 = b + i;
    *(t1) = *(t2) - m;
}
```

Or the compiler may even go as far as this:

```
t1 = a + 24;
t2 = b + 24;
for( x = 0 ; x < m; x++, t1 += 4, t2 += 4 )
    *t1 = *t2 - m;
```

In many cases the local or parameter value is available for the first few dtrees in the function and then becomes unavailable even though, looking at the code, you can see it is still in scope.

Similarly, if the value of the **return** statement is determined early, it may be moved to the return register early and the debugger may no longer know the variable by its original name.

Local variables could also be unavailable because the compiler has reordered the code to make all references to a given variable local to the same dtree.

Variables May Have Been Optimized Away

Parameters or variables that are set but never used may be completely removed. More confusingly, variables that are only copies of other variables may be removed even if they are different types. For example:

```
int foo (char f) {
    int tmp = f;
    printf("f=%c tmp=%d\n", f, tmp);
    return tmp;
}
```

In the case above, even though **f** is a **char** and **tmp** is an **int** they can both be represented by the same register. The compiler knows that **f** is in register 5 on entry. It also knows that the value in register 5 is equivalent to **tmp** so it throws away **tmp**.

Code May Have Been Moved

This means that a value may have already been updated despite the fact that the source code has the value updated at a point below the current breakpoint. The simplest example of this is when a variable is determined to be invariant during a loop and is assigned outside the loop.

In order to create maximum opportunity for parallelism, the compiler will ‘hoist’ code from sections below the current code into the dtree with the current code. For example, a dtree (say DT_5) could have code generated from lines 127–130 and 150–154 because a **goto** at line 130 goes to line 150 and the compiler has brought the code into the same dtree.

The compiler may also move all references to a local variable into the same dtree. Doing so can cause confusion. Although visually it seems that the variable should be available at the given breakpoint, the compiler has moved all references into the same dtree, and the variable is not available.

Code May Have Been Unrolled

While you are looking at code that goes sequentially through a char array the compiler may have unrolled it to actually do 2 or 4 values at every loop iteration. Values may be updated before you think they should be.

Code May Have Been Inlined

When an instance of a function is inlined into another function the function call is removed and your access to the local variables or parameter of the function that was

inlined is also removed. The inlined function itself creates more chances for optimization and the code that is actually executed may be quite different from the code that you are looking at in the debugger.

Chapter 23

tmdbg Command Reference

Topic	Page
Overview of Debugger Commands	98
Debugger Expressions	98
Debugger Commands	98

Overview of Debugger Commands

The following are the most basic **tmdbg** commands:

break	Place a breakpoint in the program
continue	Continue execution
help	List available commands and their usage
print	Display variables
run	Run the program being debugged
status	List currently active breakpoints
where	List call stack trace

This release fully implements all essential features such as setting and removing breakpoints, listing of source files, and examining program data structures. The following sections provide reference information that you may need when using the debugger. You can also find the most up-to-date reference information in **tmdbg** manual page.

Debugger Expressions

tmdbg expressions are combinations of variables, constants, function calls, and operators. Variables are the currently active and visible C variables defined in the program being debugged. Hexadecimal constants must be preceded by a **0x** and octal constants by a **0**. Character constants must be enclosed in single quotation marks. In general, the expression rules follow those of the C language.

tmdbg updates the current scope as the program enters and exits functions and files during execution. **tmdbg** resolves scope conflicts based on the current function and file. You can also change the scope explicitly with the **func** and **file** commands. When the current function is changed, **tmdbg** updates the current file accordingly, and vice versa.

Debugger Commands

Debugger commands are listed according to the following categories:

- Execution control commands
- Data and stack commands
- Source files commands
- pSOS+ commands
- Miscellaneous commands

Execution Control Commands

The following are the commands which control execution.

break <i>n</i>	Set a breakpoint at source line <i>n</i> .
break <i>func</i>	Set a breakpoint at the beginning of function <i>func</i> .
break <i>n</i> "file"	Set a breakpoint at line <i>n</i> in source <i>file</i> .
break <i>addr address</i>	Set a breakpoint at the given address. The address must be a branch target. If not, the text segment might become corrupt. The debugger will check whether the address is a branch target. If an executable is stripped, verification is not possible.
call <i>func()</i>	Make a function call to a function in the target program. Only functions of the type void f(void) , are currently supported.
cont	Continue execution of the program.
delete <i>n</i>	Remove breakpoint number <i>n</i> . The status command prints breakpoint numbers.
delete <i>dbp</i>	Remove datawatch breakpoint (set by the watch command).
delete all	Remove all breakpoints.
finish	Finish executing the current function, then stop at the caller. This is identical to the step up command.
hwbreak <i>line</i> [<i>after n</i>]	Set a hardware breakpoint at source <i>line</i> . If the after field is provided, then the breakpoint is triggered after <i>n</i> executions.
hwbreak <i>func</i> [<i>after n</i>]	Set a hardware breakpoint at the beginning of function <i>func</i> . If the after field is provided, then the breakpoint is triggered after <i>n</i> executions.
hwbreak inrange <i>addr1 addr2</i> [<i>after n</i>]	Set a hardware breakpoint in the range of instructions <i>addr1</i> to <i>addr2</i> (inclusive). If the after field is provided, then the breakpoint is triggered after <i>n</i> executions.
hwbreak outrange <i>addr1 addr2</i> [<i>after n</i>]	Set a hardware breakpoint outside of the range of instructions <i>addr1</i> to <i>addr2</i> (exclusive). If the after field is provided, then the breakpoint is triggered after <i>n</i> executions.
load <i>prog</i>	Download a TriMedia executable named <i>prog</i> . The currently loaded program will be terminated.

loadmi <i>mifile</i>	Download a memory image file <i>mifile</i> to a stand-alone Tri-Media board via JTAG. The memory image must not be created with the debug monitor, libmon.o, and linked in or else it hangs after it is downloaded. There is no debugging support for memory image files.
next [<i>n</i>]	Step <i>n</i> lines (default: one line); do not step into function calls. See also step .
run [<i>arg...</i>]	With no argument, begin executing the target program with current arguments. Otherwise, begin executing the target program with the given arguments <i>arg1</i> , <i>arg2</i> , and so on. Arguments containing non-alphanumeric characters (such as “.” or “/”), must be enclosed in quotation marks.
status	Print the current breakpoints.
step [<i>n</i>]	Single-step <i>n</i> decision trees, stepping into function calls. The default is <i>n</i> = 1 decision tree. See also next .
step up	Finish executing the current function, then stop at the caller. Same as the finish command.
stop	stop is synonymous with break .
watch <i>loadstore</i> [<i>range</i> <i>value</i>] [<i>passcount</i>]	The <i>loadstore</i> specifier denotes whether loads, stores, or both, should be watched. <i>loadstore</i> ::= load store loadstore <i>range</i> ::= [inrange outrange] <i>addr</i> <i>addr</i> <i>value</i> ::= when [== !=] <i>const</i> <i>mask</i> <i>passcount</i> ::= after <i>const</i>
	The <i>range</i> specifier requests that addresses should be checked in the given range.
	The <i>value</i> specifier allows specific values to be compared to a constant <i>const</i> (with <i>mask</i>) and the watchpoint can be taken if the value is equal to, or not equal to, the constant and mask conjunction (logical AND).
	Either the <i>range</i> specifier or the <i>value</i> specifier (or both) must be included.
	The optional <i>passcount</i> specifier is used to skip a number of breakpoint occurrences.

Data and Stack Commands

The following are the commands that control data and stack.

assign <i>var</i> = <i>expr</i>	Assign the value of expression <i>expr</i> to variable <i>var</i> .
browse [<i>file</i> all]	Browse classes in global scope. If <i>file</i> is given, browse classes in scope of <i>file</i> , or if all is given, browse classes in global scope and all source files with debug information.

dis [n]	Disassemble 10 (or optionally <i>n</i>) instructions starting at the current program counter.
dis <i>addr n</i>	Disassemble <i>n</i> instructions starting at address <i>addr</i> .
dis <i>func [n]</i>	Disassemble 10 (or optionally <i>n</i>) instructions starting at the address of function <i>func</i> .
down [n]	Move down the call stack <i>n</i> levels. The default is one level).
examine <i>addr [fmt [n]]</i>	Print the contents of memory at address <i>addr</i> in hexadecimal for 4 bytes. If <i>fmt</i> is given, then print the memory in <i>fmt</i> format; if <i>n</i> is given also, print in <i>fmt</i> format <i>n</i> times. The supported formats and their corresponding default sizes are as follows:

fmt	Output	Size
b	Hexadecimal	1 byte
c	ASCII char	1 byte
d	Signed decimal	2 bytes
D	Signed decimal	4 bytes
f	Both hexadecimal & float	4 bytes, 6-digit precision
o	Octal	2 bytes
O	octal	4 bytes
s	String	null-terminated string
u	Unsigned decimal	2 bytes
U	Unsigned decimal	4 bytes
x	Hexadecimal	2 bytes
X	Hexadecimal	4 bytes

For example, the following command will print in hexadecimal, sixty-four 4-byte integers starting at address 0xac00:

examine 0xac00 X 64

locals [func]	Show definition of all local (automatic) variables in a function. If the <i>func</i> argument is not given, the current scope's function is used.
mmio [all group]	Show all or a group of MMIO registers. The supported MMIO register group names are as follows:

Name	Description
gen general	DRAM base, DRAM limit, and MMIObase
intr interrupt	Exception handler and interrupt vectors
timer	Timer register

Name	Description
hwbp	Hardware breakpoint registers
cache	I-cache and D-cache
vi	Video in
vo	Video out
ai	Audio in
ao	Audio out
pci	PCI registers
sem	SEM registers
jtag	JTAG registers
icp	ICP registers
vld	VLD registers
i2c	I2C registers
ssi	SSI registers
evo	EVO registers
xio	XIO registers
gpi	GPI registers

print *expr* | *reg*

Print the value of expression *expr* or the value of register *reg*.
 The supported register names are as follows:

Name	Equ	Meaning
\$n		n is in the range 0 to 127
\$RP	\$2	Return Pointer register
\$FP	\$3	Frame Pointer register
\$SP	\$4	Stack Pointer register
\$RV	\$5	Return Value register
\$PCSW		Program Control and Status Word

registers [group | r1 | r1 r2] Show registers of the current (stack) context. Without any argument, all registers are shown.

When a group is given, the particular group of registers is shown. The supported register group names are as follows:

Name	Meaning
a	All registers (except locals)
g	Global registers (\$0 through \$63)
l	Local registers (\$64 through \$127)
s	System registers (plus \$2 through \$8)

When a single register number is given, that register's content is shown. If a pair of register numbers are given, they specify a range of registers to be shown.

trace [f]

Display the verbose stack traceback, with register FP and SP printed as well. This **trace** command is useful when you suspect that the target program has a stack space overrun. The FP and SP register value should give a clear indication of the stack usage.

up [n]

Move up the call stack *n* levels. The default is one level).

whatis name

Print the currently visible definition (either variable or typedef) of *name*.

where

Print the stack traceback. The **set stacklevel** command limits the number of levels of traceback printed. The **set stack-verbose** command controls the verbosity of stack information printed.

where q

Quick traceback: print only function names.

where v

Verbose traceback: print function names, arguments and source line/file information.

Source File Commands

The following commands control source files.

file ["file"]

With no argument, print the name of the current file. Otherwise, change the current file scope to *file*.

files [all]

Without the **all** option, list the names of source files that were compiled with debug information. Otherwise, list the names of all source files.

list [line]

List 10 lines of the source file (starting at line number *line* if given).

list line1 line2

List source file from *line1* to *line2*.

list "file" [<i>line1</i> [<i>line2</i>]]	List the given source file from <i>line1</i> (default: line 1) to <i>line2</i> if given.
path ["dir" [:<i>dir</i>]..."]	With no argument, display current source file search path. Otherwise, set the source file search path to the given colon-separated directory list. A source file is located using pathnames in the following search order:
	<ol style="list-style-type: none"> 1. Pathname generated by the compiler. 2. The pathname with leading "/tmp_mnt" removed. 3. In the directories set by the path command.

pSOS+ Commands

The following commands control pSOS+ information.

bp <i>servicecall</i> [<i>arg</i>]	Set a breakpoint on pSOS+ <i>servicecall</i> . When an argument is passed, stop only when <i>servicecall</i> is called with its first argument <i>arg</i> .
bt [all <i>taskname</i> <i>taskID</i>]	Set a breakpoint on pSOS task scheduling. If you specify all , tmdbg stops whenever pSOS+ schedules a task. If you specify <i>taskname</i> , tmdbg stops whenever pSOS+ schedules task <i>taskname</i> . If you specify <i>taskID</i> , tmdbg stops whenever pSOS+ schedules task <i>taskID</i> .
cp	Clear pSOS+ profile information.
dp [<i>number</i> all]	Either delete pSOS+ breakpoint of number <i>number</i> or delete all breakpoints.
lp [<i>type</i>]	Show pSOS+ profile information on all objects or on objects of the specified <i>type</i> . Possible types are partition , queue , region , semaphore , and task .
profiling on off	Start/restart or stop profiling pSOS+ events.
qc	Query pSOS+ configuration.
qd	Query pSOS+ date and time.
qo [<i>type</i>]	Query all pSOS+ objects or objects of the specified <i>type</i> . Possible types are partition , queue , region , semaphore , and task .
qp [<i>partition</i>]	Query all pSOS+ partitions, or one partition with name or ID <i>partition</i> in more detail.
qq [<i>queue</i>]	Query all pSOS+ queues, or one queue with name or ID <i>queue</i> in more detail.
qr [<i>region</i>]	Query all pSOS+ regions, or one region with name or ID <i>region</i> in more detail.
qs [<i>semaphore</i>]	Query all pSOS+ semaphores, or one semaphore with name or ID <i>semaphore</i> in more detail.

qt [task]	Query all pSOS+ tasks, or one partition with name or ID task in more detail.
--------------------	--

Task-Level Debugging Commands

listtasks	List all currently active tasks. Task indices can be found by using this command.
stoptask <i>task_index</i>	Stop the execution of task <i>task_index</i> .
resumetask <i>task_index</i>	Resume the execution of task <i>task_index</i> .
focustask <i>task_index</i>	Focus on task <i>task_index</i> .

Multi-Processor Debugging Commands

focus [<i>proc_index</i>]	Focus on processor <i>proc_index</i> . If no <i>proc_index</i> is given, the currently focused processor number is shown. Valid processor can be found by using the procs command.
procs	List all installed TriMedia processors.

Miscellaneous Commands

The following commands control miscellaneous information.

args [<i>arg...</i>]	Given no arguments, print the current target program's arguments. Otherwise, set target program's arguments to the arguments specified.
chdir [<i>dir</i>]	Set the current working directory to <i>dir</i> .
dumpdp [<i>file</i>]	Print out the volatile buffer in target SDRAM that is filled by the DP macro in the target program. If a <i>file</i> argument is given, write the buffer to that <i>file</i> .
	tmdbg will look for the DP buffer by using the information that is stored in the debug section of the object file. Because the location of this buffer is stored at a fixed address per program, you can reload the same program during a debugging session after a crash has occurred and still be able to dump the buffer with the old contents (as long as no DP_START is called).
	The macro DP is defined in tmlib/dprintf.h and can be used to write into a buffer that can be dumped by tmgmon , tmmon and tmdbg .
	You must call DP with two sets of parentheses, for example, DP(("hello %s\n", name));

because when compiled with **-DNDEBUG**, DP will be compiled out. To use DP, you must initialize the internal buffer in SDRAM by calling the macro **DP_START(size, buffer_addr)** where size is the size of the buffer the user wants to use and **buffer_addr** the address of the buffer if the user wants to determine where the buffer will be. If **buffer_addr** is 0, DP will use **malloc** to allocate the buffer. **DP_STOP** will free the allocated memory.

DP uses **vfprintf** and allows all formatting of this C library function to be used. Each DP call can only print 512 bytes. In case of an overflow, the program (or task) is exited immediately, with error code FFFF03OF (**DPOverFlow**).

For backward compatibility with the Windows95-specific previous implementation, **DPsize(x)** does the same as **DP_START(x, 0)**.

echo [arg...]	Echo arguments <i>arg...</i> to stdout.
exit	Exit from tmdbg . Same as the quit command.
help [cmd]	With no argument, print a summary of all tmdbg commands; otherwise, print the command syntax for the given debugger command.
pwd	Print the current working directory path.
quit	Quit tmdbg . Same as the exit command.
script [file]	Execute commands from debugger command script file. Script files can be nested up to 64 files deep.
set	Display the current values of debugger environment variables.
set arraycolumns <i>n</i>	Set array output format to <i>n</i> columns. Default number of columns is calculated automatically based on the TTY's column width.
set arrayindex on off	on is the default setting.
set chararray array	Print array of characters in its natural format—as an array of ASCII characters.
set chararray string	Print array of characters as a null-terminated ASCII character string.
set float float	Print floating-point numbers in their natural format (%e).
set float hex	Print floating-point numbers as hexadecimal bit patterns.
set intbase dec hex oct	Set output integer base to decimal, hexadecimal, or octal.
set mode system task	Set the debug mode.
set stacklevel <i>n</i>	Set the number of stack frames for the where command to <i>n</i> .

set stackshowfpfp on off	With the on flag, each stackframe in the stack traceback is shown, by default, with its FP and SP register values. With the off flag set, the FP and SP registers will not be shown. The trace f command can be used to display the FP and SP registers even if the stackshowfpfp option is turned off.
set stackverbose on off	Set the desired verbosity of the output of the where command. off means quick traceback and on means verbose traceback that includes arguments and line information. on is the default setting.
set strlen n	Set the maximum ASCII string length to <i>n</i> . If <i>n</i> is less than 1, the string length is then set to the default length of 512.
set taskeventnotice event on off	Turn task <i>event</i> notification on or off. Supported events are: create , delete , resume , start , and suspend .
set verbose on off	Turn command verbose mode on or off . off is the default setting.
simargs	Show arguments to the simulator.
simargs arg...	Set arguments to the simulator.

Chapter 24

Code Listings

Topic	Page
x.c	110
foo.c	110
d.h	111
d1.c	112
d2.c	113

X.C

```
#include <stdio.h>
typedef int (*fun)();
extern int foo(fun f, int b);
typedef struct aap {
    int x;
    int y;
} aap;
typedef union noot {
    int x;
    char y[4];
} noot;
aap x = { 0x12345678, 2 };
static int y = -1;
static int
bar(int x){
    return x;
}
int zzz;

int main(int argc, char *argv[]){
    noot y;
    static int z = 42;
    fun f = &main;

    y.x = foo(f, bar(x.x));
    printf ("%d %d\n", x.x, x.y);
    printf ("%x %x %x %x %x\n", y.x, y.y[0], y.y[1], y.y[2], y.y[3]);
    return z;
}
```

foo.c

```
typedef int (*fun)();
int zzz;

int
foo(fun f, int b){
    return b;
}
```

d.h

```
typedef struct person {
    char      *name;
    int       age;
    int      misc[20];
    struct   person *kids;
} person;

int          foo (int, float, char, int, person *, person);
int          fool (person, int);
int          sum (int);
float        bogus(int);
custom_op float fsqrt(float);

extern  int   buffer1[], buffer2[], buffer3[];
```

d1.c

```
#include <stdio.h>
#include "d.h"

person john = { "John Doe", 25,
                 {20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1},
                 NULL
               };

person *doe = &john;

main(){
    volatile int i = 0;
    float      f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11;
    volatile float fs[12];

    for( i = 0; i < 10; i++ ){
        printf( "Sum ( %d ) = %d \n", i, sum(i) );

        f0 = rand();   fs[0] = f0;
        f1 = rand();   fs[1] = f1;
        f2 = rand();   fs[2] = f2;
        f3 = rand();   fs[3] = f3;
        f4 = rand();   fs[4] = f4;
        f5 = rand();   fs[5] = f5;
        f6 = rand();   fs[6] = f6;
        f7 = rand();   fs[7] = f7;
        f8 = rand();   fs[8] = f8;
        f9 = rand();   fs[9] = f9;
        f10 = rand();  fs[10] = f10;
        f11 = rand();  fs[11] = f11;

        printf ("bogus( %d ) = %f\n", i, bogus(i));

        buffer1[i] = i;
        buffer2[i] = sum(i);
        buffer3[i] = buffer1[i] + buffer2[i] + f11 ;
        foo ((int)f11, 2.75, 'g', (int)f11 + buffer2[i], doe, john);
        fool (john, i);
    }

    return (5871);
}
```

d2.c

```
#include <stdio.h>
#include "d.h"

int buffer1 [160]; int buffer2 [160]; int buffer3 [160];

int
sum( int i ){
    int j, sum;
    for( j = 0, sum = 0; j < i+1; j++ ) sum += j;
    return (sum);
}

volatile int dbpc = 0;

float
bogus( int i ){
    int j = 0; float f = 0.0;
    for( j = 0; j < i; j++ ){
        dbpc++;
    }
    printf (" dbpc = %d \n", dbpc);
    buffer1[0] = 1234;
    f = f + fsqrt(1.235467);
    return (f + (float) sum((int) f));
}
int
foo( int i1, float f, char c, int i2, person *p2, person p1 ){
    int j, sum = 0;
    for (j = i1; j < i2; j++) sum += (int) c * (int)f;
    p2->age = sum;
    return (p2->age);
}
int
fool( person p1, int i1 ){
    int j, sum = 0;
    for (j = 0; j < i1; j++) sum += i1;
    p1.age = sum;
    return (p1.age);
}
```

