RENESAS

IDT/sim
User/Developer's Manual

Version 2.3
February 2003

Notes

Notes

Notes

Notes

Notes

List of Tables

Notes

IDT/sim Debug Monitor Overview

Introduction

IDT System Integration Manager (IDT/sim) is a software/firmware product that facilitates convenient download of programs to IDT evaluation boards and execution as well as debug of such programs. IDT/sim is available in two forms:

◆ *Executable code programmed in EPROM or Flash: In this form, IDT/sim is present in the read-only-memory of IDT's evaluation boards and performs as the operating system, the low-level debugger, and the run-time environment from 'C'/assembler code. A command line interface over a serial port is also available for user interaction with the hardware.*

◆ *Software source code on CD-ROM or at IDT's FTP site: IDT/sim is also available as software source code written in 'C' and assembler. In this form, modifications or enhancements can be made to IDT/sim functionality through a user configurable "command table". This feature allows the user to simply link and load the enhanced functions into memory and enter the entry point into the command table. Some special entry points for supporting stand-alone systems are also available. IDT/sim source code CD-ROMs are shipped with a majority of IDT evaluation board kits. Instructions regarding obtaining source code from IDT's FTP site can be obtained by emailing rischelp@idt.com.*

What Does IDT/sim Do?

IDT/sim performs several functions. On start-up, the monitor automatically determines the cache and main memory sizes. It then performs as the operating system for the evaluation board on which it is installed. It manages the hardware resources of the board, provides the user with a command line interface; offers low-level debugging capabilities, and provides an interface for remote high-level debuggers (GDB stub).

IDT/sim is equipped with a built-in assembler, disassembler, entry points for user-defined commands, interrupt handlers, a ROM resident 'C' library, and I/O interface for a variety of devices that include serial, ethernet, PCI, I2C, SPI, and much more.

Who Should Read This Manual?

This manual addresses the needs of both *users* and *developers*. To use this manual effectively and to obtain information most relevant to your needs, it is important for you to identify the scope of your task. You are primarily a *user* of IDT/sim if you fall into one of the following categories:

◆ *do not intend to change the available functions of IDT/sim*

◆ *are not interested in how IDT evaluation boards work but are only interested in learning the operating system commands for the boards*

◆ *are interested in evaluating an IDT Integrated Communication Processor (ICP), using one of the IDT evaluation boards as a vehicle for executing benchmarks*

◆ *have already decided to use an IDT ICP for your application and are now evaluating whether one of IDT's existing evaluation boards—possibly with some modifications—will satisfy your needs or should a completely new board be designed*

◆ *want to develop system-independent application code (written in standard 'C' or assembler) to run on IDT evaluation boards while your custom designed board, based on an IDT ICP, is under development.*

You are primarily a *developer* of IDT/sim if you:

◆ *want to understand how IDT/sim works*

◆ *want to evaluate the IDT/sim source code for portability*

◆ *want to gain an understanding of IDT ICP behavior and/or IDT evaluation boards; both ICP and board documentation are good but not adequate and you want to verify documentation and experiment more*

◆ *want to use an IDT evaluation board but do not want to use all of the IDT/sim features; you want to reduce the IDT/sim code size and use the memory space made available for some other purpose*

◆ *want to modify or enhance the functions of IDT/sim in IDT evaluation boards*

◆ *want to "port" IDT/sim to your custom designed board.*

Chapters 2 through 8 of this manual contain information of interest to "developers". Chapters 9 through 14 contain references to IDT/sim commands and are intended for all IDT/sim "users".

Revision History

March 1997: Initial publication.

July 28, 1999: Version 2.0 update.

July 8, 2002: Version 2.1 update.

September, 12, 2002: Version 2.2 adds support for the RC32333 and RC32438.

February 3, 2003: In Chapter 6, revised the command table. In Chapter 9, added Flash Test under Introduction and added Test Specific to EB438 and RP355 Boards section. In Chapter 11, revised Download Program from Host to Board, Dump Registers, and Fill Memory sections. In Chapter 12, revised dr command and added fb and -n commands.

Developing IDT/sim

Introduction

Before starting any IDT/sim software development project, the *IDT/sim Release Notes* should be reviewed. A copy of the release notes is included with the IDT/sim source code and provides installation instructions, a list of bug-fixes and enhancements, build procedures, and technical assistance contact information.

IDT/sim Source Code Installation

IDT/sim source code can be installed on one of the following development platforms: Solaris 2.5.1, SunOS 5.6, and Windows 95/98/ME/2000/XP. The types of platforms supported is limited only by the platforms upon which the compiler needed to compile the IDT/sim source code is currently supported. This compiler is "Algorithmics sde4.0c" (at the time of creation of this manual) and is provided by IDT. IDT/sim source code installation is simple and will take approximately 20 minutes to complete.

> **Note:** With some modifications to the Makefiles, compilers other than those from IDT or Algorithmics may be able to build IDT/sim; however, none have been used or tested by IDT.

Code for Sun machines is supplied on a CD-ROM in tar/gzip format. The file name could be something similar to: sim1042sun.tar.gz. To install the code, (1) create a new directory on your hard disk, (2) extract the file using gzip utility, followed by the tar command, into the newly created directory.

Code for Windows is also supplied on the same CD-ROM as a zip file. The file name could be something similar to: sim1042dos.zip. To install the code, (1) create a new directory on your hard disk, (2) Unzip the file into the newly created directory. Follow the installation instructions provided in the *IDT/sim Release Notes*.

Compiler Installation

Install the C cross compiler (Algorithmics sde 4.0c, at the time of creation of this manual) on the development platform of your choice.

In a majority of the tool-chains, during the compilation process, the target microprocessor is specified by the developer with a compile-time switch (-mcpu=RC32364 or -mcpu=r4k if you are using the Algorithmics compiler). To verify that your target processor is supported by your tool-chain, review the compiler documentation.

Compiler Test Runs

To test the basic functionality of the newly installed compiler, compile one or more 'C' programs. The majority of tool-chains provide some sample code for this purpose. For example, Algorithmics Compiler has a few sample source files that can be tried immediately after installation of the compiler. All of the batch or makefiles required to compile the code are provided.

Cross-compiling Issues

The compiler you use for IDT/sim development will typically be a cross compiler. This means that the machine code generated by the compiler/assembler/linker will be for a target processor quite different from the native processor of your development platform; therefore, it is important to use the correct compiler.

During the initial phases of a project, because the commands used for native and cross compilation are very similar, a common mistake is to use the native compiler—or some part of the native tool-chain—for cross development, resulting in strange error messages during compilation. If two or more cross-compilers are installed on the same machine, complicated errors can occur.

However, in a single-user Windows environment for example, operating conditions are relatively easy to manage by creating a unique batch file that establishes the correct environment for each cross compiler. Once this has been done, any cross-compiler can be used by first running the appropriate batch file. Creation of the batch file is a one-time task usually completed after installation of a new tool-chain.

In the UNIX environment, it is the system administrator's responsibility to educate users on all compiler set-ups and to correctly install each cross-compiler so that operating environments do not conflict.

Building IDT/sim

To build a working version of IDT/sim, you will need some knowledge of IDT's source code. The actual process of building executable IDT/sim code can be as simple as running the 'make' utility on a specific "makefile." There are several 'makefiles' available that support a variety of evaluation boards and both endianness. Selection of the appropriate 'makefile' must be based on the evaluation board you will be working with.

There are several subdirectories below the root of installation level:'MAKE','S334','S355', ''S438',common', 'drivers', 'header', 'net', and 'pci'. Depending on the board you are using, change to the appropriate sub-directory under the MAKE directory and you will find makefiles and linker scripts to build IDT/sim.

After locating the correct 'makefile,' review it. Verify that the declarations made in the file (such as paths and filenames) are applicable to your installation and modify them, if needed. Any necessary changes will be made within the first few lines, so studying the entire 'makefile' is usually not necessary.

Next, run the 'make' utility and fix any errors seen during the 'make' process. At the end of a successful 'make,' depending on your evaluation board, either one or four files with the extension '.sre' or '.prm' will have been created. These are the Motorola standard S-record files that can be downloaded to your EPROM/FLASH programmer or ROM emulator. The number of S-record files created is equal to the number of IDT/sim EPROMs/FLASHes on your evaluation board.

Testing the IDT/sim Executable

Now that the executable version of IDT/sim has been created, test it by using the S-record files you created to program a set of EPROMs/FLASHes and replace the existing ones in your evaluation board[1]. When the board is powered up, the IDT/sim sign-on message should appear on the terminal connected to the first serial port on the evaluation board (port labeled TTY0 or UART A on IDT Boards).

If the sign-on message does not appear—and if your evaluation board has IDT/sim split into four EPROMs/FLASHes—try reversing the order in which you placed the EPROMs/FLASHes. The correct EPROM/FLASH order can be derived from your knowledge of (a) endianness of the evaluation board and (b) the "-b" switch parameter used in the makefile while creating the S-record files; however, it is quicker just to reverse the board's EPROM/FLASH order. If you are using a ROM emulator, reversing the pod order is even easier because it simply takes a command to the emulator.

Once the sign-on message appears, try some simple commands. Detailed knowledge of IDT/sim commands is not necessary at this point. Simply enter "help" or "?" and try commands such as "dump memory" or "run diagnostics."

If the IDT/sim created on your development platform does not function properly, review the information in this chapter and look for simple clues to errors such as:

 ◆ *Check the length of the S-record files. If they are not in tens or hundreds of Kilobytes, an error might have occurred during the process of creating the S-record files.*

 ◆ *Check the length of the executable file from which the S-record files were generated. The name of*

---

[1] To avoid programming errors for each IDT/sim revision during the development cycle, consider purchasing a ROM emulator. If you have access to one, download the newly created S-record files to the ROM emulator. With the pods plugged into the evaluation board, the IDT/sim sign-on message will appear on the terminal that is connected to the evaluation board's first serial port (port labeled TTY0).

the executable file can be found in the makefile you used to create the current version of IDT/sim. If the length of this executable file is not in tens or hundreds of Kilobytes, the file was not created properly.

◆ *Verify that all of the object files were created. Object files have an extension of '.o'. The number of object files created must match the number of files listed in the makefile. See if any of the object files have a zero length.*

Most of the errors mentioned above will be detected during the 'make' process itself; but it is possible to have not made a clean start through your several iterations of 'make.' In the early stages of your development efforts, it is a good practice to begin with a 'make clean' command, before actually running 'make' to build the IDT/sim executable. If additional assistance is needed, call the IDT RISC hotline or email rischelp@idt.com.

IDT/sim or Micromonitor

Micromonitor is provided in source code form along with the IDT/sim source code. This product is intended to assist hardware engineers in bringing up and debugging their board-level products. Micromonitor is written in assembler language and needs minimal hardware to function. If your board has a functioning CPU, a UART interface, and an EPROM/FLASH interface, you may port the Micromonitor code to your hardware and begin using it (information on using Micromonitor is available in Chapter 8).

When working with a newly designed board, it is advisable to begin by porting the Micromonitor. Once the Micromonitor is functioning properly, you may want to begin working with the IDT/sim in a "one step at a time" manner: begin with minimum sim functions and add features as hardware confidence grows. A new developer should decide early whether to begin with the Micromonitor or with IDT/sim.

When working with a modification of a working board or design, you may want to begin porting IDT/sim right away. Under most circumstances, it is advisable to begin working with a version of IDT/sim that has minimal functionality. Guidelines on how to create a stripped down version of IDT/sim and how to progressively add features to it are provided in this manual.

Minimum IDT/sim Start-up File

Notes

Overview of IDT/sim Source Files

Before you begin porting IDT/sim, a basic understanding of the source code organization will be helpful.

The IDT/sim source code that is used for all evaluation boards is organized into a single directory structure. Variations for different boards and processors are created by using different directories, aided by different 'Makefiles' and conditionally compiled source code.

There are several files common to all IDT/sim variants. Majority of the code in these files is common to all ICPs and all boards. Where there are ICP-specific and/or board specific pieces of code within these files, appropriate use of conditional compiling using "#if defined()" or "#ifdef" directives is made.

There are also files that have similar names but exist in different directories. These particular files contain code that performs similar tasks but for different target boards or processors. The implementations are so different that the simplicity of source code management achieved by conditional compiling would not justify the potential confusion while reading the code.

Finally, there are also source files unique to a single specific board. These files contain no conditional compile statements, no equivalent files in any other directory exist, and are called only for building IDT/sim for that specific board.

Directory Structure

Note the following definitions used in this manual:

RC3233x includes the RC32332, RC32333, and RC32334.

RC323xx includes the above 3 devices plus the RC32355.

RC32xxx Include all the previous devices plus the RC32438.

The common/, header/, S334, S355/, S438/, net/, pci/ and drivers/ directories are located at the top level of the IDT/sim source code installation.

common/: This subdirectory contains some "C" and "assembler" files that are common to all variants of IDT/sim. Some of these files contain conditional compile directives for different boards or ICPs. For example, "#ifdef RP355" indicates code specific to the IDT79RP355 board.

header/: This subdirectory contains header files common to all targets

S334/: This directory contains the code for IDT79RC3233x specific initialization.

S355/: This directory contains the code for the IDT79RC32355 specific initialization.

S438/: This directory contains the code for the IDT79RC32438 specific initialization.

 net/: This directory contains UDP/IP stack and ethernet drivers that are specific to IDT79RC32xxx-based evaluation boards.

 drivers/: This directory contains various tested and untested device drivers.

 pci/: This directory contains PCI initialization code for all RC32xxx-'based evaluation boards except the RC32355.

 Make/: This directory contains various board-specific Makefiles and linker script files. Depending on the target, you can change to the appropriate sub-directory and build IDT/sim images for that target.

Creating a Minimum Version of IDT/sim

Porting the entire IDT/sim to a new board can be an overwhelming task, especially if the new board is significantly different from IDT's existing evaluation boards. For example, in many cases the functions of a new board design may be similar to an existing board from IDT, but the new board's I/O interface hardware may be different from the IDT design. In some cases, functions may have been removed from IDT's design to create a new board design. In other cases, functions may have been added.

In general, it is best to begin by creating a version of IDT/sim that validates only the most basic features of the newly designed board. For example, at the start of a project, although it is not critical to port and verify the ethernet connections on the new board, it is crucial that the target CPU registers be set-up correctly. Also, although at first it is not important for the mechanism to automatically detect the size of available non-volatile memory (SRAM/SDRAM), it is very important that the memory interface be correctly programmed and fully functional in order to facilitate loading code into the memory space.

Assuming that the new board's EPROM/FLASH and Serial I/O interfaces are both working (which can be tested with the Micromonitor), a good place to begin the porting process is to identify an IDT eval board design that is closest to the new board's design and to locate the 'Makefile'[1] that is related to this IDT evaluation board.

Modifying Start-up File csu_idt.S

The first piece of code in IDT/sim is located in the start-up file called 'csu_idt.S'. This is an assembler file located in "S334/", "S355/", or "S438/" directory depending on whether the target CPU is RC3233x, RC32355, or RC32438 respectively. When a board is powered up or reset, this is the code that is executed first as it is placed at the reset vector. As one would expect, the file 'csu_idt.S' contains a majority of the code, or subroutine calls to the code, which performs the initial configuration of the target CPU and board-specific hardware devices.

Additionally, the start-up file contains code to perform many other tasks which are not necessarily required in a bare minimum IDT/sim. The following sections contain a review of the start-up files S334/csu_idt.S, S355/csu_idt.S, and S438/csu_idt.S and show how bare-minimum versions of these files can be generated.

csu_idt.S

The first subroutine in the file is *start.* At the beginning of *start,* there are a series of 128 *jump* instructions. This is a PROM entry point table, which provides entry points into IDT/sim code that pertains to many standard C language functions such as *open, close, printf, read, write, gets, puts,* etc. Some IDT-specific functions are also accessible through this jump table. This jump table is provided so that a user program running out of RAM can simply use these entry points instead of linking run-time library code, which may occupy large amounts of RAM space.

Linking C functions through the PROM entry point table will result in slower code because the entry points and the actual functions are programmed into PROM and will execute from the PROM as well. However, where compactness of user code is more important than speed, this entry point table is made available to the user. The functions that the prom entry point table points to, are not required for basic functionality of IDT/sim, and these functions may be removed or commented out after replacing the existing target for jump instructions with a "not_implemented" target for jump.

The csu idt.S file contains a number of *#ifdef* statements. A majority of these statements reflect conditional compile directives that are specific to a particular IDT evaluation board. If your board is based on one of IDT's evaluation boards, you may leave code pertaining only to that board in the csu_idt.S file and delete the code that is specific to other boards. Leaving the unrelated code is harmless, but deleting it will improve source code readability.

---

[1] The process of locating a particular "Makefile' is discussed in Chapter 2.

As an example, if your board is not based on the S334A board, you may safely delete all lines of code between and including the lines *#ifdef S334A* and the corresponding occurrence of the line *#else* or *#endif* (if no corresponding *#else* was present). If *#else* was present and you deleted it, remember to delete the corresponding *#endif*. It is critically important to delete all *corresponding* occurrences of these directives, especially in cases where there are nested *#ifdef* statements.

The next step in the start-up code deals with initializing specific registers. Not all registers are present in all IDT parts. Remove parts of the code that are not applicable to your IDT part for clarity of code.

Clear Interrupts

The next step in the sequence of initialization is to clear all the software interrupts that might have occurred before the board was reset.

Cache/Device Controller Setup

The next step is to program the cache mode. We can choose between write back cache for the RC323xx, write through cache, or uncached mode.

The device controller can now be programmed to assign various chip selects to various devices on the board. The device controller provides a glueless interface to SRAM's, ROM's, dual port memories, IO devices, and many other peripheral devices.

RAM Setup

Memory is initialized depending on the type of memory being used (e.g., SRAM/SDRAM/DDR-SDRAM). You may need to refer to the data sheet of the memory device to understand more regarding the specific steps needed to initialize the memory. Once setup is completed, a simple test is performed to see if the memory is accessible. If this test fails, it is because RAM is not properly accessible and the current version of IDT/sim simply hangs in an infinite loop (search for the string 'memory not' in the file 'csu_idt.S' for the location of this infinite loop). It may be useful to have an LED/LCD display indicate that a RAM error has occurred.

However, if this first test is passed, a second test is performed with the data pattern of -1. After it has been determined that RAM is accessible, a number of important tasks are then performed.

Subroutine 'initmem'

A stack pointer is set up and the entire stack is set to zero for IDT/sim execution. Leave this portion of the code untouched. The default size of the stack is defined in *P_STACKSIZE,* which is set to 8 Kb (16Kb if ethernet support is present) in the file 'header/idtmon.h.'

Next, a certain amount of cache configuration is performed. First the sizes of both data and instruction caches are stored in variables for future reference (*dcache_size, icache_size*).

Following cache configuration and sizing, caches are flushed by the subroutine 'flush_cache'. Leave this code untouched.

Initialize Device Table

The subroutine 'init_dev_tab' moves the I/O device handling table from ROM to RAM for faster access. By default, only two serial I/O devices, 'tty0' and 'tty1', are installed at this stage (some IDT79S334A boards also have two additional (external) serial I/O devices called 'ity0' and 'ity1'). If your board does not have the second serial I/O port 'tty1', you may delete it from the table. The table is defined as 'device_init[]' in the file 'common/idtconf.c'. It should be verified that the I/O base address declared in the table matches the address on your board. The definition of the 'init_dev_tab' function is located in the same file.

Initialize IDT/sim to Known State

To initialize some of the remaining hardware, the next step is to assign known legitimate values to some of the system variables that are used by IDT/sim. This is done in the call to the function "initialize".

A call is made to subroutine 'init_memory'. The first step in this subroutine is to invalidate the TLB. The rest of the code uses a couple of techniques to determine the size of available RAM. If you already know the size of the RAM, hard code it in the variable 'mem_size' in the subroutine 'init_memory,' which is defined in the file 'common/excepth.c'.

The next few lines, enclosed within the pair of *#ifdef INET - #endif* initialize the board's ethernet ports and PCI controller, and are not necessary for a minimal version of IDT/sim to address ethernet interface issues. Remove the define for INET from Makefile to compile out these lines.

Initialize Command Table

The call to subroutine 'init_cmd_tab' copies the command table from ROM space to RAM space for faster access. This command table is the data structure that the IDT/sim command line interpreter uses as a look-up mechanism for translating user commands into actions.

IDT/sim currently supports a large number of user commands. In the minimal version, a majority of these commands may be disabled to reduce the size of IDT/sim (Chapter 4 discusses which commands to disable). At this point, leave the call to 'init_cmd_tab' as it is.

Clear Breakpoints

Leave this call untouched. During the testing phase, even a minimal version of IDT/sim may need low level debugging facilities of IDT/sim.

Makefile

Makefiles are used to build IDT/sim monitor code. The makefiles are specific to boards. Please choose the makefile according to the board you plan to use. The makefiles contain various options specified as -D definitions.   Here is a brief explanation of the various options.

1. TARGET - Specifies the target for which you wish to build IDT/sim.
2. EIGHT_BIT_FLASH - Specifies whether 8, 16, or 32 bit FLASH is used to program IDT/sim.
3. CPU_R32364 - Includes code common to both RC32334/2 and RC32355 targets.
4. CPU_R32438 - Includes code specific to the RC32438.
5. DEFCON - Specifies the default console port.
6. MIPSEL - Specifies little endian mode for code build. Switch settings on the board must be the same.
7. INET - Specifies whether you wish to include ethernet support or not.
8. MEMCFG - Specifies the type of memory on the board.
9. DRAMSZ - Specifies the size of the nonvolatile memory.
10. EPRMPRTWD - Specifies the port width.
11. MHZ - Specifies the frequency of the crystal on the target.
12. PCIHOST - Distinguishes between PCI Host and Satellite modes.
13. NVRAM_RTC - Use RTC for storing environment variables. Otherwise, use I^{2}C-based EEPROM for environment variables.

Flowcharts

Introduction

This chapter includes some of the flow charts which will aid in understanding the IDT/sim initialization code flow. It also includes the PCI initialization flowcharts for RC3233x based boards.

Reading Flow Charts

1. When a function is in the form xyz() and is followed by a file name xyz.c, it means that the function xyz can be found in this file xyz.c. In the example shown in Figure 4.1 below, *initmem* is a function in *csu_idt*.S.

2. When a function is in the form uvw() and is followed by a file name uvw.c in Italics, it means that this function is called within the above function. In the example below, both *init_tlb_local* and *config_cache* are called within function *initmem*.

3. When a function is underlined and the block is shaded, it means it is expanded further and has a flow chart of its own.
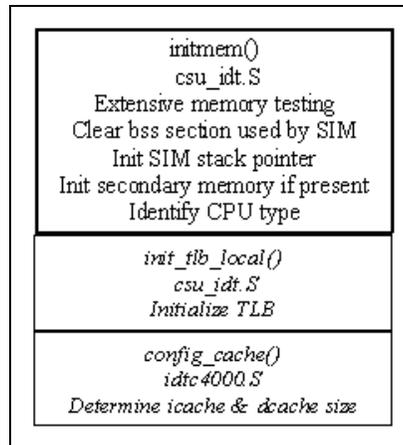
Figure 4.1 Example of IDT/sim Function
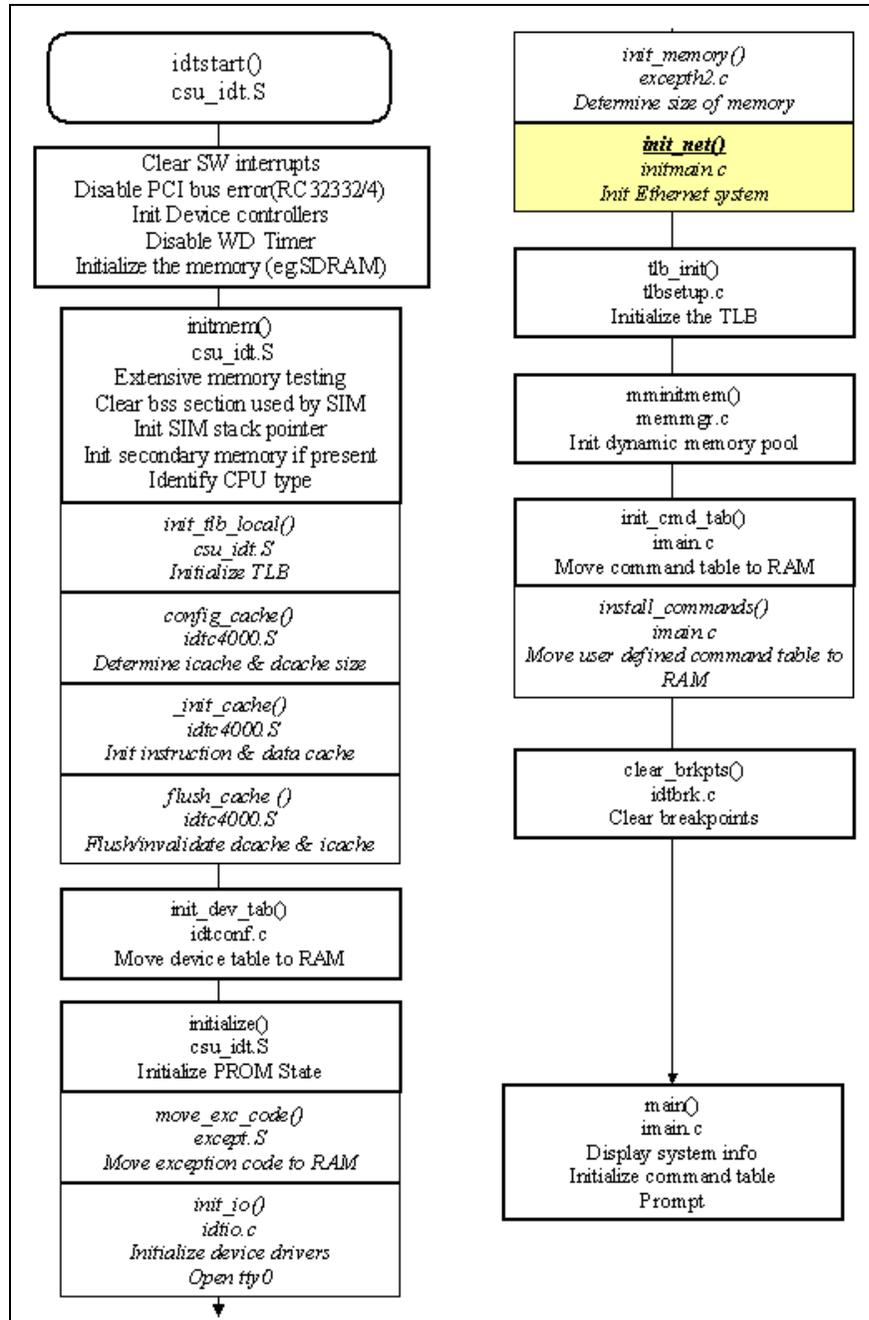
Figure 4.2 Boot Sequence of IDT/sim 10.4.2
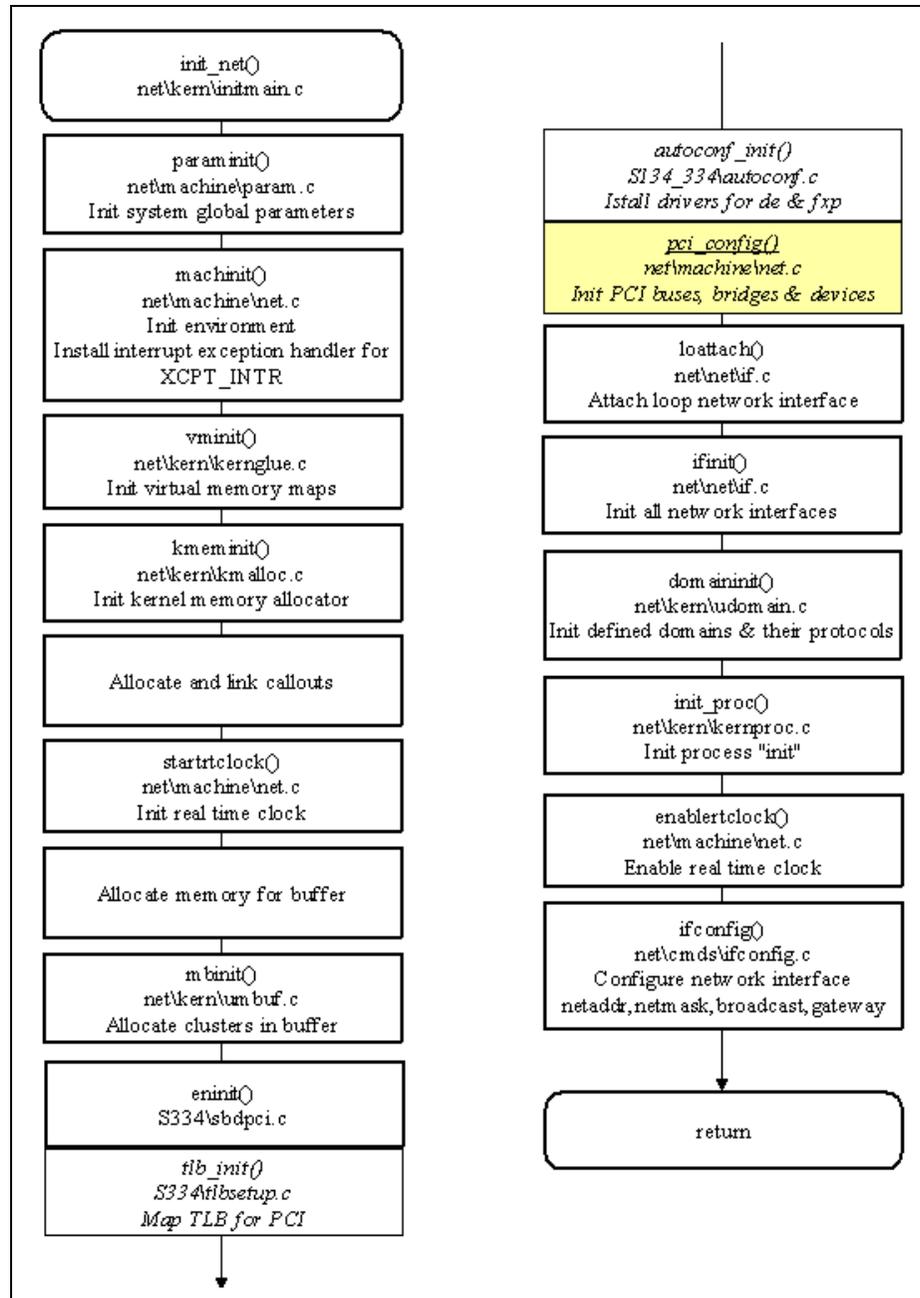
Figure 4.3 Network Interface Initialization for RC3233x Based Evaluation Boards
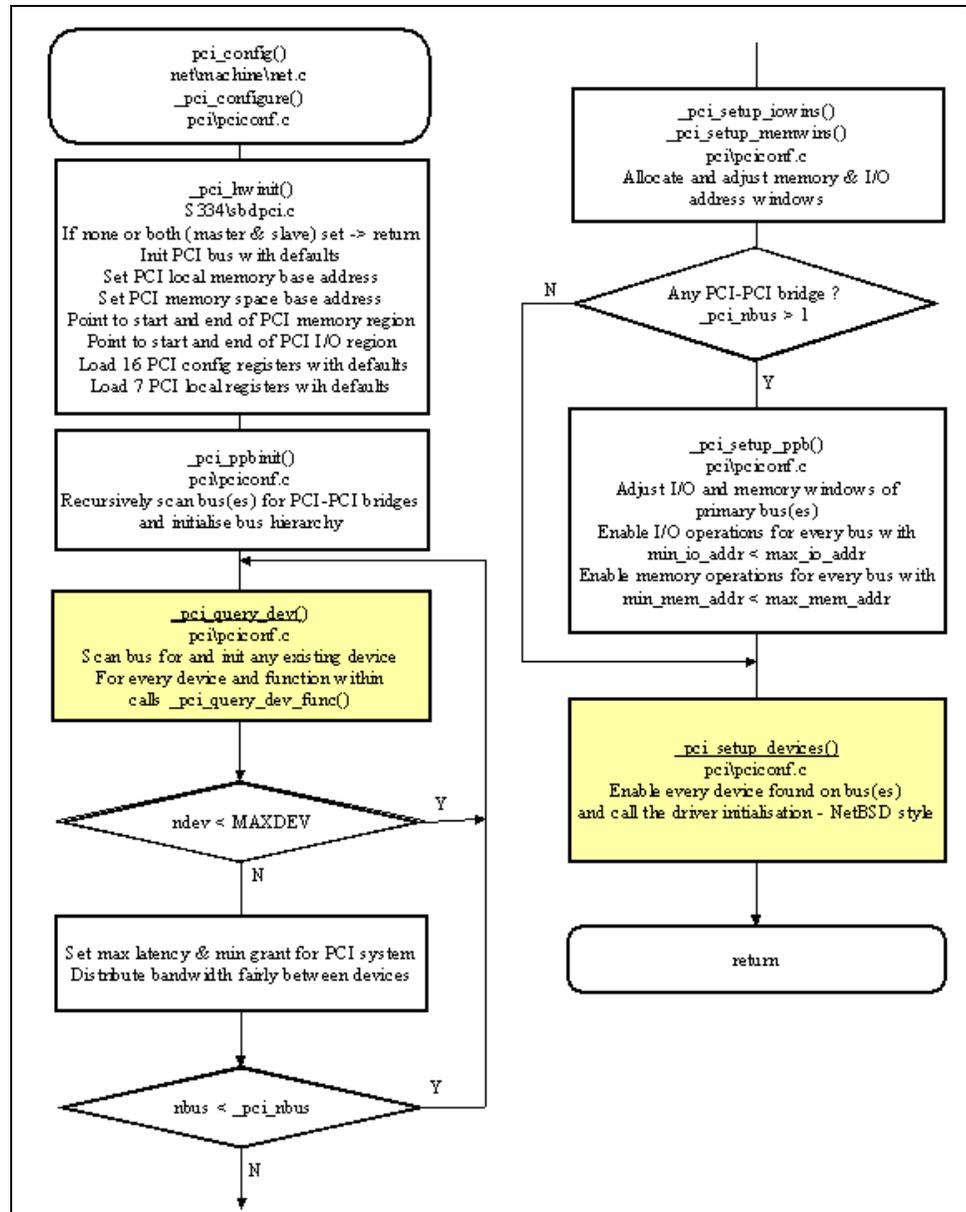
Figure 4.4 Initialization of PCI Buses, Bridges, and Devices in RC3233x Based Evaluation Boards
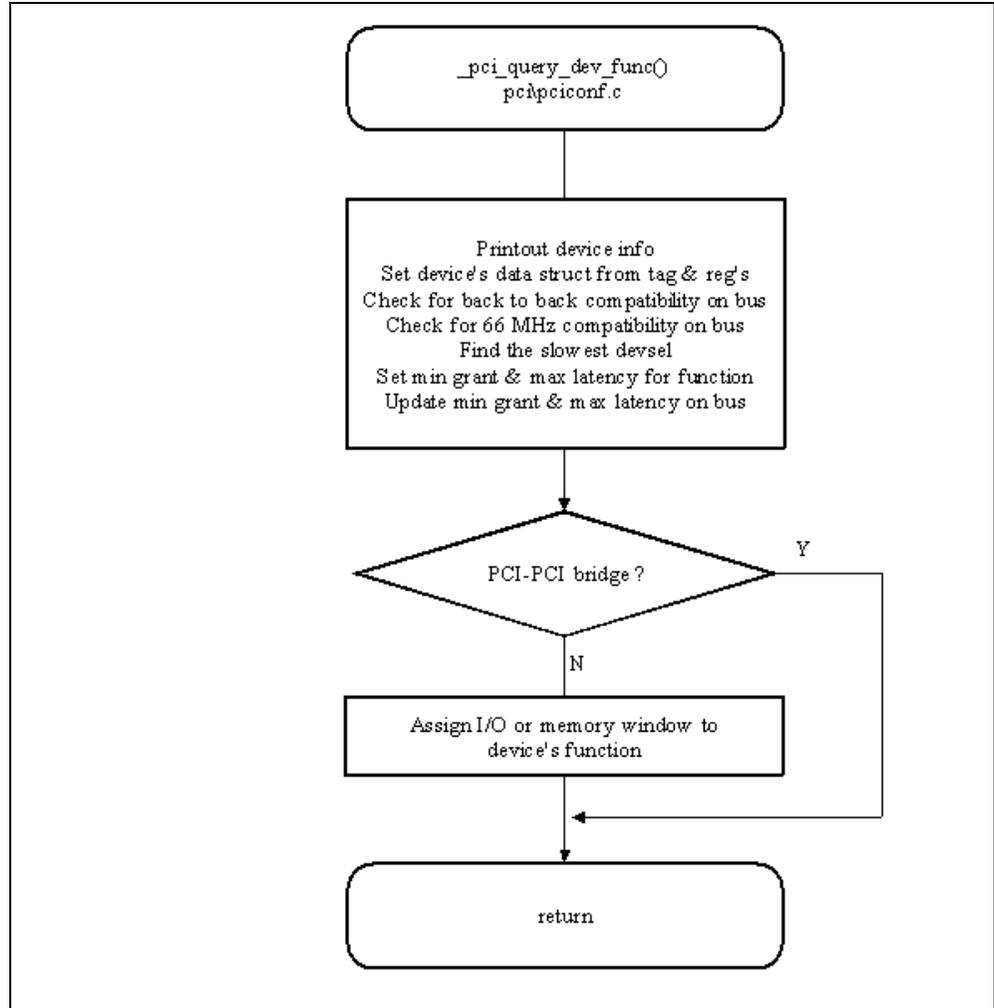
Figure 4.5 Initialization of PCI Device Functions in RC3233x Based Evaluation Boards
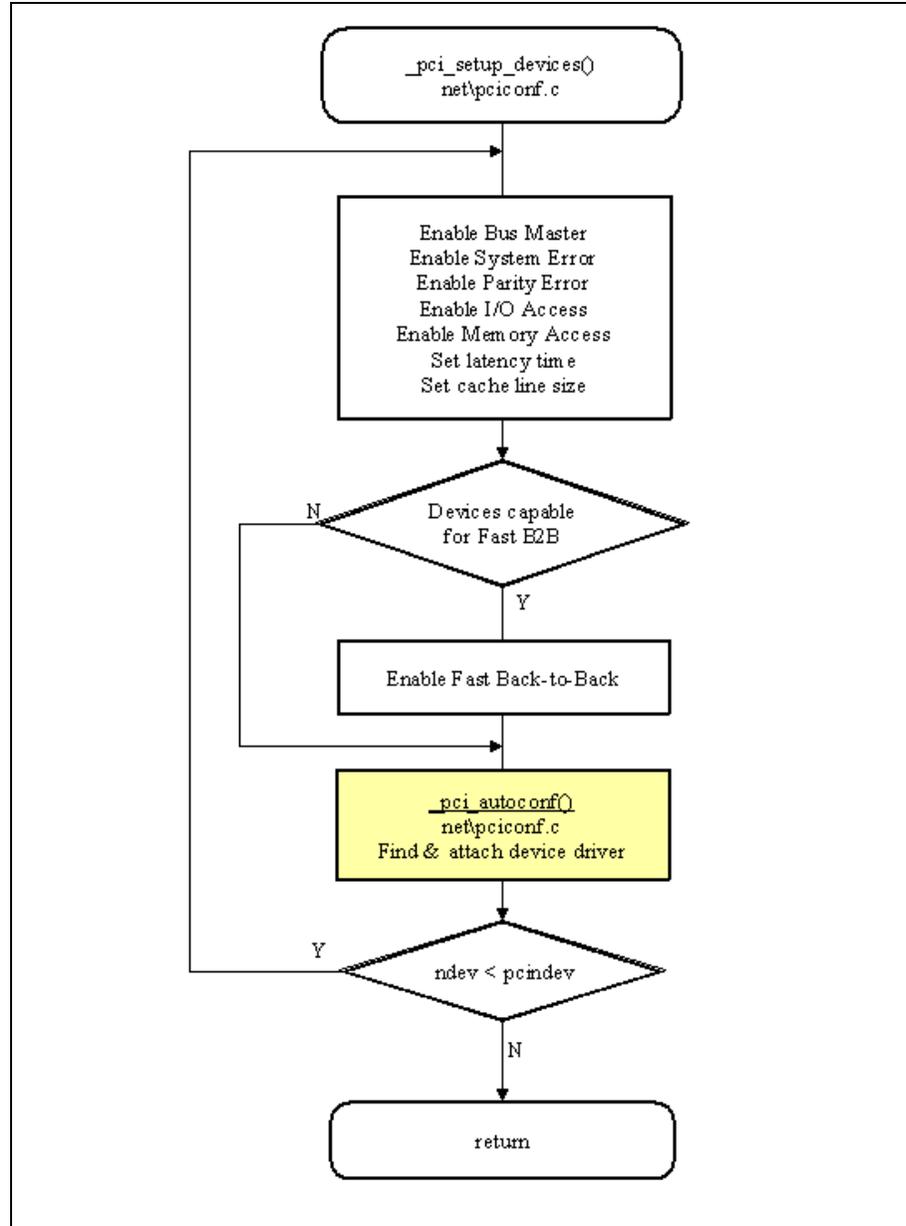
Figure 4.6 Initialization of Device Drivers in RC3233x Based Evaluation Boards
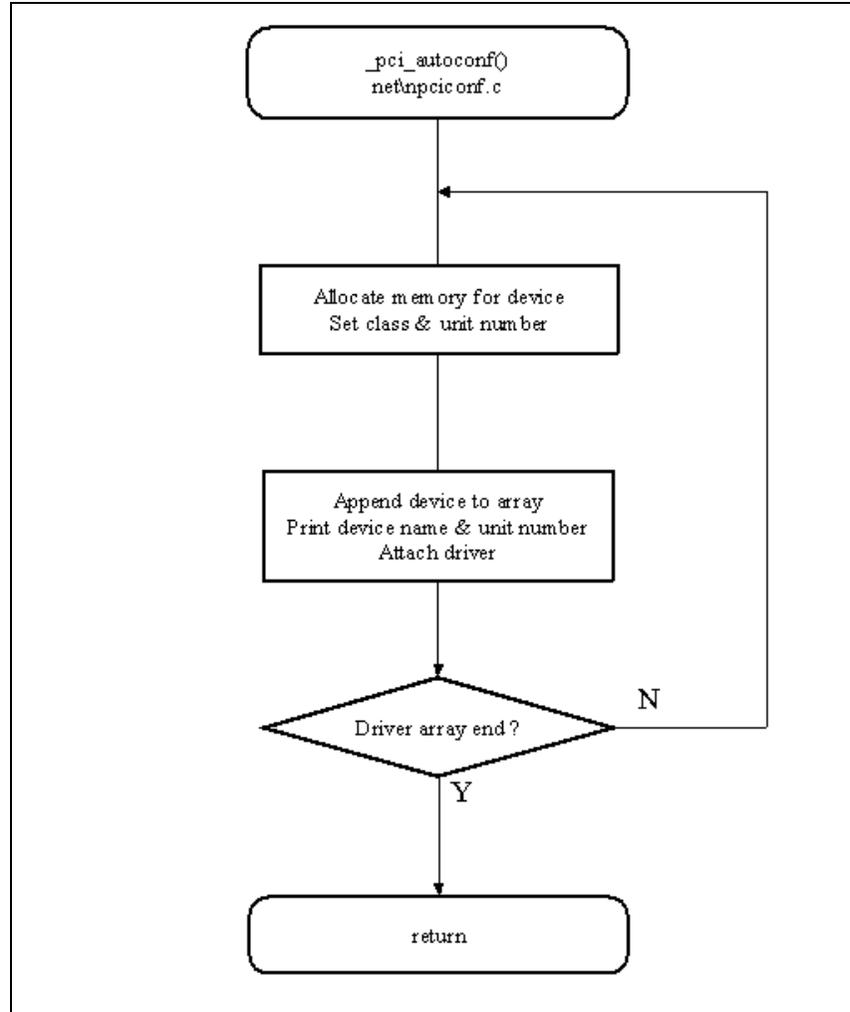
Figure 4.7 Device Driver Attachment in RC3233x Based Evaluation Boards

Minimum IDT/sim User Commands

Command Table

The size of IDT/sim code largely depends upon the number of user commands it supports. As shown earlier, 'csu_idt.S' is the start-up file first loaded when linking IDT/sim. The 'start' subroutine—explained in Chapter 3—performs all system initializations and jumps to the function 'main.'

The function 'main' is defined in the 'common/imain.c' file. When an IDT evaluation board is powered up, the function 'main' displays the sign-on message and calls a command line interpreter function named 'cli,' which accepts keyboard commands at the '<IDT>' prompt through one of the 'tty' ports.

Once a user command is received, a command table is looked up. If the command is located in the table, it is executed based on the action that is specified in the table entry for that command. The command table is defined as a structure called 'command_tab[]' in the file 'imain.c.' Begin with this table when creating a minimal version of IDT/sim.

Commands not Required for Minimal IDT/sim Versions

Table 5.1 contains a list of user commands that do not require support during the initial port of IDT/sim, and you may delete these command entries from 'command_tab[]'. In general, code that supporting a single command is contained in a single file. Consequently, once you delete a command from the table, you may also remove the line(s) that correspond to compiling and linking of this file in the related 'Makefile' used to build the minimal version of IDT/sim.

The list in the table below includes command names (alternate command names are in parentheses) in the first column and file names which support the command in the second column. There are cases where one file contains support for more than one command. In such cases, the relevant function name is also listed in the last column. Modifications to these particular files should only be done to the extent of removing these specifically listed functions.

If there is no function name listed in the third column, you may delete the entire file. Also, please do not forget to delete the 'extern' declarations of the functions that you are deleting from the file 'common/ imain.c'. For networking commands such as 'boot', where many files in 'net/' are involved, it is easier to just modify the 'Makefile' by simply removing the occurrence of the definition '-DINET' from the 'Makefile'.

Command	File Name	Function
asm	common/idtcmds1.c	asm_cmd()
boot	net/*.*	delete '-DINET' from Makefile
call (ca)	common/idtcmds2.c	callcmd()
	common/except.S	do_call()
dbgint (di)	common/idtcmds2.c	ri_sel()
debug (db)	common/idtdebug.c	
disable	common/idtfio.c	
dt	common/idtbrk.c	dump_trace()
env	common/p4000cmd.c	

Table 5.1 Commands Not Required for Minimal IDT/sim Versions (Page 1 of 2)

Command	File Name	Function
gotill (gt)	common/idtbrk.c	gotill()
history (h)	common/icli.c	linebuf[], hist_init(), history()
ping	net/*.*	delete '-DINET' from Makefile
setenv (set)	p4000cmd.c	purgeenv()
term (te)	common/idtcmds1.c	em(), to_transpar(), from_transpar()
t	common/idtbrk.c	trace_cmd()
tc	common/idtbrk.c	cond_trace_cmd()
tex	common/idtbrk.c	excl_cmd(), exc_init()
ts	common/idtbrk.c	stop_trace()
unsetenv	p4000cmd.c	

Table 5.1 Commands Not Required for Minimal IDT/sim Versions (Page 2 of 2)

Adding & Deleting User Commands

Introduction

The command table is in the source file 'common/imain.c'. All commands are decoded through this table. To add or delete a command, the user must either add or remove an entry to/from this table. When adding a command, the user will have to supply the implementation code for the command which may be in a separate file.

> **Note:** If a new file is created, the "Makefile" affected by this new file must be modified to include compiler and linker commands for this new file.

Command Table Structure

```
struct command_tab {
        char *cmdt_name;                /* command name */
        char *cmdt_abrv_str;            /* cmd. name abbreviation */
        int (*cmdt_routine)();          /* implementing function */
        int (*cmdt_init_rt)();          /* cmd. init function */
        char *cmdt_usage;               /* help string/usage */
};
```

In the above structure:

cmdt_name is a null terminated string that contains the unabbreviated command name.

cmdt_abrv_str is a null terminated string that corresponds to an abbreviated form of the command name (a second form of entering the command).

cmdt_routine is a pointer to the function that implements the command. The command line interpreter tokenizes the entire command line then places the null-terminated strings into the argv array and sets argc to the count of arguments. Delimiters used by the tokenizer are:

```
space              ' '
comma              ,
tab                \t
left parenthesis   (
right parenthesis  )
```

Each string is an argument from the command line; the command name is in argv[0]. The function pointed to by cmdt_routine in the command table is called with the following arguments:

```
cmdt_routine(argc, argv, cmd_table)
int argc; char **argv;
struct command_tab *cmd_table;
```

cmdt_init_rt - is a pointer to an initialization routine that is called on at power-up to initialize the command (such as to set up default values).

cmdt_usage - is a pointer to a null terminated string that specifies the usage or syntax of the command. In actuality, this string may be anything the user wants. It is used by the help function (see help).

Command Table Entries

Here is a listing of the complete command table currently implemented in IDT/sim.

```
static CONST struct command_tab command_tab[] = {

{ "asm",      "",    asm_cmd,   NULL, "assemble:\tasm ADDR" },

{ "benchmark",  "bm",   benchmark,  NULL, "benchmark:\tbenchmark|bm <? for help>"},
```

```
    { "brk",      "b",    brk_command,NULL, "set breakpoint:\tbrk|b [ADDRLIST]" },
#ifdef INET
    { "boot",     "",     boot_cmd,   NULL, "boot via tftp:\tboot [-n] [[HOST:]FILE]" },
#endif
    { "cacheflush", "cf",  cacheflush, NULL, "cacheflush:\tcacheflush|cf [-i|-d|-n]" },
    { "call",     "ca",   callcmd,    NULL, "call:\t\tcall|ca ADDR [ARGLIST]" },
    { "caus",     "",     intr_caus, NULL, "display Cause Register:\tcaus|caus" },
    { "checksum", "cs",   chk_sum,    NULL, "checksum:\tchecksum|cs [startaddr byte_count]" },
    { "clcs",     "clcs", intr_clcs, NULL, "clear Cause Register:\tclcs|clcs" },
    { "clsr",     "clsr", intr_clsr, NULL, "clear Status Register:\tclsr|clsr" },
    { "compare",  "cp",   compare_cmd,NULL, "compare memory:\tcompare|cp [-w|-h|-b] RANGE desti-
nation" },
    { "cmpr",     "",     tmr_cmpr,  NULL, "compare register (timer) test:\t\tcmpr|cmpr" },
    { "cont",     "c",    cont,      NULL, "continue:\tcont|c" },
#if  MEMCFG  ==  SDRAM_ONLY  ||  MEMCFG  ==  SRAM_N_SDRAM  ||  MEMCFG  ==
SDRAM_N_SRAM
    { "creg",     "creg", dram_creg,  NULL, "modify SDRAM ctrl reg:\t\tcreg|creg" },
#endif
#if defined(EB332) || (defined(S355) && !defined(RP355) && !defined(RP351)) || defined(EB438)
    { "date",     "",     com_date,   NULL, "get/set date:\tdate [mm/dd/[yy]yy]"},
    { "time",     "",     com_time,   NULL, "get/set time:\ttime [hh:mm[:ss]]"},
#endif
    { "env",      "",     com_env,    NULL, "env display:\tenv"},
#if !defined(NO_NVRAM) && !defined(PRETEND_NVRAM)
    { "setenv",      "set",  com_setenv, NULL, "set or display variable in NVRAM:\tsetenv|set [VAR
[VALUE]]"},
    { "unsetenv",   "unset",com_unsetenv,NULL,"remove variable from NVRAM:\tunsetenv|unset VAR"},
    { "purgeenv",   "purge",com_purgeenv,NULL,"purge NVRAM:\tpurgeenv|purge"},
#endif
    { "dbgint",    "di",  ri_sel,     NULL, "debug int.:\tdbgint|di [-e|-d][DEVICE|Int. Line]"},
    { "debug",     "db",  idebug,     NULL, "debug (remote):\tdebug|db [DEVICE]"},
    { "diag",      "dg",  do_diag,    NULL, "diagnostics:\tdiag|dg"},
    { "dis",       "",    _disass,    NULL, "disassemble:\tdis [RANGE]" },
    { "dc",        "",    disp_tag,   NULL, "dump cache:\tdc [-i|-d] RANGE"},
    { "dr",        "",    dump_regs,  NULL, "dump regs:\tdr [reg#|reg_name|reg_group|-u]"},
    { "dt",        "",    dump_trace, NULL, "dump trace:\tdt"},
#if __mips > 2
    { "dump",      "d",   dump,       NULL, "dump:\t\tdump|d [-d|-w|-h|-b] [RANGE]" },
#else
    { "dump",      "d",   dump,       NULL, "dump:\t\tdump|d [-w|-h|-b] [RANGE]" },
```

```
#endif
#if __mips > 2
  { "fill",     "f",    fill,      NULL, "fill:\t\tfill|f [-d|-w|-h|-b|-l|-r] RANGE [value_list]"},
#else
  { "fill",     "f",    fill,      NULL, "fill:\t\tfill|f [-w|-h|-b|-l|-r] RANGE [value_list]"},
#endif
#if defined(FLASH)
  { "fb",        "",     flashburn,  NULL, "flash burn:\tfb [[-f flash] [-e entry]] [-n] [[HOST:]FILE]"},
#endif
  { "fr",        "",     fill_reg,   NULL, "fill regs:\tfr <reg#|reg_name><value>"},
  { "go",       "g",   go,         NULL, "go:\t\tgo [-n] [INITIAL_PC]" },
  { "gotill",   "gt",  gotill,     NULL, "go untill:\tgotill|gt <brk addr>" },
#ifdef GREEN
  { "ghd",       "",     ghdebug,    NULL, "greenhills debugger:\tghd -G -B <baud-rate> [DEVICE]" },
#endif
  { "gmsk",       "gmsk", intr_gmsk,  NULL, "get status register mask bits:\tgmsk|gmsk" },
  { "help",     "?",    help,       NULL, "help:\t\thelp|? [COMMAND(S)]" },
  { "history",  "h",    history,    hist_init, "history:\thistory|h" },
  { "init",       "i",   prominit,   NULL, "initialize:\tinit|i" },
#ifdef INET
#if defined(FLASH)
  { "load",      "l",   load,       NULL, "download code:\t\tload|l [-t][-b][-s][-a][-n] DEVICE|FILE" },
#else
  { "load",      "l",   load,       NULL, "download code:\t\tload|l [-t][-b][-s][-a] DEVICE|FILE" },
#endif
#else
#if defined(FLASH)
  { "load",      "l",   load,       NULL, "download code:\t\tload|l [-b][-s][-a][-n] DEVICE|FILE" },
#else
  { "load",      "l",   load,       NULL, "download code:\t\tload|l [-b][-s][-a] DEVICE" },
#endif
#endif
  { "move",      "m",   movecmd,    NULL, "move:\t\tmove memory|m [-w|-h|-b] RANGE destination" },
  { "next",      "n",   next,       NULL, "next:\t\tnext|n [COUNT]" },
#ifdef INET
  { "ping",      "",     ping_cmd,   NULL, "ping net host:\tping [-Rdnqrv] [-c count] [-i wait] [-s size] HOST"
},
#endif
  { "rad",        "",     select_base, s_b_init, "select radix:\trad [-o|-d|-h]"},
```

```
    { "regsel",    "rs",   rs_sel,    NULL, "reg set select:\tregsel|rs [-c|-h]"},

    { "setbaud",   "sb",   setbaud,   NULL, "set baud rate of serial port:\tsetbaud|sb [CHAR_DEVICE]" },

    { "search",      "sr",    search_cmd, NULL, "search in memory:\t\tsearch|sr [-w|-h|-b] RANGE value
[MSK]" },

    { "seg",       "",     select_seg, s_s_init, "select segment:\tseg [-0|-1|-s|-3|-u]"},

    { "smsk",      "",      intr_smsk,  NULL, "set mask and enable bits in Status Register:\tsmsk|smsk" },

#ifdef VALID

#if MEMCFG != SRAM_ONLY

    { "basic",     "",      dram_basic, NULL, "basic:\t\tbasic|basic" },

    { "page",      "",      dram_page,  NULL, "page:\t\tpage|page" },

    { "freq",      "",      dram_freq,  NULL, "freq:\t\tfreq|freq" },

#if MEMCFG == SDRAM_ONLY || MEMCFG == SRAM_N_SDRAM || MEMCFG ==
SDRAM_N_SRAM

    { "late",      "",      dram_late,  NULL, "late:\t\tlate|late" },

    { "toggle",    "",      dram_toggle,NULL, "toggle:\t\ttoggle|toggle" },

    { "probe",     "",      dram_probe, NULL, "probe:\t\tprobe|probe" },

    { "fast",      "",      dram_fast,  NULL, "fast:\t\tfast|fast" },

#endif /* MEMCFG == SDRAM_ONLY || MEMCFG == SRAM_N_SDRAM || MEMCFG ==
SDRAM_N_SRAM */

#endif /* MEMCFG != SRAM_ONLY */

    { "m2io",      "",      dma_m2io,   NULL, "m2io:\t\tm2io|m2io" },

    { "seti",      "",      intr_seti,  NULL, "seti:\t\tseti|seti" },

    { "wrtst",     "",      iuart_wrtst,NULL, "wrtst:\t\twrtst|wrtst" },

#if DEFCON == EXTU

    { "intlp",     "",      iuart_intlp,NULL, "intlp:\t\tintlp|intlp" },

#endif

    { "cross",     "",      iuart_cross,NULL, "cross:\t\tcross|cross" },

    { "echo",      "",      iuart_echo, NULL, "echo:\t\techo|echo" },

    { "fifo",      "",      iuart_fifo, NULL, "fifo:\t\tfifo|fifo" },

    { "flip",      "",      iuart_flip, NULL, "flip:\t\tflip|flip" },

    { "sstr",      "",      dma_sstr,   NULL, "sstr:\t\tsstr|sstr" },

    { "mstr",      "",      dma_mstr,   NULL, "mstr:\t\tmstr|mstr [DMA_TST_TYPE]" },

#endif

    { "step",      "s",    single_step,NULL, "single step debugger:\t\tstep|s [COUNT]" },

    { "sub",       "",     subst,     NULL, "substitute memory:\t\tsub [-w|-h|-b|-l|-r] ADDRESS"},

    { "term",      "te",   em,        NULL, "terminal emul.:\tterm|te"},

    { "tlbdump",   "td",  tlbdump,    NULL, "display TLB:\ttlbdump|td [RANGE]" },

    { "tlbflush",  "tf",  tlbflush,  NULL, "flush TLB:\ttlbflush|tf [RANGE]" },

    { "tlbmap",      "tm",  tlbmap,     NULL, "tlbmap:\t\ttlbmap|tm [-i INX] [-(v/d/g)[01]] [-p PAGESIZE] [-c
CACHEALG] VADDR PADDR [PADDR]" },
```

```
    { "tlbpid",    "ti",   tlbpid,    NULL, "tlbpid:\t\ttlbpid|ti [PID]" },

    { "tlbptov",   "tp",   tlbptov,   NULL, "tlbptov:\ttlbptov|tp ADDR" },

    { "t",         "",     trace_cmd , NULL, "trace:\t\tt [-a/-o/-e/-d/-r RANGE/-w RANGE/-c RANGE/-i INS/-m
MSK]" },

    { "tc",        "",     cond_trace_cmd,NULL,"trace cond.:\ttc [-e BPNUM][-d BPNUM]"},

    { "tex",       "",     excl_cmd,   exc_init,"trace exclude:\ttex [RANGE]"},

    { "ts",        "",     stop_trace, NULL, "trc stop cond.:\tts [-b/-f/-o/-r RANGE/-w RANGE/-i INS/-m MSK]"},

    { "unbrk",     "ub",   unbrk,      NULL,"remove breakpoints:\t\tunbrk|ub BPNUMLIST|ALL" },

    { 0, 0, 0, 0, "" }

};
```

Adding & Deleting IDT/sim Device Drivers

Notes

Introduction

The tables that link the file functions to the hardware device driver are contained in the source module **common/idtconf.c**. The file functions are: *open, close, read, write, ioctl and strategy*. There are two tables: the Device Switch Table and the Device Initialization Table. To add a device driver, a new entry must be made in the 'device switch table'.

A single device driver may control several devices. For example, a driver named 'tty' controls two devices named 'tty0' and 'tty1.' There must be an entry for each device in the Device Initialization Table. Currently, IDT/sim is set up for a maximum of 8 device drivers and 16 devices. This is arbitrary and may be changed by the developer. To reduce the number of drivers and devices, change the sizes of arrays 'work_dev_tab[]' and 'work_init_tab[]' in the module common/idtconf.c.

> **Note:** There is no protection for overrunning the Device Switch Table and Device Initialization Table. Developers must not install more than 8 drivers and 16 devices. If more than 8 and 16 are required, the arrays 'work_dev_tab[]' and 'work_init_tab[]' must be expanded.

Device Switch Table

The format of Device Switch Table is shown below:

```
struct dev_sw_tab {
    int (*d_open)();              /* open routine */
    int (*d_close)();             /* close routine */
    int (*d_read)();              /* read routine */
    int (*d_write)();             /* write routine */
    int (*d_init)();              /* initialization routine */
    int (*d_strategy)();          /* io strategy routine */
    int (*d_ioctl)();             /* io control routine */
    char *d_driver_name;          /* pointer to driver name */
};
```

Each of the members in the above structure points to implementation routines that are supplied by the device driver. Currently, there are only two device drivers built into IDT/sim. These are the 'tty' and the 'ity' (only for the S334/S355 boards) drivers, both of which are serial I/O drivers. Source code for these drivers can be found in one of the subdirectories of "drivers/" directory, based on the serial I/O device used on the specific board under consideration. If the device that the driver is written for does not require a particular function (e.g. *strategy*), then that member should contain a pointer to a routine that does nothing (see example below for 'null device' routine). There is one entry in the Device Switch Table for each driver in the system.

Device Initialization Table

The structure shown below is an entry in the Device Initialization Table.

```
struct dev_init_tab {
    char *dev_name;        /* device name */
    char *dev_descrip;     /* device description */
    char *dev_drv_name;    /* driver name */
    int dev_cntl;          /* device controller number */
    int  dev_unit;         /* unit number */
    int  dev_part;         /* partition number */
    int  dev_io_addr;      /* device I/O base address */
};
```

The tables currently present in IDT/sim are shown below. The Device Switch Table must be terminated with a null entry. For the 'tty' driver, the implemented functions are: *open, read, write, init, and ioctl*. There is no hardware action necessary to *close* the 'tty' devices, so this entry in the table points to the *nulldev*. There is also no need for a *strategy* routine.

Actual Device Switch Table

```
CONST struct dev_sw_tab device_table[] = {
    {
    ttyopen,   /* device open routine */
    nulldev,   /* device close routine */
    ttyread,   /* devive read routine */
    ttywrite,  /* device write routine */
    ttyinit,   /* device init routine */
    nulldev,   /* device strategy routine */
    ttyioctl,  /* device ioctl routine */
    "tty"      /* device driver name */
    },
#if defined(S134) || defined(S355)
    {
    ityopen,   /* device open routine */
    nulldev,   /* device close routine */
    ityread,   /* devive read routine */
    itywrite,  /* device write routine */
    ityinit,   /* device init routine */
    nulldev,   /* device strategy routine */
    ityioctl,  /* device ioctl routine */
    "ity"      /* device driver name */
    },
#endif
    { 0,0,0,0,0,0,0,0 } /* null entry to mark end of table */
};
```

In the above table, the entry nulldev points to a routine that just returns zero.

```
 int
nulldev()
{
    return(0);
}
```

The Device Initialization Table has an entry for each device in the system. IDT/sim supports both channels of DUART so there are two entries in this table, the first for 'tty0' and the second for 'tty1.' The same is true for ity0 and ity1 in the 79S134/S355 board.

Actual Device Initialization Table

```
CONST struct dev_init_tab device_init[] = {
    {
    "tty0",         /* name of device */
    "console",                      /* dev. description */
```

```
        "tty",          /* driver name */
        0,              /* controller number */
        0,              /* unit number */
        0,              /* partition */
        0x1fe00000      /* io base address */
        },
        {
        "tty1",         /* name of device */
        "",             /* dev description */
        "tty",          /* driver name */
        0,              /* controller number */
        1,              /* unit number */
        0,              /* partition number */
        0x1fe00000      /* io base address */
        },
#if defined(S334) || defined(S355)
        {
        "ity0",         /* name of device */
        "",             /* dev. description */
        "ity",          /* driver name */
        0,              /* controller number */
        0,              /* unit number */
        0,              /* partition */
        0x18000800      /* io base address */
        },
        {
        "ity1",         /* name of device */
        "",             /* dev description */
        "ity",          /* driver name */
        0,              /* controller number */
        1,              /* unit number */
        0,              /* partition number */
        0x18000820      /* io base address */
        },
#endif
        { 0,0,0,0,0,0,0 }
        };                  /* end device_init */
```

The device description for 'tty0' is console. Other devices have no description. The names of drivers are 'tty' and 'ity' and provide the link between device name and driver. These drivers do not embody the concept of controller, so all of these devices are assigned to controller zero. 'tty0', for example, is unit zero (0) and tty1 is unit one (1). There is no partition. The I/O in the MIPS architecture is memory mapped, and the final entry points to the base I/O address for the DUART.

Using Micromonitor

Notes

Introduction

Micromonitor is a small monitor written in assembly language for hardware engineers to begin debugging hardware design features such as DRAM and other I/O devices. The CPU, the EPROM and the UART are the only hardware parts required to run the Micromonitor.

The monitor has two small stacks of three words each, an 'operand' stack and a 'return' stack. These stacks are stored in registers. The operand stack is used by the monitor commands. For example, all 'load's and 'store's are between the stack and memory. The return stack is used by the routines that implement the commands.

All execution control instructions used in the monitor are relative branches. In this way, the code is position independent and can be moved to any location and executed using the 'transfer' and 'jump' commands.

Using Micromonitor

At the Micromonitor prompt ('$'), users can enter only two types of entities, hexadecimal numbers or commands in lower case. All numbers must begin with a digit between 0 and 9; therefore, hex numbers that start with 'a' through 'f' must be prefixed by the digit 0. When a number is terminated by an <Enter> key, it is pushed on the stack. There is no limit to the number of digits entered, only the last eight before the return and after the prompt '|' for 32-bits and the last 16 for 64-bits will be parsed by the Micromonitor.

Commands are entered as single lower case hex digits. As soon as the monitor recognizes the command, it will perform the function. It will only echo characters that are appropriate at that point; all others will be ignored.

There is a default segment address that is ORed into address parameters of all commands that reference memory. This makes it unnecessary for users to type entire addresses and prevents UTLB exceptions. The default segment address can be changed by using the 'segment' command.

Stack Before	s#	Enter Command	Stack After	s#
xxxxxx	s2	0f10000 <Enter>	0f10000	s2
yyyyyy	s1		xxxxxx	s1

User Commands

Store

This command performs word, half word, and byte stores. It stores the value that is on top of the stack to the memory address that is in the slot next to the top of the stack. When the store command is executed, the address of the store—incremented by the size of the store—becomes the value on the top of the stack.

The monitor performs one store of the appropriate size (sw, sh, or sb). For example, to store the value "1234" at location "0f10000":

Stack Before	
xxxxxx	S2
yyyyyy	S1

User enters:

0f10000 <Enter>

1234 <Enter>

sw

Stack During	
1234	S2
f10000	S1

Stack After	
f10004	S2
f10000	S1

Further values can be stored simply by entering a number <Enter> and 'sw' without entering a new address because the incremented address is left on the stack. The following commands store the respective sized values:

sb -Store a byte

sh -Store a half word

sw -Store a word

Load

The 'load' command performs word, half word, and byte 'load's. It loads from the location pointed to by the contents of the top of the stack. When the load command is executed, the address incremented by the size of the datum loaded is left on the stack. The value is displayed on the terminal but not placed on the stack. The monitor uses the load opcode for the appropriate size (lw, lh, or lb). For example, to view the value at location "0f10000":

Stack Before	
xxxxxx	S2
yyyyyy	S1

User enters:

0f10000 <Enter>

lw

Stack During	
f10000	S2
xxxxxx	S1

Stack After	
f10004	S2
xxxxxx	S1

Further values can be viewed simply by entering 'lw' without entering a new address because the incremented address is left on the stack.

The following commands load the respective sized values:

lb   - Load a byte

lh   - Load a half word

lw   - Load a word

Jump

The 'jump' command branches execution to the address on the top of the stack. To transfer control to location '1000', enter:

1000 <Enter>

j

Dump

The 'dump' command displays a range of memory locations in word format on the terminal. To use the command, first enter the starting address and then the end address. A ^s or 's' will stop the display and the monitor will prompt to continue or quit. Here is a typical example which will display all locations from 0x1000 to 0x2000:

1000 <Enter>

2000 <Enter>

d

The addresses are modified to the default segment address and forced on word boundaries.

Fill

The 'fill' command stores a word data pattern in a range of word memory locations. A typical example follows which will fill all locations from 0x1000 to 0x2000 with the value 0x12345678:

1000 <Enter>

2000 <Enter>

12345678 <Enter>

f

The addresses are modified to the default segment address and forced on word boundaries.

Increment Fill

The 'increment fill' is similar to the 'fill' command but increments the filled data by '0x1' for every sequential location. A typical example follows which will fill all locations from 0x1000 to 0x2000 starting with the value 0x12345678 and incrementing the next location filled to 0x12345679 and so on.

1000 <Enter>

2000 <Enter>

12345678 <Enter>

iw

The following commands fill respective sized values and increment respective sized address:

ib   - Fill a byte, increment the byte, point to the next byte address and fill again and so on.

> ih   - Fill a half-word, increment the half-word, point to the next half-word address and fill again and so on.
>
> iw   - Fill a word, increment the word, point to the next word address and fill again and so on.

Compare

The 'compare' command compares a byte/half-word/word data pattern with all data in a range of memory locations. A typical example follows which will compare all locations from 0x1000 to 0x2000 with the value 0x12345678:

> 1000 <Enter>
>
> 2000 <Enter>
>
> 12345678 <Enter>
>
> c

If the value 0x12345678 is not present in any one of the word wide memory locations, the monitor will display the address and the value found at that address. This command can be used to verify a fill. Another use is to clear all of memory (with fill), do one store, and then check to see that one and only one location was modified. The addresses on the stack are modified to conform to the default segment address and forced to be on word boundaries.

Increment Compare

The 'increment compare' command compares a incrementing data pattern with the bytes/half-words/words in a range of memory locations. A typical example follows which will compare all locations from 0x1000 to 0x2000. The first location 0x1000 will be compared with the value 0x12345678 and the next location 0x1004 with 0x12345679 and so on:

> 1000 <Enter>
>
> 2000 <Enter>
>
> 12345678 <Enter>
>
> nw

If the incremental pattern is not present in any one of the word wide memory locations, the monitor will display the address and the value found at that address. This command can be used to verify a increment fill. The addresses on the stack are modified to conform to the default segment address and forced to be on word boundaries. The possible commands are nb (byte-wide), nh (half-word-wide), and nw (word wide).

Transfer

The transfer command moves a block of words from the range specified on the stack to a destination pointed to by an address on the stack. A typical example follows which will transfer contents of all locations in the range of 0x1000 and 0x2000 to locations starting at 0x3000:

> 1000 <Enter>
>
> 2000 <Enter>
>
> 3000 <Enter>
>
> t

This command can be used to move the monitor itself from one location in memory to another location in memory such that executing from different parts of memory can be tested. Further testing can be accomplished with the monitor memory test when the monitor is running from within Kseg0.

Scope Loops

The 'scope loop' commands transfer control to tight 2-instruction loops which are composed of a 'load' or a 'store' instruction and a 'branch.' The scope loop can read or write a word, half word, or byte as specified by the command. For example, to loop on reading bytes from the location '1000' type the following:

> 1000 <Enter>

    rb

To write word value 12345678 to the location '1000' type the following:

    12345678 <Enter>

    1000 <Enter>

    wb

The following is a summary of the available scope loop commands:

    rb   - Load a byte and loop

    rh   - Load a half word and loop

    rw   - Load a word and loop

    wb   - Store a byte and loop

    wh   - Store a half word and loop

    ww - Store a word and loop

Set Segment Default

The 'segment' commands set the default segment address which is ORed into the addresses used to access memory. This avoids having to type the full address (8 digits) and having unexpected UTLB exceptions. The following two commands are used to set the default segment address:

    k0   - Set default to 0x80000000

    k1   - Set default to 0xa0000000

Print Stack

The stack can be viewed by entering '.' This will show the contents of the top of the stack first and the next-to-top value last on the line.

Memory Check

There are two basic memory checks that can be performed. The first check writes to each location and reads/compares it before continuing to the next location.   This identifies faults immediately, which is very useful when combined with a logic analyzer to locate and observe refresh arbitration faults. The command is implemented using word-wide loads and stores only. Error reporting can be turned off with the 'mq' command. The memory check command is invoked by entering the following (start value first, termination value last):

    1000 <Enter>

    2000 <Enter>

    x

Alternating '-' and '|' will be displayed on the terminal to indicate each pass through the address range.

The second memory check is more effective at testing things, such as address line validity. In this test, a pattern of ascending values is first written to memory across the entire specified address range. Then and only then, the monitor reads/compares the range to verify that the values are present. The range is then written to again, but starting with the next larger value such that all locations will eventually receive all values possible. This command is invoked to check byte memory by entering the following (start value first, termination value last):

    1000 <Enter>

    2000 <Enter>

    mb

The following is a summary of the available memory check commands which write the entire address range first before read/comparing:

    mb   - Check memory as bytes

    mh   - Check memory as half words

mw - Check memory as words

mq - Turns off error reporting

Both memory checks can be terminated by typing ^s or 's', at which point the monitor prompts the user to either continue or terminate the memory check.

Porting to New Hardware

The hardware designer must verify that the EPROM and the UART work, before using the Micromonitor. Next the UART code must be modified to match the addressing and control of the UART. There are four routines in the monitor that involve the UART, as shown below in Table 8.1:

init_uart	Initializes the UART
conin	Returns a character from the console in v0
constat	Returns non-zero if character entered at keyboard
conout	Outputs a character in the register a0 to the terminal
echo_test	Endless loop to test console by echoing whatever is typed on keyboard

Table 8.1 Micromonitor's UART Functions

These routines can be found at the end of the Micromonitor source. It is recommended that a stand-alone copy of these routines be assembled into a set of EPROMS and tested separately before using the monitor.

All temporary variables are kept in registers. There are two stacks implemented using these registers: the 'operand' stack and 'return' stack. The operand stack uses registers *S2, S1,* and *S7*. The stack is operated on and used by the monitor commands.

For example, all loads and stores are between the stack and memory. The return stack is used by code which implements the commands. Registers *k0* and *k1* along with *ra* are used as the return stack. In this way, there can be two levels of subroutine calling. Macros ('save' & 'return') are provided to facilitate the handling of the return stack.

The code before the 'start:' and 'memerror:' labels and at the end of the file is used to get the addresses of these locations at run time. The positional relationship of the code before the labels and the label must be maintained.

Recommended Debug Technique

When using the Micromonitor to debug a fresh design, the following sequence is recommended to catch various memory system errors before moving onto the IDT/sim monitor:

1. Use store word, byte, and halfword with the **dump** command to test the simple ability of the memory system to store data.
2. If a fundamental error occurs, use the **scope loop** commands rb, rh, rw, wb,... to focus on the timing of read/write cycles using an oscilloscope.
3. Use the **'x' memory check** command to test for refresh arbitration error problems (if DRAM). Trigger on the 'Main Memory Trigger Point' address with the logic analyzer to observe timing and state machine faults if an error occurs. If multiport memory is involved, then run the memory test on the other port at the same the time as a final check for this step before proceeding.
4. Use the **'m' memory check** command to test for refresh and addressing problems (if DRAM). Trigger on the 'Main Memory Trigger Point' address with the logic analyzer to observe timing and state machine faults if error occurs. If multiport memory is involved, then run the memory test on the other port at the same the time as a final check for this step before proceeding.
5. Move the monitor to memory; repeat the memory tests; then use the jump command to start executing from memory. This will further exercise the DRAM memory system.
6. Change the default segment to'k0' by entering **'k0'** and jump to the monitor in memory. Next, run the memory check test (words, bytes, and half words) in the'k1' address space. This will saturate the read/write data bus and buffers in order to catch possible state machine problems.

7.   Install the IDT/sim monitor in the same EPROM with the Micromonitor. Put the IDT/sim first and Micromonitor second (the opposite is possible but this order makes run time support for downloaded code possible). Run debug command and try system tests.

Exception Handling

When an exception is encountered, the monitor jumps to location 0xbfc00100 or 0xbfc00180, and the monitor stores the EPC, Status, and Cause registers to the memory location specified by EPC_LOC (usually 0xa0000000), in successive words respectively. In this way, if a logic analyzer is to trigger at location 0xbfc00180, it will capture the values as they are put on the memory/cache bus. If the monitor and UART are functioning, the monitor will print the EPC and Cause registers on the terminal.

Using the Systems Diagnostics Command

Introduction

The 'diag' command enables the user to perform a variety of low-level hardware diagnostic tests. Upon entering the command at the IDT/sim prompt, a secondary menu listing the various available tests is presented.

Because the use of the 'diag' command is different from the other user commands, a separate explanation is provided in this chapter. Details of the other IDT/sim user commands are provided in Chapter 11. This chapter also includes some specific tests for the IDT 79S134 board.

Following is a list of tests and actions that the user may select. These tests are generally available on all IDT evaluation boards.

- *Run All Tests*
- *Memory Test*
- *Cache Test*
- *System Test*
- *DRAM Test*
- *DMA Test*
- *Timer Test*
- *Interrupt Test*
- *Set Options*
- *History*
- *Loop On All Tests*
- *Help*
- *Quit*

The following additional test is offered for the RC32334/2 based target boards depending on its configuration:

- *PCI Test*

The Following additional test is offered on the RC32355 based target boards depending on its configuration:

- *I2C Test*

The following additional test is offered on the EB438 and RP355 boards if flash support is enabled.

- *Flash Test*

Tests Common to all IDT Boards

Memory Test

The Memory Test checks the main memory and isolates all open/shorts and stuck-at faults for both address and data lines. The memory range over which to run the test can be defined. It defaults to the entire available address space less the low memory required by IDT/sim.

Cache Memory Test

This verifies that the cache memory, both instruction and data, pass the memory test by isolating the respective cache and running the memory test on each. It also verifies that instructions run cached and that they execute at one instruction per cycle. The cache tags are also tested for both instruction and data caches. Each cache is, in turn, isolated. Incrementing tag/cache addresses are written and then read with a check to make sure that there was a cache hit for all valid tags.

System Test

The System Test performs a memory test running in cached mode with varying data patterns. Full word, half word, and byte accesses are performed to verify proper state machine functionality. Write-through and read-back are verified for all cache conditions (valid-hit/miss and not valid). This test executes cached, if possible.

DRAM Test

A pattern is written to each memory location and read back. This test is performed across the entire DRAM address space which can be several MegaBytes. An informative message is displayed at the end of reading or writing each MegaByte of data. If the first message does not get displayed within the first few seconds of the test, one can conclude that a malfunction has occurred. At the end of the entire test (write, read, compare across the whole address space of available DRAM), a final message "Passed" or "Failed" is displayed. This test may take several minutes depending on the DRAM available.

DMA Test

The DMA can transfer data between:
1. memory and memory
2. memory and I/O device
3. memory and PCI device (RC32334/2 only)
4. PCI device and I/O device(RC32334/2 only)
5. PCI device and PCI device(RC32334/2 only)
6. I/O device and I/O device

Not all of the above combinations are tested. In fact, at the present time, only two types of tests are performed.

Single Configuration: A small (16 bytes) block of data is transferred from location 0xA0009000 to 0xA0009500 using memory to memory transfer capability. The transfer is verified and a "Passed" or "Failed" status is reported.

Multiple Configuration: This is a more comprehensive memory to memory test. Several features of the DMA is exercised in this test. "Passed" or "Failed" status is reported.

Timers Test

There are various timers on the RC323XX processors. The Timer Test does some minimal testing on the three general purpose timers which are on the chip.

The timers are tested by verifying that each timer increments and then rolls over. Each timer is polled until the current reading is less than the previous reading. Since the timers only count up (from lower value to higher value), this polling technique insures that the timer roll over is detected.

Interrupt Test

This is a simple test of whether or not the interrupt handler is reached on an interrupt. On-chip timer #0 is used to generate an interrupt upon roll-over. A message indicating success is displayed when the interrupt occurs and the handler is reached.

Set Options

The 'Set Options' command allows the user to specify global specifications which will affect activities undertaken by some of the other commands. For example, the user will be allowed to set the memory range to be tested.

Tests Specific to RC3233x Based Boards

PCI Test

This test scans the PCI bus and detects any PCI devices that are present on the board. If it detects a known PCI device, it displays which PCI card/device was found on the PCI bus.

SPI Test

This test performs read/write operations to the EEPROM on the board through the SPI interface.

Test Specific to RC32355 Based Boards

I^2C Test

This test performs read/write operations to the NVRAM on the board through the I^2C Interface. In this process, the I^2C functionality and the NVRAM are tested.

Tests Specific to RC32438 Based Boards

PCI Test

This test scans the PCI bus and detects any PCI devices that are present on the board. If it detects a known PCI device, it displays which PCI card/device was found on the PCI bus.

SPI Test

This test performs read/write operations to the EEPROM on the board through the SPI interface.

I^2C Test

This test performs read/write operations to the NVRAM on the board through the I^2C Interface. In this process, the I^2C functionality and the NVRAM are tested.

Test Specific to EB438 and RP355 Boards

Flash Test

This test performs erase/program/read operations to the flash memory on the board. This test is destructive and will erase the contents of flash memory.

IDT/sim PROM Entry Points

General Description and Use

IDT/sim supplies a set of entry points to the functions inside the PROM code. Application programmers may access these functions by name.

A call to a function in IDT/sim PROM will cause a jump to a fixed absolute address at the beginning of the monitor from which, in turn, a jump will occur to the actual implementing routine within the monitor.

Table 10.1 contains a comprehensive list of all entry points in IDT/sim.

Entry #	Name	Description
0	idtstart	enter the monitor
1	n/i	
2	promexit	back to monitor code
3	prominit	reinit then cmd loop
4	n/i	not implemented
5	n/i	not implemented
6	open	open device
7	read	read device
8	write	write device
9	ioctl	i/o control
10	close	close device
11	getchar	get character from console
12	putchar	put char to con
13	showchar	show char
14	gets	get string
15	puts	put string
16	printf	formatted print
17	promexit	return to cmd loop
18-27	n/i	not implemented
28	flush_cache	flush all caches completely
29	clear_cache	flush portion of a cache
30	setjmp32	save stack state
31	longjmp32	restore stack state
32	n/i	not implemented
33	n/i	not implemented
34	sprintf	sprintf
35	atob	ASCII to bin
36	strcmp	string comp
37	strlen	string length
38	strcpy	string copy

Table 10.1 PROM Function Entry Points (Page 1 of 2)

Entry #	Name	Description
39	strcat	string concat
40	cli	command line interpreter
41	get_range	parse - range
42	tokenize	tokenizer
43	help	show help menu
44	timer_start(RC32355)	start timer
45	timer_stop(RC32355)	stop timer & return time
46	n/i	not implemented
47	n/i	not implemented
48	n/i	not implemented
49	printf32	formatted print for 32-bit code for R4xxx
50	iInitLcd(RC32355)	Init lcd on board
51	lcd_print(RC32355)	print lcd message
52-54	n/i	not implemented
55	get_mem_conf	Get memory configuration
56	set_mem_conf	Set memory configuration
57-63	n/i	not implemented
64	exc_tlb_code	TLB exc boot vector
65	install_new_dev	user installed driver rt.
66	install_immediate_int	user in. handler hook - fast response
67	install_normal_int	user int.handler hook - normal response
68	install_command	user installed command
for ethernet support:		
69	tftpopen	open TFTP file
70	tftpclose	close TFTP file
71	tftpread	read TFTP file
72	tftpwrite	write TFTP file
73	tftplseek	seek TFTP file
74	soc_syscall	Indirect socket "system call"
75-95	n/i	not implemented
96	exc_cache_code	cache error exception
97-111	n/i	not implemented
112	exc_norm_code	general exc boot v.
113-127	n/i	not implemented
128	exc_norm_code	

Table 10.1 PROM Function Entry Points (Page 2 of 2)

IDT/sim User Commands

Notes

Overview

This chapter describes the implementation of the IDT/sim User Commands. All commands, with the exception of the system diagnostics command 'diag | dg', are covered. The 'diag | dg' command is described in Chapter 9.

A majority of the user commands enable the user to monitor both the system hardware and software. It is for this reason that IDT/sim is sometimes referred to as a 'monitor.'

Facilities provided include operating the CPU under controlled conditions, examining and altering the contents of memory, manipulating and controlling resources for the CPU (such as cache, TLB and copro-cessors), loading programs from host machines and controlling the execution path of these programs.

Issuing Commands

All commands to the monitor are entered on the command line when the cursor is at the input prompt **<IDT>**. The command line can be edited by using the following special characters.

^c (control-c)	Terminates current input/output and/or command in progress
^h (control-h)	Backspaces and deletes the previous character
^p (control-p)	Brings up the previous command on the command line
^u (control-u)	Deletes the entire line
Break key	If debug interrupt is enabled, returns control to <IDT> prompt

Command Format

The general command format is as follows:

command   *[options]   [argument 1] [argument 2]... [argument n]*

All *options* are entered as a minus sign (-) followed by an alphanumeric character (e.g., -w -b).

Arguments may be such items as a device name, address or count. Later in this chapter, the description of each command will specify the options and arguments required or accepted by that command.

Documentation Conventions

Conventions used in this document to show the commands and their arguments are as follows:

< > - An option or argument surrounded by these symbols is 'required'.

[] - An option or argument surrounded by these symbols is 'optional'.

| - When options or arguments within brackets are separated by the 'or' (|) symbol, it means that only one of the options or arguments may be specified.

/ - When options or arguments within brackets are separated by the (/) symbol it means that one or more of the options or arguments may be specified.

Command Specifications

◆ *To explicitly specify the radix when entering a number, the following convention must be used:*

It is assumed that host port 'tty3' is hooked up to the board port 'tty1'.

To start the download, enter the escape character (^z) followed by the letter 'l'.

UNIX "cat" command will work as well. In place of the "cp" command above, use the following command:

        cat program.srec

Do not hit the carriage return. Use ^zl as explained above. Once the download is complete, the user should exit the emulation mode by entering the escape character (^z) followed by the letter 'q'. This will return the user to the IDT/sim prompt. At this point, the user may start execution of the downloaded program.

Execution Control Commands

Set or Display Breakpoint

brk|b [address list]

The 'brk' command will display all of the currently set breakpoints if no address list is supplied. If an address list is supplied, breakpoints are set at each of the addresses in the list. There can only be up to 16 breakpoints set at any one time. It is also important to note that breakpoints are segment-specific. If the code to be executed is to run in kseg0, then it is necessary to set the breakpoint in kseg0. For example, if the program starts executing at address 0x80014000 (0x14000 in kseg0) and a breakpoint is desired at address 0x14008, the following sequence of commands ought be executed:

    <IDT>seg -0

    <IDT>b 14008

This assumes the default radix is hexadecimal. Once the default segment has been set, it will remain in force until changed by the user or until the board is reset. Setting breakpoints at the same address in both kseg0 and kseg1 is not permitted.

Call a Subroutine

call|ca <address> [arg1 arg2... arg8]

This command invokes a subroutine under the monitor environment. It will perform a *jump and link* to 'address' passing any arguments (up to 8) while still in monitor mode. The arguments must be integers and will be placed in the appropriate registers according to the MIPS calling convention.

When a client program is started by executing a 'go' command, a client environment is established which includes all registers, a stack and a set of global data. When a 'call' command from the monitor mode is made, this client environment is maintained (i.e., the client's stack and register contents are left unchanged). The 'gp' register is initialized to the value of the client's 'gp' prior to invoking the *jump and link*. Any effect that the called procedure has on the client's global data will persist after the call.

Continue Execution

cont|c

This command continues execution of the client process from where it last halted execution as a result of a 'brk', 'next', 'step', or 'gotill' command.

Go (Run Program)

go|g [-n] [address]

-l	Tribyte left access
-r	Tribyte right access

The default access type is 'word'(32-bits). This is true for all commands that allow size types. Word accesses may be directed to non-word-aligned addresses (i.e., fill memory with words starting at address offset 1). To handle non-word-aligned writes, the monitor will use the *swl* and *swr* instructions.

This will allow the user to debug software utilizing data structures that are not word aligned. In the 'RANGE' specification, the 'count' is the number of words, halfwords, or bytes to store (i.e., if the option is '-w' and 'RANGE = 1000/256', then 256 words would be affected). Note that the 'count' is always a decimal number independent of default radix.

Assembler

asm <addr>

This command allows the user to examine and change the memory interactively, using standard assembler mnemonics. When the user enters this command, on the next line the monitor will output the address specified by 'addr' followed by the contents of this address in hexadecimal and a disassembly of the contents.

At this point, the user may enter a new instruction mnemonic, a carriage-return or a period(.).

If the user enters a new instruction, the current contents at the address are replaced by the instruction, and the monitor outputs the next sequential address and its contents to the next line on the screen and waits for the next user input.

If the user presses the 'Enter' key, the monitor will not alter the contents of address and will just output the next sequential address and its contents to the next line on the screen. This sequence can be repeated over and over until the user enters a period (.). This terminates the 'asm' command and the monitor will output the standard command prompt (<IDT>) and wait for the next command to be entered.

Examples: assume, seg=kseg1 and rad=hexadecimal.

Memory starting at 0xa0005000 contains:

```
0xa0005000: 24090001     li      t1,1
0xa0005004: 00000000      nop
0xa0005008: 240a0002     li      t2,2
0xa000500c: 0c001400     jal      ra,0xa0005000
0xa0005010: 00000000     nop
```

User input is underlined in the following listing.

```
<IDT>asm 5000
0xa0005000: 24090001     li      t1,1
                         add      t1,t1,t2
0xa0005004: 00000000     nop
                            move    t3,t1
0xa0005008: 240a0002     li      t2,2
                          nop
0xa000500c: 0c001400     jal      ra,0xa0005000
                            b        -3
0xa0005010: 00000000     nop

                            .

 <IDT>
```

The above sequence would leave the following pattern in memory:

```
0xa0005000: 012a4820     add      t1,t1,t2
```

The above command fills memory as shown below:

a0050000: a0050000

a0050004: a0050004

       :   :

a00500fc: a00500fc

Flash Programming

fb [[-f flash][-e entry][-n]][[HOST;]FILE]

This command downloads the binary file via TFTP and programs flash memory on the board. Currently, this command is supported on the EB438 and RP355 boards.

The 'fb' command recognizes MIPS ECOFF and ELF format files. The file to be downloaded is specified by the HOST:FILE argument. The HOST section is the internet address of the remote TFTP server in internet dot notation. If the HOST section is not specified, a default value is obtained from the 'bootaddr' environment variable. The FILE section is a filename in a suitable format for the remote file server. If the FILE section is not specified, a default value is obtained from the 'bootfile' environment variable.

If the file format is RAW binary, the user must use -f flash to specify the starting address in flash memory to begin programming. If the -e entry switch is given, the environmental variable $ep is updated in NVRAM with the value given by entry.

Using the -n switch will suppress the updating of $ep. If the $ep variable is set, the user must press any key during the IDT/sim boot process, otherwise the program will jump to the Location address stored in $ep.

Fill Register

fr [-s|-d] <reg#|name> <value>

This command puts 'value' into the register specified by 'reg#|name'. However, an exception is made for double precision floating point registers. In case of double precision, the registers must be accessed as "dn" where "n" is an even number. To fill register r3 with the value ox12345678, the user may enter either of the commands below:

        <IDT>fr r3 0x12345678

        <IDT>fr v1 0x12345678

As a convenience, the special name 'pc' is used for the current program counter. In the MIPS architecture, the current program counter is not contained in any register. The name 'pc', as far as the IDT monitor is concerned, refers to the contents of the exception program counter (*EPC*) register in coprocessor zero (CP0).

When the execution control commands (go, continue, gotill) are used, execution will continue from the contents of the *EPC*. The names 'pc' and 'co_epc' are identical internally. The command sequence shown below would start execution at location 0xa0020000. In that the 'fill register' command does not assume that an address is being entered, it should be noted that the entire 32 bit virtual address needs to be entered.

        <IDT>fr pc 0xa0020000

        <IDT>go

Lock Cache

lc -[i|d]

This command "locks" the Instruction cache or the Data cache on RISControllers which support the cache locking feature.

Move Block

move|m [-w|-b|-h] <RANGE> <destination>

Moves the block of memory specified by RANGE to the address specified by 'destination'. The destination address may be before, after, or within the block of memory to be moved, and the 'move' algorithm will move through the block forward or backward so as not to destroy any data. The second example shows the case of overlapped source and destination regions as specified by the addresses.

Examples:

assume: seg.=kseg1 and rad=hexadecimal.

<IDT>m 5000/128 5800

The above command moves 128 words of data, starting at location 0xa0005000 to a 128 word block of data starting at location 0xa0005800.

<IDT>move 5000-5fff 5800

The above command moves the block data between addresses 0xa0005000 and 0xa0005fff to the block of data between addresses 0xa0005800 and 0xa00067ff. The 'move' algorithm is such that this will not result in destroying the original data between addresses 0xa0005800 and 0xa0005fff.

Read Cache Memory

rc [-i] [-w|-b|-h] <RANGE>

Addresses are automatically set to Kseg0 and the caches are isolated. Memory contents are read starting at 'start_addr' until 'end_addr' is reached. If a 'count' was specified instead of an end address then memory is read from 'start_addr' to 'start_addr+count'.

The options have the following meaning:

-i   - Select the instruction cache.

-w   - Word access.

-b   - Byte access.

-h   - Halfword access.

The default access type is 'word' and the default cache is the 'data' cache.

Search Memory

search|sr [-w|-b|-h] <RANGE> <value> [mask]

This command will search the area of memory specified by 'RANGE' for the 'value'. Prior to the check, each memory location and value will be 'anded' with the mask, if specified. When a match is found, the address is displayed and the user may enter a carriage return to continue searching or a period (.) to terminate the search operation. User input is underlined below.

Examples:

assume: seg.=kseg1 and rad=hexadecimal.

<IDT>sr 5000/128 12345678

Match: a0005010=12345678

<IDT>

The above command searches 128 words of data, starting at location 0xa0005000, for the word containing the value 0x12345678. A match is found at location 0xa0005010. Only one match was found and the monitor then returned to the command line displaying the command line prompt when the search completed.

<IDT>sr 4000/128 12345678 00ffff00

The above command searches 128 words of data, starting at location 0xa0004000, for a word containing the value xx3456xx, where 'x' can be any hex value from '0' to 'f'. Halfword access use only the least significant 16 bits of the mask and byte accesses use only the least significant 8 bits of the mask.

This command displays the contents of the Translation Lookaside Buffer (TLB). If a 'RANGE' is specified, just the contents within the 'RANGE' are dumped, otherwise the entire buffer is dumped.

TLB Flush

tlbflush|tf [RANGE]

This command flushes the contents of the TLB. If a 'RANGE' is specified, just the contents of the 'RANGE' are flushed, otherwise the entire buffer is flushed.

TLB Map

For RC323xx CPU:

tlbmap|tm [-i index] [-cdgv] [-p pagesize] [-c cachealg] <vaddress> <paddress>

The 'tlbmap' command establishes a virtual to physical mapping in the TLB. A particular entry may be specified with the *-i* option; if no TLB entry is specified, a random entry is selected between 8 and 63. The *-n, -d, -g* and *-v* options cause the corresponding bits in the TLB entry to be set to **1**, otherwise they are set to **0**. The default segment is not applied to these addresses.

The 'tlbmap' command establishes a virtual to physical mapping in the TLB. A particular entry may be specified with the *-i* option;

These switches may optionally be followed by '0' or '1' to specify an individual TLB low entry. By default, both TLB low entries are affected. The *-p* switch allows the TLB page size to be set. It may take one of the following values:

0x00001000	4 kbyte page
0x00004000	16 kbyte page
0x00010000	64 kbyte page
0x00040000	256 kbyte page
0x00100000	1 Mbyte page
0x00400000	4 Mbyte page
0x01000000	16 Mbyte page

The *-c* switch allows the TLB cache algorithm to be set. It may take one of the following values:

0-	Cacheable, noncoherent, write-through, no write allocate
1-	Cacheable, noncoherent, write-thourgh, write allocate
2-	Uncached
3-	Cacheable, noncoherent, write-back
4-	Reserved
5-	Reserved
6-	Reserved
7-	Reserved

If two physical addresses are supplied, the two 'tlblo' entries are set to use them. If one physical address is supplied, 'tlblo0' entry is set to the physical address and 'tlblo1' entry is set to the physical address plus the selected page size.

TLB Process ID

tlbpid|ti [pid]

The 'tlbpid' command without argument displays the current process identifier in the system coprocessor register 'tlbhi'. If an argument is supplied, the current process identifier in 'tlbhi' is set to 'pid'.

TLB Search For Physical Address Map

tlbptov|tp <physaddr>

The 'boot' command recognizes MIPS ECOFF and ELF format files. The file to be downloaded is specified by the HOST:FILE argument. The HOST section is the internet address of the remote TFTP server in internet dot notation. If the HOST section is not specified, a default value is obtained from the 'bootaddr' environment variable. The FILE section is a filename in a suitable format for the remote file server. If the FILE section is not specified, a default value is obtained from the 'bootfile' environment variable.

Ping a Host

ping [-lnqrv] [-c COUNT] [-i WAIT] [-s SIZE] HOST

This command allows a remote host to be 'pinged' to see if it is available. ICMP echo packets are sent to the host and the reply packets are displayed showing the round-trip time.

The HOST argument selects the host to be 'ping'ed. It is given in internet dot notation.

The *-c* option allows a specified number of packets to be sent. By default "ping" will send packets continuously until it is interrupted by typing control-c.

The *-l* option allows a specified number of packets to be sent before timing commences. This allows the network load to reach a steady state.

The *-n* option forces internet addresses to be printed out in dot notation (this is the default behavior).

The *-q* option stops "ping" from displaying the results of each packet sent.

The *-r* option prevents the low level network code from routing packets.

The *-v* option enables verbose messages when unexpected packets are received.

The *-s* option allows you to specify a different packet size (maximum 1432, default 56).

Board Specific Commands

The following commands will work on boards with real-time capability and non-volatile memory.

Set / Display Date

date [[[mm]/dd]/yyyy]

The Set / display date command is used to display and set the date in a real time clock. With no arguments, the current date is displayed. The optional argument allows the date to be set. Each section of the string consists of two digits: yyyy - the year, you can give the entire 4 digits or the last two digits, mm - the month (01-12), dd - the day (01-31). Any unspecified field defaults to the current value.

Set / Display Time

time [[[hh]:mm]:ss]

The Set / display time command is used to display and set the time (and not the date) portion of a real time clock. With no arguments, the current time is simply displayed. The optional argument allows the time to be set. Each section of the string consists of two digits: hh - the hour (0-23), mm - the minutes (0-59) and ss - the seconds (0-59). Any unspecified field defaults to the current value.

Display Settings of Environment Variables

env

Displays the environment strings set in the NVRAM.

Set Environment Variable Values

setenv <VAR> <VALUE>

Sets the environment variable 'VAR' to 'VALUE'. The environment is stored in NVRAM and is preserved when the power is off. Spaces may be included in the 'VALUE' string by encompassing them inside quote marks.

For example:

                    `<IDT> setenv cmd "load -b -a tty1"`

                    `<IDT> $cmd`

Environment variables is a great way of saving repetitious typing at the sim command line.

Delete (unset) Environment Variable

unsetenv <VAR>

Deletes the environment variable 'VAR' from the NVRAM environment.

Delete (unset) all Environment Variable

purge

Deletes all the environment variable from the NVRAM environment.

    For Example:

              `<IDT>purge`

General Purpose Commands

caus

caus

This command reads and displays the *cause* register.

clcs

clcs

This command clears (sets to zero) the *cause* register bits.

clsr

clsr

This command clears (sets to zero) the *status* register bits.

cmpr

cmpr

This command runs a simple timer test.

creg

creg

This command allows the user to interactively set individual fields of the SDRAM control register.

gmsk

gmsk

This command is the retrieving counterpart of the *smsk* command explained below; it "gets" the *status* register to show the interrupt mask.

smsk

smsk

This command allows the user to set 'interrupt enable' and 'mask' bits. It asks the user to enter *status* register's interrupt mask and 'enable' bit values as hexadecimal numbers.

creg

Sets individual fields of the SDRAM control register.

date [[[mm]dd]yyyy]

Displays and sets date and time. Available on boards with NVRAM only.

dbgint|di [-e|-d] [DEVICE | Int. Line]

Enables or disables the facility that allows the user to interrupt an application program's execution and return to the monitor.

dc [-i|-d] RANGE

Displays the instruction or data cache contents and the tag values, if the cache location is valid.

debug|db [DEV]

Enters remote debug mode. 'DEV' can be 'tty0' or 'tty1'.

diag|dg

Runs low-level system diagnostics.

dis <RANGE>

Dis-assembles the contents of memory specified by RANGE. If RANGE consists only of a beginning address, enough locations following the beginning address are disassembled to fill one page.

dr [reg#|name|reg_group|-u]

Prints out the current contents of register(s).

dt

Displays the contents of the trace buffer.

dump|d [-d|-w|-h|-b] <RANGE>

Displays the contents of memory specified by RANGE.

env

Displays the environment variable string settings in boards with NVRAM.

fill|f [-d|-w|-h|-b|-l|-r] <RANGE> [value_list]

Fills memory specified by RANGE with 'value_list'.

fb [[-f flash][-e entry]][-n][[HOST:]FILE]

Downloads a binary file via ethernet and programs the flash memory on the target board.

fr [-s|-d] <reg#|name> <value>

Puts <value> into the register specified by <reg#|name>.

go|g [-n] <address>

Begins execution at address <address>.

gotill|gt <address>

Continues execution from the current value of the program counter. The program will stop execution just prior to the execution of the instruction pointed to by 'address'.

help|? [command list]

Prints out a list of commands available in the monitor. If a command list is supplied, only the syntax for the commands in the list is displayed.

history|h

Displays the last eight commands entered with identifying numbers so that the user may re-execute the command by entering '!#', where '#' is the command number.

init|i

Initializes prom monitor (warm reset).

isrt

Tests the interrupt handling of 79S134 board.

load|l [-b|-a|-s|-t] <device>

Input S-records from 'device'.

-b: binary download; needs the program 'bdl'.

-a: turns off handshake protocol which indicates end of each S-record.

-s: silent mode; no dots are printed on screen.

-t: download via ethernet; needs entire ip address and filename following '-t'.

-n: suppress update of $ep variable if flash is programmed.

lockcache|lc [-i|-d]

Locks i-cache or d-cache in RISControllers capable of doing so.

move|m [-w|-b|-h] <RANGE> <destination>

Moves the block of memory specified by RANGE to the address specified by destination.

next|n [count]

Similar to the 'step' command except that when a *jal* or *bal* instruction is encountered, all instructions of the sub-procedure are executed until the sub-procedure returns to the instruction following the *jal* or *bal*.

ping [-lnqrv] [-c COUNT] [-i WAIT] [-s SIZE] HOST

Sends ICMP echo packages to remote host to check if the host is available.

purge

Clears the contents of the NVRAM.

rad [-o|-d|-h]

Sets the default radix to the requested base (Octal / Decimal / Hexadecimal).

rc [-i] [-w|-b|-h] <RANGE>

Reads the cache memory specified by RANGE. Addresses are automatically set to Kseg0 and the caches are isolated.

regsel|rs [-c|-h]

Selects either the compiler names or the hardware names for registers.

search|sr [-w|-b|-h] <RANGE> <value> [mask]

Searches the area of memory specified by RANGE for 'value'.

seg [-0|-1|-2|-s|-3|-u]

Set the default segment to the requested k-segment.

setbaud|sb DEV

Allows user to select the baud rate for the device specified by DEV which may be either 'tty0' or 'tty1'.

setenv VAR VALUE

Allows the user to set environment variables in NVRAM.

smsk

Sets interrupt mask.

step|s [count]

Executes a single step or if <count> is supplied then 'count' number of steps.

sub [-w|-h|-b|-l|-r] <address>

Allows user to examine and change memory interactively.

t [-a/-o/-e/-d/-r RANGE/-w RANGE/-c RANGE/-i   INS/-m MSK]

Defines the trace equation and/or enables/disables tracing.

tc [-e BPNUM] [-d BPNUM]

This command defines the limits of tracing. By placing breakpoints at the beginning and end of a code segment and using the 'trace conditionally' command a user may specify the section of code to trace.

te

Puts IDT/sim in a transparent mode and connects the console port straight through to another serial port.

tex [RANGE]

Excludes a memory range from being traced.

time [[[hh]:mm]:ss]

Displays and sets time. Available on boards with NVRAM only.

tlbdump|td [RANGE]

Dumps the contents of the translation look aside buffer. If a range is specified, just the range is dumped, otherwise the entire buffer is dumped.

tlbflush|tf [RANGE]

This command flushes the contents of the translation buffer. If a range is specified, just the range is flushed, otherwise the entire buffer is flushed.

tlbmap|tm [-i INX} [-(v|d|g)[0\1]] [-g] [-p PAGESIZE] [-c CACHEALG] VADDR PADDR [PADDR]

Establishes a virtual-to-physical mapping in the translation buffer.

tlbpid|ti [pid]

Without arguments, this command displays the current process identifier in the system coprocessor register 'tlbhi'. If an argument is supplied, then the current process identifier in 'tlbhi' is set to < pid >.