



IDT/sim User/Developer's Manual

**Version 2.0
July 1999**

2975 Stender Way, Santa Clara, California 95054
Telephone: (800) 345-7015 • TWX: 910-338-2070 • FAX: (408) 492-8674
Printed in U.S.A.
© 1999 Integrated Device Technology, Inc.

Integrated Device Technology, Inc. reserves the right to make changes to its products or specifications at any time, without notice, in order to improve design or performance and to supply the best possible product. IDT does not assume any responsibility for use of any circuitry described other than the circuitry embodied in an IDT product. The Company makes no representations that circuitry described herein is free from patent infringement or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent, patent rights or other rights, of Integrated Device Technology, Inc.

LIFE SUPPORT POLICY

Integrated Device Technology's products are not authorized for use as critical components in life support devices or systems unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of IDT.

1. Life support devices or systems are devices or systems which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any components of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

The IDT logo is a registered trademark, and BiCameral, BurstRAM, BUSMUX, CacheRAM, DECnet, Double-Density, FASTX, Four-Port, FLEXI-CACHE, Flexi-PAK, Flow-thruEDC, IDT/c, IDTenvY, IDT/sae, IDT/sim, IDT/ux, MacStation, MICROSLICE, PalatteDAC, REAL8, RC3041, RC3051, RC3052, RC3081, RC36100, RC4600, RC4640, RC4650, RC4700, RC5000, RISController, RISCORE, RISC Subsystem, RISC Windows, SARAM, SmartLogic, SyncFIFO, SyncBiFIFO, SPC, TargetSystem and WideBus are trademarks of Integrated Device Technology, Inc.

MIPS is a registered trademark, and RISCompiler, RISComponent, RISComputer, RISCware, RISC/os, R3000, and R3010 are trademarks of MIPS Computer Systems, Inc. Postscript is a registered trademark of Adobe Systems, Inc. AppleTalk, LocalTalk, and Macintosh are registered trademarks of Apple Computer, Inc. Centronics is a registered trademark of Genicom, Inc. Ethernet is a registered trademark of Digital Equipment Corp. PS2 is a registered trademark of IBM Corp.



Table of Contents

Notes

1 IDT/sim Debug Monitor

Overview

Overview	1-1
What does IDT/sim do?	1-1
Who should read this manual?	1-1

2 Developing IDT/sim

IDT/sim source code installation	2-1
Compiler installation	2-1
Compiler test runs	2-1
Cross-compiling issues	2-1
Building IDT/sim	2-2
Testing the IDT/sim executable	2-2
IDT/sim or Micromonitor	2-3

3 Minimum IDT/sim Start-up File

Overview of IDT/sim source files	3-1
Directory structure	3-1
Creating a minimum version of IDT/sim	3-2
Modifying start-up file csu_idt.S	3-2
SIM3000/csu_idt.S	3-2
CPU identification (R30xx)	3-3
RAM accessibility (R30xx)	3-3
Subroutine 'initmem' (R30xx)	3-4
Initialize device table (R30xx)	3-4
Initialize IDT/sim to known state (R30xx)	3-4
Initialize command table (R30xx)	3-4
Clear breakpoints (R30xx)	3-5
SIM4000/csu_idt.S	3-5
RAM accessibility (R4xxx)	3-5
Subroutine 'initmem' (R4xxx)	3-5
Initialize device table (R4xxx)	3-6
Initialize IDT/sim to known state (R4xxx)	3-6
Initialize command table (R4xxx)	3-6
Clear breakpoints (R4xxx)	3-6

4 Minimum IDT/sim User Commands

Command table	4-1
Commands not required for minimal IDT/sim versions	4-1

5 Adding & Deleting User Commands

Command Table Structure	5-1
Command Table Entries	5-1
SIM3000	5-1
SIM4000 & R32364-based	5-3

6 Adding & Deleting IDT/sim Device Drivers

Overview	6-1
Device Switch Table	6-1
Device Initialization Table	6-1
Actual Device Switch Table	6-2
Actual Device Initialization Table	6-2
Example Device Driver	6-4
Source File cendrvr.c	6-4
Source File centron.s	6-5
Header file centron.h	6-7
Make File Makecen	6-7

7 Using Micromonitor

Introduction	7-1
Using Micromonitor	7-1
User Commands	7-1
Store	7-1
Load	7-2
Jump	7-3
Dump	7-3
Fill	7-3
Compare	7-3
Transfer	7-4
Scope Loops	7-4
Set Segment Default	7-4
Print Stack	7-4
Memory check	7-5
Porting to new hardware	7-5
Recommended Debug Technique	7-6
Exception handling	7-6

8 Using the Systems Diagnostics Command

Introduction	8-1
List of available tests	8-1
Tests common to all IDT boards	8-2
Memory Test	8-2
Cache Memory Test	8-2
System Test	8-2
MMU Test	8-2
FPU Test	8-2

Set Options	8-2
Tests specific to R36100.....	8-2
Cache Locking Test.....	8-2
Timer Write Compare Ack Test	8-2
PIO Test	8-3
Parallel Test.....	8-3
Serial Test	8-3
DMA Test.....	8-3
Tests specific to IDT79S134 board	8-3
SRAM test.....	8-3
DRAM test.....	8-3
DMA test	8-3
PIO test	8-3
Interrupt Test	8-4
Internal UART test.....	8-4
Timers test.....	8-4
SPI tests for 79S134 board.....	8-4
spi256.....	8-4

9 IDT/sim PROM Entry Points

General Description and Use.....	9-1
_EXIT	9-3
ATOB.....	9-3
CLEAR_CACHE.....	9-3
CLI.....	9-4
CLOSE	9-4
EXC_CACHE_CODE.....	9-4
EXC_NORM_CODE	9-4
EXC_UTLB_CODE	9-4
EXC_XTLB_CODE	9-4
FLUSH_CACHE.....	9-4
GET_MEM_CONF	9-4
GET_RANGE	9-4
GETCHAR.....	9-5
GETS	9-5
HELP	9-5
INSTALL_COMMANDS.....	9-5
INSTALL_IMMEDIATE_INT	9-5
INSTALL_NEW_DEV	9-6
INSTALL_NORMAL_INT.....	9-6
IOCTL.....	9-8
OPEN	9-8
PRINTF/SPRINTF	9-8
PUTCHAR.....	9-8
PUTS.....	9-9
READ	9-9
REINIT	9-9
RESET	9-9
RESTART	9-9
SET_MEM_CONF	9-9

SETJMP/LONGJMP & SETJMP64/LONGJMP64	9-9
SHOWCHAR	9-10
STRING	9-10
TFTPCLOSE	9-10
TFTPLSEEK	9-10
TFTPOPEN	9-10
TFTPREAD	9-10
TFTPWRITE	9-10
TIMER_START	9-10
TIMER_STOP	9-11
TOKENIZE	9-11
WRITE	9-11

10 IDT/sim User Commands

Overview	10-1
Issuing Commands	10-1
Command Format	10-1
Documentation conventions	10-1
Command Specifications	10-1
Command categories	10-2
Communication/Host interface commands	10-2
Debug - GDB and IDT/c version 5.0 or later	10-2
GreenHills debugger	10-3
Download Program from Host to Board	10-3
Set Baud rate of tty Port	10-4
Terminal Emulator	10-4
Execution Control Commands	10-5
Run User Benchmark	10-5
Set or Display Breakpoint	10-5
Call a Subroutine	10-6
Continue Execution	10-6
Go (Run Program)	10-6
GoTill	10-6
Next (step over subroutine)	10-6
Single Step	10-7
Unbreakpoint	10-7
Memory/Register and Assembly/Disassembly commands	10-7
Assembler	10-7
Cache Flush	10-9
Compare Block	10-10
Disassemble Contents Of Memory	10-10
Dump Cache	10-10
Dump Memory	10-11
Dump Registers	10-11
Fill Memory	10-12
Fill Register	10-12
Lock Cache	10-13
Move Block	10-13
Read Cache Memory	10-13
Search Memory	10-13

Substitute Memory	10-14
Write Cache Memory.....	10-15
Set-up and Environment Commands	10-15
Checksum	10-15
Help Command	10-15
History Command	10-15
Initialize	10-16
Register Set Select	10-16
Set Default Radix	10-16
Set Default Segment	10-16
TLB Commands.....	10-16
TLB Dump	10-16
TLB Flush.....	10-16
TLB Map.....	10-17
TLB Process ID	10-17
TLB Search For Physical Address Map	10-17
Trace Commands	10-18
Trace Command.....	10-18
Trace Stop Command	10-19
Trace Conditionally Command	10-20
Trace Dump Command.....	10-21
Trace Exclude Command.....	10-21
Trace Command Examples.....	10-22
Network Related Commands	10-22
Download and execute binary file (boot)	10-22
Ping a host	10-23
Board specific commands.....	10-23
Set / display date and time	10-23
Display settings of environment variables	10-23
Set environment variable values	10-23
Delete (unset) environment variable	10-23
Commands specific to the 79S134 evaluation board	10-24
caus.....	10-24
clcs	10-24
clsr.....	10-24
cmpr	10-24
creg	10-24
gmsk.....	10-24
isrt	10-24
smsk.....	10-24
SPI256 related commands.....	10-24

11 IDT/sim User Command Summary

Quick Reference	11-1
IDT/sim User Commands	11-1

12 Motorola S-record Format

13 Register Numbers and Names

Floating-Point Registers.....	13-3
Control registers.....	13-3

Appendix A IDT/sim and Secondary CACHE of R5000



List of Tables

Notes

Table 4.1	Commands not required for minimal IDT/sim versions (Page 1 of 2).....	4-1
Table 7.1	Micromonitor's UART Functions.....	7-5
Table 9.1	PROM Function Entry Points (Page 1 of 3)	9-1
Table 13.1	Floating Point Register Usage	13-3





IDT/sim Debug Monitor Overview

Notes

Overview

The System Integration Manager (IDT/sim) Debug Monitor is a software/firmware tool that permits the convenient downloading, execution, troubleshooting and diagnoses of code for a variety of IDT RISC system products. IDT/sim is available in two forms:

- ◆ **Executable code programmed in a set of EPROMS as firmware.** In this form, IDT/sim is present in the read-only-memory of all RISC microprocessor evaluation boards (available through Integrated Device Technology (IDT)) and is the operating system, the low-level debugger and the system monitor, with textual user interface for the boards.
- ◆ **Software source code on CD-ROM.** IDT/sim is also available as software source code written in 'C' and assembler. In this form, IDT/sim function modifications or enhancements can be made through a user configurable command table. This feature allows the user to simply link and load the enhanced function into memory and enter the entry point into the command table. Some special entry points for supporting stand-alone systems are also available.

What does IDT/sim do?

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

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

Who should read this manual?

The intention of this manual is to address the needs of both *users* and *developers*. So to effectively use this manual and obtain the most relevant information, it is important to identify the scope of your task. You are primarily a *user* of IDT/sim if you:

- ◆ *do not intend to change the available functions of IDT/sim, or*
- ◆ *are not interested in how IDT evaluation boards work but are interested in understanding the operating system commands for the boards, or*
- ◆ *are interested in evaluating an IDT RISC microprocessor, using one of the IDT evaluation boards as a vehicle for executing benchmarks, or*
- ◆ *have already decided to use an IDT RISC microprocessor 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, or*
- ◆ *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 RISC microprocessor, 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 RISC microprocessor behavior and/or IDT evaluation boards; both processor 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, based on an IDT RISC microprocessor*

Chapters 1 through 7 of this manual contain information of interest to developers. Chapters 8 through 13 contain references to IDT/sim commands and are intended for all IDT/sim users.



Developing IDT/sim

Notes

Before beginning any IDT/sim software development project, the *IDT/sim 8.0 Release Notes* should be reviewed. A copy of the release notes is included with the IDT/sim source code and provides installation commands, 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 the development platform of your choice, or on a member of the network to which your development platform is connected. The development platforms currently supported are Solaris 2.5.1, SunOS 4.1.3 and DOS 5.0 or above. 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 may build IDT/sim 8.0; however, none have been used or tested by IDT.

Sun code is supplied on a 150 MB QIC tape or a CD-ROM in tar format. To access the code, (1) **create** a new directory on your hard disk, (2) **assign** a name to the new directory, and (3) **extract** the tar formatted file over to the newly created directory. To install on Sun, simply use the following command:

tar xvof /dev/rst0 (or whatever the source device is)

The DOS code is supplied on floppy disks or CD-ROM. To access the code, (1) **create** a new directory on your hard disk, (2) **assign** a name to the new directory, and (3) **copy** all files from the disks over to the new directory, using the 'xcopy' command. Sim8.0 consists of three diskettes that have been compressed using a "pkzip" utility. When using the "pkunzip" or "winzip" utilities, follow the installation instructions provided in the *IDT/sim 8.0 Release Notes* mentioned above.

Compiler installation

Install IDT/C or another 'C' compiler tool-chain on the development platform of your choice. Because each IDT RISC microprocessor family (for example, the R30xx or R4xxx) has unique instructions, the tool-chain should include assembler support for that family. Some members of a family (for example, the R4600 or the R4650) also have instructions that are unique to each member.

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=r3000 or -m32364 in the case of IDT/C, for example). 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. Majority of the tool-chains provide some sample code for this purpose. For example, IDT/C has two 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. If you installed the IDT portion (as opposed to the GNU portion) of the IDT/C compiler in a directory called 'IDTC,' you will find the sample code located in 'IDTC/samples/hello' and 'IDTC/samples/stanford.'

Cross-compiling issues

The compiler you use for IDT/sim development will most likely 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 commonly made 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 the single-user DOS environment, 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 should have some knowledge of IDT's source code. The 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 tool-chains, evaluation boards and endianness. Selection of the appropriate 'makefile' must be based on the evaluation board you will be working with.

Make the directory where the IDT/sim source code was installed the current directory. There are several subdirectories below the root of installation level: 'COMMON', 'S364', 'S381', 'S465', 'S134', 'SIM3000', 'SIM4000', 'binutils', 'drivers', 'header', 'TIMERS', 'lib' and 'net'. If your evaluation board is based on a member of the R30xx RISController family, change the directory to SIM3000. If your evaluation board is based on a member of the R4xxx microprocessor family, go into the SIM4000 subdirectory.

Listing the contents of the directory selected (either the SIM3000 or the SIM4000) will reveal the following directories: SUN_MAKE and DOS_MAKE, which contain the 'makefiles' for building IDT/sim on either SUN host or DOS host machines. Both the SUN_MAKE and DOS_MAKE directories contain a list of directories that contain build environments (makefiles and linker scripts) for a specific IDT evaluation target, as reflected in the each file name.

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 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 'S?3' or 'prm' will have been created. These are the Motorola standard S-record files that can be downloaded to your EPROM programmer or ROM emulator. The number of S-record files created is equal to the number of IDT/sim EPROMs 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 and replace the existing ones in your evaluation board¹. 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).

If the sign-on message does not appear—and if your evaluation board has IDT/sim split into four EPROMs—try reversing the order in which you placed the EPROMs. The correct EPROM 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 to reverse the board's EPROM 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."

¹. To avoid programming errors for each IDT/sim revision during the development cycle, consider buying 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).

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 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.' At the early stages of your development efforts, it is a good practice to begin with a 'make clobber' or 'make clean' command, before actually running the '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 RISC-based 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 interface, you may port the Micromonitor code to your hardware and begin using it (information on using Micromonitor is available in Chapter 7).

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 the chapters ahead.



Minimum IDT/sim Start-up File

Notes

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

Overview of IDT/sim source files

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 code in the sources.

There are several files common to all IDT/sim variations. These particular files do not contain code that addresses unique board or processor conditions. However, there are also files common to all IDT/sim variations that do contain code that is used to address specific boards (a feature implemented by using `"#if defined()"` or `"#ifdef"` conditional compile directives).

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

Finally, there are also source files unique to one specific board. These files contain no conditional compile statements, no equivalent files in any other directory, and are called only for compiling and linking while creating IDT/sim for that specific board.

Directory structure¹

The `common/`, `header/`, `S134`, `S361`, `S364`, `S381/`, `SIM3000/`, `SIM4000/`, `net/`, `binutils/`, `drivers/`, `TIMERS`, and `lib/` 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 flavors of IDT/sim. Some of these files contain conditional compile directives for different boards or processors, or both. For example, `"#ifdef R381"` indicates code specific to the 79S381 board; `"if defined(CPU_R4000)"` indicates code applicable to all derivatives of R4000, including the Orion family; and `"if defined(CPU_R3000)"` indicates code applicable to all IDT derivatives of R3000 CPUs.

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

S134/: This directory contains the start-up code and R32364 specific CPU initialization and PCI initialization code.

S361/: This directory contains the start-up code and R36100 specific CPU initialization and 8530 UART initialization code.

S364/: This directory contains the start-up code and R32364 specific CPU initialization, NVRAM driver and exception handling code.

S381/: This directory contains the start-up code and 2681 UART drivers specific to S381 target.

S465/: This directory contains an 8530 UART driver and NVRAM driver specific to an S465 target.

TIMERS/: This directory contains timer routines that are based on R4000 CPU count/compare registers and the SONIC-ethernet chip's watchdog timers.

SIM3000/: This directory contains code that is common to boards designed for derivatives of an R3000 core. In addition, this directory contains the environments to create IDT/sim for each R3000 CPU and

¹ For supplementary information on this topic, see Technical Note TN-20 in the *RISC Microprocessor Application Guide*.

R32364 target under SUN-host and MSDOS-host respectively. For example, the 'Makefile' for creating IDT/sim for a BigEndian S381 target under SUN-host can be located under './SUN_MAKE/S381.'

SIM4000/: This directory contains code that is common to boards designed for derivatives of an R4000 and R32000 cores. In addition, it contains the environments for creating IDT/sim for each R4000 CPU target, under SUN-host and MSDOS-host respectively. For example, the 'Makefile' for creating IDT/sim for a BigEndian S465 target, under SUN-host, can be found under './SUN_MAKE/S465.'

net/: This directory contains the ethernet code (from UC-Berkley) and ethernet drivers that are specific to S381, S364 and S465 targets.

binutils/: This directory contains the "conv" utility—for SUN-host and MSDOS host—that converts from ELF-executable to Motorola Style S3-records.

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

lib/: This directory contains BigEndian and LittleEndian math-emulation libraries for 64-bit R4650 targets and is required by the diagnostics (dg) command for an R4650 CPU. These libraries are not required if use of the diagnostics command is not planned.

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, in the beginning 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 DRAM, it is very important that the DRAM interface be correctly programmed and fully functional, to facilitate loading code into the DRAM space. If the board has an SRAM interface, it should be considered the next priority, and so on.

Assuming that the new board's EPROM 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 the IDT eval board design that is most similar 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 /SIM3000, /SIM4000 or /Sxyz where "xyz" is the abbreviated name of an evaluation board (such as /S134, /S381), depending on the target CPU. 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 majority of the code or subroutine calls to code which does initial configuration (if needed) of the target CPU and the board hardware.

Additionally, the start-up file contains code to perform many other tasks which are not necessarily required in a bare minimum IDT/sim. Let us review the start-up files SIM3000/csu_idt.S and SIM4000/csu_idt.S and see how we can come up with bare-minimum versions of these files. If your interest is in boards based on R4xxx CPU, please skip over to the section titled 'SIM4000/csu_idt.S'. Others may continue to read the next section and may ignore 'SIM4000/csu_idt.S' section.

SIM3000/csu_idt.S

Note: While reading this section, please refer to a print-out of the 'SIM3000/csu_idt.S' file.

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

The first subroutine in the file is *start*. At the beginning of *start*, there are a series of 80 *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 prom entry point table is not required for basic functionality of IDT/sim, and the entire table may be removed or commented out. When deleted, your code will start at the label *idtstart*.

The SIM3000/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.

As an example, if your board is not based on the 79S381 board, you may safely delete all lines of code between and including the lines *#ifdef RS381* 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.

Let us assume now that the jump table and all unrelated code has been deleted. The next step of the *start* subroutine deals with initializing specific registers in some of the R30xx processors. To locate this code, search for the string 'CPU identification' in the file.

CPU identification (R30xx)

In this portion of code, the logic implemented is relatively uncomplicated. The coprocessor register that contains the product identification is read into a temporary register. If the value read is hexadecimal 700 or 701, the target CPU is identified as an R3041 or 3041A, and the execution jumps to the label *its4* where the R3041 *BusCtrl* and *PortWidth* registers are initialized.

Also, if the product identification value is identified as hexadecimal 700 or 701, then a test for the presence of a floating point accelerator (FPA) is performed. If an FPA is detected, the target CPU is identified as an R3081. If an FPA is not detected, then the CPU is identified as an R3051 or R3052. You could initialize the Config register to suit your needs, if you detect an R3081.

If the R30XX CPU type is already known, code size can be reduced by removing the portion of code that relates to CPU identification. The IDT/sim port can then be hard coded for that particular CPU. You can always bring the CPU identification code back into play if you desire more flexibility in the future.

RAM accessibility (R30xx)

If the data does not match, it is determined that 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). At early porting stages, and even on a more permanent basis, it may be useful to have an LED display indicate that a RAM error has occurred, if the test fails.

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. To evaluate which of these are important and should be ported to a minimum version of IDT/sim the sequence that these tasks are performed will be detailed in the following section, using a 'SIM3000/csu_idt.S' file as an example.

Subroutine 'initmem' (R30xx)

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, the status register is read to see if the target CPU is of 'E' type (TLB present) or not. To indicate the presence of TLB, a variable is set to a non-zero value. Now a test for the presence of FPA is performed and the PRID register of CP0 is read and a variable is set to a unique value to indicate the type of CPU. If you already know what target you are going to work with, you may delete all of these tests and simply set the variables to known values. Note that this will make the code inflexible and you will not be able to change CPUs on your board.

Next, a certain amount of cache related work is performed. First the sizes of both data and instruction caches are determined and stored in variables for future reference. This is done in the subroutine 'config_cache' which calls the subroutine '_size_cache'. Both subroutines are for R3000 CPU targets and are defined in the file 'common/except.S'. If you know which CPU you are going to be using, the cache sizes are already known, and you can simply set the appropriate variables (`dcache_size`, `icache_size`) to the known values, deleting the remaining code that sizes the caches.

Following cache configuration and sizing, the caches are flushed by the subroutine 'flush_cache'. Leave this code untouched. As a final attempt to distinguish between R3051 and R3052, cache sizes are used. If you have already hard coded the CPU type, you may delete this portion of the code as well.

Initialize device table (R30xx)

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 (the 79S134 board also has two additional 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. Near the end of this subroutine is a call to install a centronics port driver. Delete this line unless you absolutely wish to have your minimal version of IDT/sim support centronics parallel port.

Initialize IDT/sim to known state (R30xx)

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.

A call is made to subroutine 'init_memory'. The first step in this subroutine is to invalidate the TLB. If you know that your target CPU does not have a TLB (is not an 'E' type device), you may delete this portion of the code. 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/except.c'.

After assigning memory size, the next few lines of code set up global variables related to the floating point unit. If you are not using the R3081, you may delete these lines.

The next few lines, enclosed within the pair of `#ifdef INET` - `#endif`, should be deleted for the sake of clarity, even though they are only conditionally compiled. These lines initialize the board's ethernet ports, and it is not necessary for a minimal version of IDT/sim to address ethernet interface issues.

Initialize command table (R30xx)

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' (defined in file SIM3000/main.c) as it is.

Clear breakpoints (R30xx)

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

SIM4000/csu_idt.S

Note: While reading this section, please refer to a print-out of the 'SIM4000/csu_idt.S' file.

The first subroutine in the file is *start*. At the beginning of *start*, there are a series of 112 *jump* instructions. This is a PROM entry point table, which provides entry points into IDT/sim code that pertains to 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, which is provided so that a user program running out of RAM can simply use these entry points instead of linking run-time library code that may occupy large amounts of RAM space.

Because the entry points and the actual functions are programmed into PROM and execute from the PROM as well, linking C functions through the PROM entry point table will result in slower code. However, where compactness of user code is more important than speed, the entry point table is made available to the user. The prom entry point table is not required for basic functionality of IDT/sim, and the entire table may be removed or commented out. When deleted, your code will start at the label *idtstart*.

When the R4xxx processors came into existence, the 'SIM4000/csu_idt.S' file was created based on the 'SIM3000/csu_idt.S' file. For this historical reason, there are still a number of conditional compile directives such as *#if defined (CPU_R3000)* in the 'SIM4000/csu_idt.S' file. Because this file is used only for boards that are based on R4xxx, these conditional compile directives, and the code activated by them, are redundant. Within this build environment, for clarity and code readability, all lines of code that are compiled if *CPU_R3000* is defined can be deleted.

RAM accessibility (R4xxx)

After initializing the *Status* and *Cause* registers of CP0 to reasonable default values, the next crucial element is to test the accessibility of on-board RAM. A simple data pattern test is performed where *hexadecimal AAAA5555* is written to the base address of the RAM, and a zero is written to the next address. The base address location is read back and the data read is compared with *hexadecimal AAAA5555*.

If the data does not match, it's determined that 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). If the first test passes, a second test is performed with the data pattern of -1. If the test fails, at the early stages of porting, or on a more permanent basis, it may be helpful to display a message or have an LED display indicate that a RAM error has occurred.

Once RAM is determined to be accessible, a number of important tasks are performed. Using a 'SIM4000/csu_idt.S' file example, let us review the tasks in the sequence in which they are undertaken, to evaluate which of them are important and should be ported to a minimal version of IDT/sim.

Subroutine 'initmem' (R4xxx)

After testing memory accessibility, some memory initialization is done to facilitate cache testing. Next the code makes a call to the function 'initmem' which initializes memory.

The first task in the 'initmem' subroutine is to set the entire 'bss' section to zero. The 'bss' section (reference: C language) holds uninitialized data. C programmers will expect the 'bss' section to be zeroed out prior to code execution. However, unless your port of IDT/sim specifically makes use of the fact that all uninitialized data is set to zero at the outset, you do not need this portion of the code.

A stack pointer is set up next, and the 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 in the file 'header/ldtmon.h.'

Next, the PRID register of CP0 is read and a variable to indicate the type of CPU is set to a unique value. If you know the target you will be working with, you may delete all of these tests and simply set the variables to known values. This will make the code inflexible and you will not be able to change CPUs on your board, so please exercise caution. A test for the presence of a floating point unit follows. All R4xxx have a floating point unit, and you may delete this test.

Next, a certain amount of cache related work is done. First the sizes of both data and instruction caches are determined and stored in variables for future reference. This is done in the subroutine 'config_cache' which calls the subroutine '_size_cache.' Both subroutines are defined in the file 'common/idtc4000.S.' If you know the CPU type that you will be using, the cache sizes are also known, and you can simply set the appropriate variables (*dcache_size*, *icache_size*, *scache_size*, *icache_linesize*, *dcache_linesize*, *scache_linesize*) to the known values. The rest of the code that determines the cache sizes can be deleted. After performing cache configuration and sizing, the caches are flushed by the subroutine 'flush_cache.' This code should remain untouched.

Initialize device table (R4xxx)

The function '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. If your board does not have the second serial i/o port 'tty1,' you may delete it from the table.

The initialize device table is defined as 'device_init[]' in the 'common/idtconf.c' file. Verify that the i/o base address declared in the table matches the address on your board. In the same file is the definition of the 'init_dev_tab' subroutine. Near the end of this subroutine is a call to install a centronics port driver. Delete this line unless you want the minimal version of IDT/sim to support centronics port.

Initialize IDT/sim to known state (R4xxx)

The next step is to assign known legitimate values to some of the system variables used by IDT/sim and to initialize some of the remaining hardware.

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/except.c'.

The next line, enclosed within the pair of `#ifdef INET - #endif`, should be deleted for the sake of clarity even though it is only conditionally compiled. This line deals with initializing the ethernet ports. A minimal version of IDT/sim certainly should not be dealing with ethernet interface issues even accidentally!

Initialize command table (R4xxx)

The call to subroutine 'init_cmd_tab' copies the command table from ROM space to RAM space for faster access. The 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, majority of these commands may be disabled to reduce the size of IDT/sim. The topic of which commands to disable is addressed in the next chapter. At this point, leave the call to 'init_cmd_tab' (defined in file SIM4000/imap.c) as it is.

Clear breakpoints (R4xxx)

Leave this call untouched. During the testing phase, even a minimal version of IDT/sim may need low-level debugging facilities of IDT/sim. This completes the treatment of the start-up file SIM4000/csu_idt.S



Minimum IDT/sim User Commands

Notes

Command table

The size of IDT/sim code largely depends upon the number of user commands it supports. As shown in the previous chapter, '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 'SIM3000/MAIN.C' file for R30xx based boards and in the 'SIM4000/MAIN.C' file, for R4xx based boards. When an IDT RISC 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's 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 'MAIN.C.' While creating a minimal version of IDT/sim, begin with this table.

Commands not required for minimal IDT/sim versions

Table 4.1 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 the 'MAIN.C' file. In general, a single file contains the code that supports a single command. Consequently, once you delete a command from the table, you may also remove the line(s) that correspond to compiling and linking of the related 'Makefile' used to build the minimal version of IDT/sim.

The list 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 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 'MAIN.C'. For networking commands such as 'boot', where many files in 'SIM4000/net/' are involved, it is easier to just modify the 'Makefile' by simply removing each occurrence of the global define '-DINET' from the 'Makefile'.

Command	File name	Function
asm	common/IDTCMD1.C	asm_cmd()
	SIM3(4)000/DISASM.C	
benchmark (bm) This command is only valid in S381 target	common/IDTBK.C	benchmark()
	S381/S68681CO.C	timer_start(), timer_stop()
	common/SONICFNS.S	
	TIMERS/timer_t.c	
	TIMERS/r4Ksonic.S	
boot	net/*.*	delete '-DINET' from Makefile
call (ca)	common/IDTCMD2.C	callcmd()
	common/EXCEPT.S	do_call()
dbgint (di)	common/IDTCMD2.C	ri_sel()
debug (db)	common/IDTDEBUG.C	

Table 4.1 Commands not required for minimal IDT/sim versions (Page 1 of 2)

disable	common/idthfio.c	
dt	common/idthbrk.c	dump_trace()
enable	common/idthfio.c	
env	common/p4000cmd.c	
gotill (gt)	common/idthbrk.c	gotill()
history (h)	common/icli.c	linebuf[], hist_init(), history()
ping	net/*.*	delete '-DINET' from Makefile
setenv (set)	(S381, S134, S465)/p4000rtc.c	
term (te)	common/idthcmds1.c	em(), to_transpar(), from_transpar()
t	common/idthbrk.c	trace_cmd()
tc	common/idthbrk.c	cond_trace_cmd()
tex	common/idthbrk.c	excl_cmd(), exc_init()
ts	common/idthbrk.c	stop_trace()
unsetenv	(S381, S134, S465)/p4000rtc.c	

Table 4.1 Commands not required for minimal IDT/sim versions (Page 2 of 2)



Adding & Deleting User Commands

Notes

Depending on the target board, the command table is contained either in the source file 'SIM3000/main.c' or 'SIM4000/main.c'. All commands are decoded through this table. To add or delete a command, the user must either remove or add an entry to 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(s)" 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. First the table from SIM3000/main.c is listed. Next the table from SIM4000/main.c is listed and this also applies to R32364 based boards.

SIM3000

```
CONST static struct command_tab command_tab[] = {
    { "asm", "", asm_cmd, NULL, "assemble:\tasm ADDR" },
```

```
#ifdef INET
```

```

    { "regsel", "rs", rs_sel, NULL, "reg set select:\tregsel|rs [-c|-h]" },
    { "setbaud", "sb", setbaud, NULL, "setbaud:\tsetbaud|sb [CHAR_DEVICE]" },
    { "search", "sr", search_cmd, NULL,
        "search:\t\tsearch|sr [-w|-h|-b] RANGE value [MSK]" },
    { "seg", "", select_seg, s_s_init, "select seg.:\tseg [-0|-1|-2|-u]" },
#if defined(INET) && !defined(NO_NVRAM)
    { "setenv", "set", com_setenv, NULL, "setenv var:\tsetenv|VARVALUE" },
#endif
    { "step", "s", single_step, NULL, "step:\t\tstep|s [COUNT]" },
    { "sub", "sub", sub, NULL, "sub:\t\tsub [-w|-h|-b|-l|-r] ADDRESS" },
    { "term", "te", em, NULL, "terminal emul.:\tterm|te" },
    { "tlbdump", "td", tlbdump, NULL, "tlbdump:\t\ttlbdump|td [RANGE]" },
    { "tlbflush", "tf", tlbflush, NULL, "tlbflush:\t\ttlbflush|tf [RANGE]" },
    { "tlbmap", "tm", tlbmap, NULL,
        "tlbmap:\t\ttlbmap|tm [-i INX] [-(v/d/g/n)] VADDR PADDR" },
    { "tlbpid", "ti", tlbpid, NULL, "tlbpid:\t\ttlbpid|ti [PID]" },
    { "tlbptov", "tp", tlbptov, NULL, "tlbptov:\t\ttlbptov|tp ADDR" },
    { "t", "", trace_cmd, NULL,
        "trace:\t\ttt [-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, "unbrk:\t\tunbrk|ub BPNUMLIST|ALL" },
#if defined(INET) && !defined(NO_NVRAM)
    { "unsetenv", "unset", com_unsetenv, NULL, "unsetenv var:\tunsetenv|unset VAR" },
#endif
    { "wc", "wc", do_wc, NULL,
        "write cache:\twc [-i][-w|-h|-b] <RANGE> [value_list]" },
    { 0, 0, 0, 0, "" }
};

```

SIM4000 & R32364-based

```

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, "breakpoint:\tbrk|b [ADDRLIST]" },
#ifdef INET
    { "boot", "", boot_cmd, NULL, "boot via tftp:\tboot [-n] [[HOST:]FILE]" },
#endif
#ifdef defined(CPU_R32364)
    { "diag", "dg", do_diag, NULL, "diagnostics:\tdiag|dg" },
#endif
#ifdef defined(CPU_R4000)
    { "cacheflush", "cf", cacheflush, NULL, "cacheflush:\tcacheflush|cf [-i|-d|-s|-n]" },

```

```
#endif
    { "call", "ca", callcmd, NULL, "call:\t\tcall|ca ADDR [ARGLIST]" },
    { "checksum", "cs", chk_sum, NULL, "checksum:\tchecksum|cs [startaddr byte_count]" },
    { "compare", "cp", compare_cmd, NULL,
        "compare:\tcompare|cp [-w|-h|-b] RANGE destination" },
    { "cont", "c", cont, NULL, "continue:\tcont|c" },
    { "dbgint", "di", ri_sel, NULL, "debug int.:\tdbgint|di [-e|-d][DEVICE|Int. Line]" },
    { "debug", "db", idebug, NULL, "debug (remote):\tdebug|db [DEVICE]" },
    { "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]" },
    { "dt", "", dump_trace, NULL, "dump trace:\tdt" },
#endif __mips > 2
    { "dump", "d", dump, NULL, "dump:\t\ttdump|d [-d|-w|-h|-b] [RANGE]" },
#else
    { "dump", "d", dump, NULL, "dump:\t\ttdump|d [-w|-h|-b] [RANGE]" },
#endif
#endif
#endif !defined(NO_NVRAM)
#endif defined(P4000) || defined(S465) || defined(S364) || defined(S134)
    { "env", "", com_env, NULL, "env display:\tenv" },
#endif
#endif
#endif __mips > 2
    { "fill", "f", fill, NULL, "fill:\t\ttfill|f [-d|-w|-h|-b|-l|-r] RANGE [value_list]" },
#else
    { "fill", "f", fill, NULL, "fill:\t\ttfill|f [-w|-h|-b|-l|-r] RANGE [value_list]" },
#endif
    { "fr", "", fill_reg, NULL, "fill regs:\tfr [-s|-d] <reg#|reg_name><value>" },
    { "go", "g", go, NULL, "go:\t\ttgo [-n] [INITIAL_PC]" },
    { "gotill", "gt", gotill, NULL, "go untill:\tgotill|gt <brk addr>" },
    { "help", "?", help, NULL, "help:\t\tthelp? [COMMAND(S)]" },
    { "history", "h", history, hist_init, "history:\thistory|h" },
    { "init", "i", prominit, NULL, "initialize:\tinit|i" },
#endif INET
    { "load", "l", load, NULL, "load:\t\ttload|l [-t][-b][-s][-a] DEVICE|FILE" },
#else
    { "load", "l", load, NULL, "load:\t\ttload|l [-b][-s][-a] DEVICE" },
#endif
    { "move", "m", movecmd, NULL, "move:\t\ttmove|m [-w|-h|-b] RANGE destination" },
    { "next", "n", next, NULL, "next:\t\ttnext|n [COUNT]" },
#endif INET
    { "ping", "", ping_cmd, NULL,
        "ping net host:\tping [-Rdnqr] [-c count] [-i wait] [-s size] HOST" },
#endif
```

```

    { "rad", "", select_base, s_b_init, "radix select:\trad [-o|-d|-h]"},
#if defined(CPU_R4000)
    { "rc", "rc", do_rc, NULL, "read cache:\trc <RANGE>" },
    { "lockcache", "lc", lock_cache, NULL, "lock cache:\tlc [-i] [-d]"},
#endif
    { "regsel", "rs", rs_sel, NULL, "reg set select:\tregsel|rs [-c|-h]"},
    { "setbaud", "sb", setbaud, NULL, "setbaud:\tsetbaud|sb [CHAR_DEVICE]" },
    { "search", "sr", search_cmd, NULL, "search:\t\tsearch|sr [-w|-h|-b] RANGE value [MSK]" },
#if defined(CPU_R4000)||defined(CPU_R32364)
    { "seg", "", select_seg, s_s_init, "select seg.:\tseg [-0|-1|-s|-3|-u]"},
#endif
#if defined(S134)
    { "isrt", "isrt", isr_test, NULL, "isrt:\t\tisrt|isrt" },
    { "smsk", "smsk", intr_smsk, NULL, "smsk:\t\tsmsk|smsk" },
    { "gmsk", "gmsk", intr_gmsk, NULL, "gmsk:\t\tgmsk|gmsk" },
    { "clsr", "clsr", intr_clsr, NULL, "clsr:\t\tclsr|clsr" },
    { "clcs", "clcs", intr_clcs, NULL, "clcs:\t\tclcs|clcs" },
    { "caus", "caus", intr_caus, NULL, "caus:\t\tcaus|caus" },
#endif
#if MEMCFG != SRAM_ONLY
#if MEMCFG == SDRAM_ONLY || MEMCFG == SRAM_N_SDRAM || MEMCFG ==
SDRAM_N_SRAM
    { "creg", "creg", dram_creg, NULL, "creg:\t\tcreg|creg" },
#endif /* MEMCFG == SDRAM_ONLY || MEMCFG == SRAM_N_SDRAM || MEMCFG ==
SDRAM_N_SRAM */
#endif /* MEMCFG != SRAM_ONLY */
    { "cmpr", "cmpr", tmr_cmpr, NULL, "cmpr:\t\tcmpr|cmpr" },
#endif /* S134 */
#endif /* (CPU_R4000)|| (CPU_R32364) */
#ifdef SPI
    { "spi256", "spi256", spi256, NULL, "spi256:\t\tspi256|spi256" },
    { "spi_wr", "spi_wr", spi_wr, NULL, "spi_wr:\t\tspi_wr|spi_wr [mem_addr, count, spi_addr]" },
    { "spi_rd", "spi_rd", spi_rd, NULL, "spi_rd:\t\tspi_rd|spi_rd [spi_addr, count, mem_addr]" },
    { "spi_bp", "spi_bp", spi_bp, NULL, "spi_bp:\t\tspi_bp|spi_bp [0 | 1 | 2 | 3 | 4]" },
#endif /* SPI */
#if !defined(NO_NVRAM)
#if defined(P4000) || defined(S465) || defined(S364) || defined(S134)
    { "setenv", "set", com_setenv, NULL, "set variable:\tsetenv|set [VAR [VALUE]]"},
#endif
#endif
    { "step", "s", single_step, NULL, "step:\t\tstep|s [COUNT]" },
    { "sub", "sub", subst, NULL, "sub:\t\tsub [-w|-h|-b|-l|-r] ADDRESS"},
    { "term", "te", em, NULL, "terminal emul.:\tterm|te" },
    { "tlbdump", "td", tlbdump, NULL, "tlbdump:\ttlbdump|td [RANGE]" },
    { "tlbflush", "tf", tlbflush, NULL, "tlbflush:\ttlbflush|tf [RANGE]" },
    { "tlbmap", "tm", tlbmap, NULL, "tlbmap:\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:\t\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, "unbrk:\t\tunbrk|ub BPNUMLIST|ALL" },
#if !defined(NO_NVRAM)
#if defined(P4000) || defined(S465) || defined(S364) || defined(S134)
    { "unsetenv", "unset", com_unsetenv, NULL, "unset variable:\tunsetenv|unset VAR" },
#endif
#endif
    { 0, 0, 0, 0, "" }
};

```



Adding & Deleting IDT/sim Device Drivers

Notes

Overview

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 tables '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 79S134 board) 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 */
    },
#ifdef(S134)
    {
        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 } /* 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 goes for ity0 and ity1 in the 79S134 board.

Actual Device Initialization Table

```
CONST struct dev_init_tab device_init[] = {
    {
        "tty0", /* name of device */
        "console", /* dev. description */
        "tty", /* driver name */
    },
    {
        "tty1", /* name of device */
        "console", /* dev. description */
        "tty", /* driver name */
    },
    { 0,0,0 } /* null entry to mark end of table */
};
```



```

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 */
},
#ifdef(S134)
{
"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 }
};          /* 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.

Example Device Driver

The listings for a centronics parallel interface driver are shown below. This driver may be compiled and linked to run out of RAM on a target board with the hardware to support this option. When the user invokes this downloaded driver it will make calls to IDT/sim through the entry points to install the driver. The user may then return to IDT/sim running out of EPROMS and use the Centronics driver (such as the command load cen<cr> - which would attempt to download an S-record format file to the target).

Source File cendrvr.c

```
#include "idtio.h"
#include "idt_entrypt.h"
#include "centron.h"

int cenopen();          /* rtn to open centronics port */
int cenread();          /* rtn to read centronics port */
int nulldrv();          /* Null device driver rtn */

CONST struct dev_init_tab centron_init[] = {
    {
        "cen",                /* centronics device */
        "centronics",        /* dev. description */
        "centron",           /* driver name */
        0,                   /* controller number */
        0,                   /* unit number */
        0,                   /* partition */
        0                    /* io base address */
    },
    {0,0,0,0,0,0,0}
};

CONST struct dev_sw_tab centron_table[] = {
    {
        cenopen,             /* device open routine */
        nulldrv,             /* device close routine */
        cenread,             /* device read routine */
        nulldrv,             /* device write routine */
        nulldrv,             /* device init routine */
        nulldrv,             /* device strategy routine */
        nulldrv,             /* device ioctl routine */
        "centron"            /* device driver name */
    },
    {0,0,0,0,0,0,0,0}
};

/*
** When the user prg starts up, this routine is called
** to install the drivers for the centronics parallel
** download port.
*/
install_centron()
{
    install_new_dev(centron_table,centron_init);
}

/*
** Null device driver
*/
int nulldrv()
{
    return(0);
}
```

```

/*
** cenopen -- initialize centronics interface
*/

cenopen(io)
    int *io;
{
    cenempty();      /* Empty anything in the fifo */
    return(0);
}

/*
** cenread-- perform read operation
** This routine uses a circular buffer between the actual
** device char get routine (cengetc) and the users request
** for input.
*/

cenread(io, buf, count)
    int *io;
    char *buf;
    int count;
{
    int ocnt;

    ocnt = count;

    while (count > 0) {
        if(!(censtat() & CEN_RXRDY)) /* chk fifo empty */
            return(ocnt - count);

        *buf++ = cenin();           /* char to users buf */
        count--;                    /* go get another */
    }

    return(ocnt);
}

```

Source File centron.s

```

#include "idtcpu.h"
#include "iregdef.h"
#include "centron.h"

        .set        noreorder

/*
** _start - entry point for installing centronics driver
*/

_start::
    mfc0    t1,C0_SR
    nop
    nop
    or      t1,0x40000000    /* set cp2 usable bit */
    mtc0    t1,C0_SR
    nop
    jal     install_centron
    nop
    j       _exit
    nop

/*
** cenempty - make sure the fifo is empty
*/

```

```

        .globl      cenempty
cenempty:
    mfc0    t1,C0_SR
    nop
    nop
    or      t1,0x40000000
    mtc0    t1,C0_SR
    nop
    li      t0,0xa0600000
1:
    bc2f    2f          /* If empty jump to return */
    nop
    nop
    lbu     zero,0(t0)   /* If not empty read a byte */
    nop      /* Throw byte away */
    nop
    b       1b          /* Go check if another byte */
    nop
2:
    j       ra
    nop
/*
** input routine - waits for a character to be put in fifo
** and then reads it out and returns the character
*/

```

```

        .globl      cenin
cenin:
    mfc0    t1,C0_SR
    nop
    nop
    or      t1,0x40000000
    mtc0    t1,C0_SR
    nop
    li      t0,0xa0600000
1:
    bc2f    1b          /* If empty loop */
    nop
    lbu     v0,0(t0)     /* fetch byte from port */
    j       ra          /* return with byte in v0 */
    nop
/*
**censtat - checks for a character to input
** returns:
**      zero - no character
**      CEN_RXRDY - character in fifo
*/

```

```

        .globl      censtat
censtat:
    mfc0    t1,C0_SR
    nop
    nop
    or      t1,0x40000000
    mtc0    t1,C0_SR
    nop
    bc2t    1f          /* Test the empty flag bit */

```

```

        nop
        j      ra      /* if empty return a zero */
        move   v0,zero

1:
        j      ra      /* else return CEN_RXRDY */
        li     v0,CEN_RXRDY

```

Header file centron.h

```

/*
** Copyright 1989 Integrated Device Technology
** All Rights Reserved
** Header file for centronics interface
*/

```

```

#define NUM_UNIT      1
#define CEN_RXRDY     1

```

Make File Makecen

```

#
# Copyright 1989 Integrated Device Technology, Inc.
# All rights reserved.
# Makefile for Centronics driver (assumes IDT/c compiler)
# Assumes R4700 in Little Endian mode

IDT=/IDTC7.0
LIBS=$(IDT)/R4600/BE/MIPS3
INCDIRS=$(IDT)
CC2= gcc -nostdinc $(INCDIRS) -G 0 -EL -g -c
AS2= gcc $(INCDIRS) -G 0 -EL -g -o
OBS= crt0.o centron.o cendrvr.o

all: centron

crt0.o: $(IDT)/idt_csu.S
        $(AS2) crt0.o idt_csu.S

centron.o: centron.S iregdef.h idtcpu.h
        $(AS2) centron.o centron.S

cendrvr.o: cendrvr.c
        $(CC2) cendrvr.c

centron: $(OBS)
        -rm -f centron centron.map centron.nm
        gcc -nostdlibs -M -N -G 0 -o centron $(OBS) -L$(LIBS) -lnk > centron.map
        size -x centron
        nm -n -x centron > centron.nm
        objcopy -O srec centron centron.sre

```




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, and 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 the 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 types:

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 types:

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' type:

```
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.

Compare

The 'compare' command compares a word data pattern with all words in a range of word 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.

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 7.1:

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 7.1 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 & 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

Notes

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 use of the '**diag**' command varies 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 10. The only exception is the SPI command set for the 79S134 evaluation board; these commands are not a part of the '**diag**' command but are nevertheless included in this chapter because of what they do. More on this later in the chapter.

List of available tests

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

- ◆ *Run All Tests*
- ◆ *Memory Test*
- ◆ *Cache Test*
- ◆ *System Test*
- ◆ *MMU Test*
- ◆ *FPU Test*
- ◆ *Set Options*
- ◆ *History*
- ◆ *Loop On All Tests*
- ◆ *Help*
- ◆ *Quit*

Following additional tests are offered only for the **R36100** processor.

- ◆ *Cache Locking Test*
- ◆ *Timer Write Compare Ack Test*
- ◆ *PIO Test*
- ◆ *Parallel Test*
- ◆ *Serial Test*
- ◆ *DMA Test*

Several of the following additional tests are offered for the **79S134** board depending on its configuration:

- ◆ *SRAM Test*
- ◆ *DRAM Test*
- ◆ *DMA Test*
- ◆ *PIO Test*
- ◆ *Interrupt Test*
- ◆ *Internal UART Test*
- ◆ *Timers' Test*

Note: Please see additional information about diagnostics for SPI on the 79S134 board, at the end of this chapter.

Tests common to all IDT boards

Memory Test

The 'Memory Test' checks the main memory and will isolate 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 and 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.

MMU Test

The 'MMU test' verifies that the Translation Lookaside Buffer (TLB) is working correctly. The register file making up the 64 entries in the TLB is tested for all bits being able to be written and read correctly. The functionality of the TLB is tested by mapping each virtual page to a physical page and verifying that the virtual address may be used without generating a TLB miss exception. The case where a virtual address is accessed and is not mapped, is also checked (forces an exception). The four control bits (V-Valid, G-Global, D-Dirty and N-Non- cached) are also checked.

FPU Test

This tests the functionality of the FPU by executing a set of both single and double precision arithmetic operations. All of the IEEE standard exceptions are also tested for and the control bits for enable/disable exceptions and the sticky bits are exercised. This test also verifies that the FPU (R3010) can interrupt the CPU (R3000). The default external interrupt line for the FPU is external interrupt line 3. This may be changed by the user if the specific target is connected to a different interrupt line.

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, the active tag lines and the FPU interrupt line number.

Tests specific to R36100

Cache Locking Test

In this test, the data cache is first partitioned. Data is written to one partition. It is then locked. An attempt is made to corrupt that partition by writing to it repeatedly. Finally the locked partition data is read back and verified for not having been corrupted.

Timer Write Compare Ack Test

The on-chip timer is set up and allowed to cause an interrupt. Once the interrupt is detected, an attempt is made to clear the timer interrupt bit, and if successful, the test is declared as passed.

PIO Test

A loop-back connector is required for this test. Predefined data patterns are written to the PIO port, read back from the port and compared for integrity.

Parallel Test

The Bi-Centronics parallel port protocol is tested through an interactive test in which the user is guided with easy to follow instructions at each step.

Serial Test

This test is not yet implemented.

DMA Test

Memory to memory DMA data transfer is tested.

Tests specific to IDT79S134 board**SRAM test**

A simple test of validity of SRAM is performed. A pattern is written to each word of SRAM from address 0xA008DF90 to 0xA00FFFFC. The contents of these locations are then read back and compared with the original data.

DRAM test

Similar to the SRAM test, only this one 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 Mega-Byte 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 on the 79S134 can transfer data between:

1. memory and memory
2. memory and I/O device
3. memory and PCI device
4. PCI device and I/O device
5. PCI device and PCI device
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 0xA009000 to 0xA0009500 using memory to memory transfer capability. The transfer is verified and a "Passed" or "Failed" status is reported.

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

PIO test

The PIO Controller is a shared device in the 79S134 evaluation board. For verification of its functionality, a LED and the Microwire Serial EPROM devices is used.

LED (siemens) test: The functionality of the LED is implicitly verified when IDT/sim boots up, because a number of messages get displayed at various stages of the boot up process. However, during the PIO test, the LED is expected to display the characters "AAAA" if the test is successful.

Serial EPROM (Microwire) Validation: A pattern is written into the Serial EPROM and read back for verification. "Passed" or "Failed" status is reported on the console.

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.

Internal UART test

This is a loop test. The UART is programmed to link the transmitter of an internal UART to the receiver of the same internal UART. Test datum is written to the transmitter and the receiver of the same UART is immediately read back. The written and read data are compared and a "Passed" result is declared if they are identical.

Timers test

There are eight timers in 79S134. Of these, five are dedicated timers and three are general purpose timers. Of the three general purpose timers, timers #0 and #1 can be cascaded together through external connection. The remaining general purpose timer, timer #2, can be used as a time-slice clock.

Minimal testing of four out of the eight timers is performed. The timers that are tested are the three general purpose 32-bit timers and one dedicated timer. The dedicated timer tested is the DRAM refresh timer. The remaining timers are not tested because they can be disruptive. These timers that are not tested, are the watchdog timer, the CPU bus timeout timer, the IP bus timeout timer, and the warm-reset timer.

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.

SPI tests for 79S134 board

Spi256k, a software implementation of an SPI interface is available on the 79S134 board. The software is in the form of a driver that is added to the IDT/sim for the 79S134 board. Spi256k drives the Atmel 25256 EEPROM on the 79S134 board. This board demonstrates the combination of the 79RC32364 RISController processor chip combined with the 79RC32134 System Controller Chip. A hardware SPI interface is provided on the 32134 SCC but it will not operate with the Atmel 25256 EEPROM. Spi256k adds support for the Atmel EEPROM and provides a template for users to develop additional device drivers that use the PIO ports. The tests for this interface are NOT a part of the "diag" command menu as the rest of the commands explained in this chapter. The tests for the SPI are part of the main menu of IDT/sim itself. The information on SPI commands is added to this chapter only because of the fact that they are also diagnostic in nature.

The SPI test commands are: **spi256**, **spi_wr**, **spi_rd** and **spi_bp**

spi256

This is a test function that clears the protection level and rewrites every byte of the EEPROM. First the status register is written to modify the protection levels of the device. If the correct value is not read back, an error message is printed out and spi256 returns. If the status register can be changed then protection is removed and 16K random values are written to the EEPROM. The values are then read back. If the returned values do not compare with the values written then an error message is displayed.

spi_wr

`spi_wr(mem_addr, count, spi_addr)`

This test writes count bytes from a memory address to the serial EEPROM.

spi_rd

`spi_rd(spi_addr, count, mem_addr)`

This test reads count bytes from the serial EEPROM to a memory address.

spi_bp

spi_bp(protection_level)

This test changes the protection level. It calls Spi256kBlockProtect. The values of the parameter are:

- ◆ 0 - *unprotect the device*
- ◆ 1 - *protect the highest 25% of memory*
- ◆ 2 - *protect the upper 50% of memory*
- ◆ 3 - *protect all of the memory*
- ◆ 4 - *display the current protection level, does not change the protection level*

Typical SPI256 Test Process

```

<IDT>load -a tty0
<IDT>g a0800000
<IDT>fill a0c00000/100 0xdeadbeef// fill memory with write pattern
<IDT>fill a0c00400/100 0           // clear read destination memory
<IDT>spi_bp 3                     // protect EEPROM memory
<IDT>spi_bp 4                     // status should be 0C, 2 bits set
<IDT>spi_wr a0c00000 256 0        // ABORTS
<IDT>spi_wr a0c00000 256 2000    // ABORTS
<IDT>spi_bp 0                     // clear protection
<IDT>spi_wr a0c00000 256 c000    // ABORTED - beyond memory range
<IDT>spi_wr a0c00000 256 2000
<IDT>spi_rd 2000 256 a0c00400
<IDT>d a0c00400/100             // should be all 0xdeadbeef

```




IDT/sim PROM Entry Points

Notes

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 through linking the library 'liblnk.a' while building the application to run on the target board. The library is supplied with IDT/C and can be linked using the compile/link command line option '-lnk.' The source code for the library is a part of the IDT/kit.

Another method of accessing these PROM functions is to include the file 'idtlink.S' (and its header file 'idt_encrypt.h') in the build process of the application program. The files are available as a part of the source code of IDT/sim.

With either method, 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 9.1 contains a comprehensive list of all entry points in IDT/sim.

Entry #	Name	Description
0	reset	run diags/reinit
1	n/i	
2	restart	reenter monitor cmd loop - same as #17
3	reinit	reinit then cmd loop
4	reenter_mon	in current mode
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	_exit	return to monitor init - same as #2
18	rfileinit	initialize remote file access
19	ropen	open remote file
20	rclose	close remote file
21	rread	read from remote file
22	rwrite	write from remote file
23	rseek	seek remote file
24	rprintf	remote printf (stdout)

Table 9.1 PROM Function Entry Points (Page 1 of 3)

Entry #	Name	Description
25	rgets	remote gets (stdin)
26	n/i	not implemented
27	n/i	not implemented
28	flush_cache	flush all caches completely
29	clear_cache	flush portion of a cache
30	setjmp	save stack state
31	longjmp	restore stack state
32	exc_utlb_code	utlbmiss boot vect (R30xx only)
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
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	start timer
45	timer_stop	stop timer & return time
46	n/i	not implemented
47	n/i	not implemented
48	exc_norm_code	general exc boot vec(R30xx)
49	printf32	formatted print for 32-bit code for R4xxx
50-54	n/i	not implemented
55	get_mem_conf	Get memory configuration
56	set_mem_conf	Set memory configuration
57-63	n/i	not implemented
for R30xx target:		
64	install_command	user installed command
for R4xxx target:		
64	exc_tlb_code	R4xxx TLB exc boot vector
for all targets:		
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
for R4xxx targets:		
68	install_command	user installed command
for ethernet support:		
69	tftpopen	open TFTP file
70	tftpclose	close TFTP file

Table 9.1 PROM Function Entry Points (Page 2 of 3)

Entry #	Name	Description
71	tftpread	read TFTP file
72	tftpwrite	write TFTP file
73	tftplseek	seek TFTP file
74	soc_syscall	Indirect socket "system call"
75	setjmp64	save stack state for MIPSIII/MIPSIV
76	longjmp64	restore stack state for MIPSIII/MIPSIV
77-79	n/i	not implemented
80	exc_xtlb_code	R4000 XTLB exc boot vect
81-95	n/i	not implemented
96	exc_cache_code	R4000 cache exc boot vect
97-111	n/i	not implemented
112	exc_norm_code	R4000 general exc boot v.

Table 9.1 PROM Function Entry Points (Page 3 of 3)

EXIT

```

void _exit();
#include <idt_entrypt.h>
main(argc,argv)
    int argc; char **argv;
{
    /* <== user specific code here */
    _exit();      /* return to prom monitor */
}

```

ATOB

```

char *atob(str,intptr,base)
    char *str;          /* pointer to str */
    unsigned *intptr;   /* pointer to loc for result */
    unsigned base;      /* radix - 8,10,16 */

    0o      Octal
    0d      Decimal
    0x      Hexadecimal

char *atob();
void
a_user_prog()
{
    char *charptr = "1234AB";
    unsigned rsltptr;
    charptr = atob(charptr, &rsltptr, 16,0xa0000000);
}

```

CLEAR_CACHE

```

void clear_cache(begin_addr,num_bytes)
    unsigned int begin_addr;
    int num_bytes;

    li      a0,0x80000000    /* begin_addr */
    jal     clear_cache     /* args are passed in a0 and a1 */
    li      a1,12*4         /* length of vector code is 12 words
                             /* loaded in branch delay */

```

CLI

```
void cli(cmd_table,prompt)
    struct command_tab *cmd_table;
    char *prompt;

icli.h
```

CLOSE

```
int close(fd)
    int fd;

int is_closed;

is_closed = close(stdin);
```

EXC_CACHE_CODE

```
exc_cache_code();
```

EXC_NORM_CODE

```
exc_norm_code();

la      k0,(0xbfc00000 + (48 * 8))
j       k0
```

Note: If the above method is not used, continuing from a breakpoint will not be possible.

```
exc_utlb_code();
```

EXC_UTLB_CODE

```
exc_utlb_code();

la      k0,(0xbfc00000 + (32 * 8))
j       k0

exc_norm_code();
```

EXC_XTLB_CODE**FLUSH_CACHE**

```
flush_cache();
```

GET_MEM_CONF

```
int get_mem_conf(mcptr)
    mem_config *mcptr;

typedef struct {
    unsigned int mem_size;
    unsigned int icache_size;
    unsigned int dcache_size;
#if defined(CPU_R4000)
    unsigned int scache_size;
#endif
} mem_config;

set_mem_conf();

idtmon.h
```

GET_RANGE

```
int get_range(range_ptr,start,end)
    char *range_ptr;
    unsigned *start;
    int *end;
```

```
base_address/count
base_address-end_address
icli.h
```

GETCHAR

```
int getchar();
putchar(),
```

GETS

```
char *gets(str);
char *str;
```

HELP

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

INSTALL_COMMANDS

```
install_commands(cmd_table)
struct command_tab *cmd_table;

#include <icli.h>

int do_diag(); /* declare do_diag a function returning int */

struct command_tab diag_table[] = {
    {"diag", "dg", do_diag, NULL, "diagnostics:\tdiag|dg" },
    {0, 0,0,0, ""}
};

user_init_code()
{
    install_commands(diag_command);
}
```

INSTALL_IMMEDIATE_INT

```
install_immediate_int(ptr_user_int_rt)
int (*ptr_user_int_rt)();

/* end of exception/interrupt code without hooks into or being called by the IDT prom monitor. It
is assumed that AT points to the register save area. This example is R30xx specific.
*/

lw      k0,R_EPC(AT)      /* get epc contents into k0 */
lw      v0,R_V0(AT)       /* restore the contents of v0 */
lw      AT,R_AT(AT)       /* restore the contents of AT */
j       k0                /* return to normal processing */
rfe                      /* execute rfe in branch delay slot */

/* The above code sequence assumes that the registers were saved on entry to the exception/
interrupt processing routine and also that the EPC register contained the program counter of the
next instruction to be executed after the exception is processed.
*/

/* end of exception/interrupt code without hooks into or being called by the IDT prom monitor. It
is assumed that AT points to the register save area and that SR has been restored with SR_EXL
set. This example is R4xxx specific.
*/

.set    noreorder
.set    noat
```

```

lw      k0,R_EPC(AT)      /* get epc contents into k0 */
lw      v0,R_V0(AT)       /* restore the contents of v0 */
lw      AT,R_AT(AT)       /* restore AT register*/
mtc0    k0,CO_EPC         /* put exception PC back */
nop
nop
nop
eret

.set at
.set reorder

/* The above code sequence assumes that the registers were saved on entry to the exception/
interrupt processing routine and also that the EPC register contained the program counter of the
next instruction to be executed after the exception is processed.
*/

/* end of exception/interrupt code with hooks into and being called by the IDT prom monitor.
*/

lw      ra,R_RA(AT)       /* restore the contents of ra.
                           This is the return address
                           into the IDT prom monitor.
                           Register v0 must be
                           set to zero if the user did not
                           process the interrupt
                           and non-zero if the user did process
                           the interrupt
                           */

j      ra
lw      AT,R_AT(AT)

/* The above code sequence assumes that the registers were saved on entry to the exception/
interrupt processing routine. The jump register ra will return to the monitor's exception handler.
The IDT prom monitor code will then restore any state information necessary and will return to
the interrupted code via the contents of the epc register and will execute the rfe instruction for the
user.
*/

install_normal_int();

```

INSTALL_NEW_DEV

```

install_new_dev(dt_ptr,di_ptr)
    struct dev_sw_tab *dt_ptr;
    struct dev_init_tab *diptr;

open(),close(), read(), write(), ioctl()

```

INSTALL_NORMAL_INT

```

install_normal_int(ptr_user_int_rt)
    int (*ptr_user_int_rt)();

#include <idt_entrypt.h>

static int timer_int();

main(argc,argv)
    int argc;
    char **argv;
{
    /* application init code not shown to save space */

    install_normal_int(timer_int);
}

```



```

/* timer_int() - timer interrupt routine - called from monitor
** when an external interrupt occurs.
*/

static int
timer_int()
{
    int rtn_int;
    /* check to see if this is a timer interrupt */
    rtn_int = check_int(sintmask|hsintmask);
    if ((rtn_int & sintmask) != 0) {
        second += 1;          /* interrupt due to sec counter */
        clear_int0();
    }
    if ((rtn_int & hsintmask) != 0) {
        halfsec += 1;        /* interrupt due to half sec counter */
        clear_int1();
    }
    return(rtn_int);
}

/*
** check_int - checks to see if interrupt is one that was expected
** entry: a0 = mask of expected interrupts
** returns: mask of received expected interrupts
*/
FRAME(check_int,sp,0,ra)

    .set        noreorder

    mfc0        v0,C0_CAUSE      /* fetch the cause register */
    nop          /* contains interrupt pending bits */
    and         v0,a0            /* and with expected mask */
    j           ra              /* returns interrupts set in v0 */
    nop

    .set        reorder
ENDFRAME(check_int)

/*
** routine that clears the interrupt signal for the second timer
*/
FRAME(clear_int0,sp,0,ra)

    .set        noreorder

    lw         v0,TIMERCLR0
    j          ra
    nop

    .set        reorder
ENDFRAME(clear_int0)

/*
** routine that clears the interrupt signal for the half second
** timer
*/
FRAME(clear_int1,sp,0,ra)

    .set        noreorder

    lw         v0,TIMERCLR1

```

```
j      ra
nop
```

```
.set      reorder
ENDFRAME(clear_int1)
install_immediate_int();
```

IOCTL

```
ioctl(fd,cmd,arg)
int fd;
int cmd;
int arg;

#define FIOCNBLOCK      ((f<<8)|1)/*set non-blocking io */
#define FIOCSCAN        ((f<<8)|2)/* scan device for input */
#define FIOCINTBRK      ('f<<8)|3)/*enable break interrupt */
#define FIOCINTBRKNOT   ((f<<8)|4)/*disable break intr */
#define FIOCCLINT       ('f<<8)|5)/*clear extrn interrupt */

#define FIOCRAW          (('t<<8)|1)/*don't process i/o*/
#define FIOCFLUSH        (('t<<8)|2)/* flush input*/
#define FIOCREOPEN      (('t<<8)|4)/*reopen to change baud */
#define FIOCBAUD         (('t<<8)|5)/*baud rate change*/
```

OPEN

```
int open(device,flags)
char *device;
int flags;

#define O_RDONLY      0      /* open for reading only */
#define O_WRONLY      1      /* open for writing only */
#define O_RDWR        2      /* open for reading and writing */

struct iocntb {
    char *icb_addr;          /* user buffer address */
    int icb_count;           /* count of char to transfer */
    int icb_blkno;           /* random access block number */
    int icb_erno;           /* return error number */
    int icb_flags;           /* dev. type and status flags */
    struct dev_init_tab *icb_di;
    struct dev_sw_tab *icb_dt;
};

#define F_READ      0x0001      /* file opened for reading */
#define F_WRITE     0x0002      /* file opened for writing */
#define F_NBLOCK    0x0004      /* non-blocking io */
#define F_SCAN      0x0008      /* device should be scanned */
#define F_STRAT     0x0010      /* use strategy routine */
#define F_REMOTE    0x0020      /* set up to gen interrupt */
```

PRINTF/PRINTF

```
printf(format [,arg]...)
char *format;

sprintf(str,format [,arg]...)
char *str;
char *format;
```

PUTCHAR

```
void putchar(c)
char c;
```

PUTS

```
void puts(str)
char *str;
```

READ

```
read(fd, buf, cnt)
int fd;           /* file descriptor */
char *buf;        /* pointer to user buffer */
int cnt;          /* count of chars to read */
```

REINIT

```
reinit();
```

RESET

```
reset();
```

RESTART

```
restart();
```

SET_MEM_CONF

```
typedef struct {
    unsigned int mem_size;
    unsigned int icache_size;
    unsigned int dcache_size;
#ifdef CPU_R4000
    unsigned int scache_size;
#endif
} mem_config;

get_mem_conf();

idtmon.h
```

SETJMP/LONGJMP & SETJMP64/LONGJMP64

```
int setjmp(cur_cntx)
jmp_buf cur_cntx;

int longjmp(cur_cntx, val)
jmp_buf cur_cntx;
int val;

typedef int jmp_buf[JB_SIZE]; /* caller saved regs, sp, pc */

/*
 * jmp_buf indices
 */

#define JB_PC      0
#define JB_SP      1
#define JB_FP      2
#define JB_S0      3
#define JB_S1      4
#define JB_S2      5
#define JB_S3      6
#define JB_S4      7
#define JB_S5      8
#define JB_S6      9
#define JB_S7     10
#define JB_SIZE11
```

SHOWCHAR

```
void showchar(c)
int c;

'\b' '\f' '\n' '\r' '\t'
```

STRING

```
char *strcat(s,t)
char *s;
char *t;

int strcmp(s,t)
char *s;
char *t;

char *strcpy(s,t)
char *s;
char *t;

int strlen(s)
char *s;
```

TFTP_CLOSE

```
int tftpclose(tfd)
int tfd;

tftpopen(), tftpread(), tftplseek()
```

TFTPLSEEK

```
int tftpclose(tfd,offset,method)
int tfd;          /* File descriptor */
long offset;      /* number of bytes to move, in bytes */
int method;       /* 0=from start of file; 1=from current location; others illegal */

tftpopen(), tftpread(), tftpclose()
```

TFTPOPEN

```
int tftpopen(path,flags)
char *path;
int flags;
```

Note: Files may be opened read-only or write-only; read-write is not permitted. In both cases the files must already exist and be publicly readable or writable

TFTPREAD

```
int tftpread(tfd,buf,cnt)
int tfd;
char *buf;
int cnt;
```

TFTPWRITE

```
int tftpwrite(tfd,buf,cnt)
int tfd;
char *buf;
int cnt;
```

TIMER_START

```
timer_start();
timer_stop();
```

TIMER_STOP

```
unsigned int timer_stop() ;
timer_start();
```

TOKENIZE

```
int tokenize(cmdline,argv)
char *cmdline;
struct argv_array *argv;

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

struct argv_array {
    char *argv_ptrs[MAXARGS];
    char der_strings[MAXSTRLNGTH];
};

int argc,i;
struct argv_array argv;
int command_count;

while (1) {
    printf("%s", prompt);
    get_line(linebuf[i]);
    argc = tokenize(linebuf[i], &argv);
}

icli.h
```

WRITE

```
write(fd, buf, cnt)
    int fd;           /* file descriptor */
    char *buf;        /* pointer to user buffer */
    int cnt;          /* count of chars to write */
```

--	--



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 8.

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.' This monitor permits the user to develop stand-alone systems built around any of the MIPS R3000 based CPUs including the R3041, R3051, R3052, R3071, R3081, R4000 based CPUs including R4400, R4600, R4640, R4650, R4700 and other processors and integrated controllers including R5000, R36100, R32364, R64474 and R64475.

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 coprocessors), 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:

Hexadecimal	0xnnnnnnnn
Octal	0nnnnnnnn
Decimal	0nnnnnnnn
Default radix	nnnnnnnn

The user command for selecting a default radix is provided in "Set Default Radix" on page 16.

- ◆ *RANGE* - When a command specifies a RANGE to be entered, it can be entered in one of the following three ways:
 - *start_address-end_address*
 - *start_address/count*
 - *start_address*

Ranges cannot contain embedded blanks. Numbers entered by a user, unless explicitly specified, will be assumed to be the selected default radix. One exception to this is the 'count' which is always in decimal. If no 'end_address' or 'count' is entered, then a count of 20 is assumed.

- ◆ *When entering an 'address' as an argument of a command, there is also the concept of a default segment. In the context of R30xx, user may be referring to one of four memory segments (kuseg, kseg0, kseg1 or kseg2). User may select a default segment such that all addresses entered will be modified appropriately (e.g. default seg = kseg1 and the address entered = 0x1000 would result in an address 0xa0001000). To override the default segment the user must enter all 8 nibbles for the 32 bit address.*
- ◆ *The command line interpreter will provide some shorthand methods to reenter commands. To repeat a command just entered, the user may enter !c...; where 'c...' are the first few characters of the previously entered command. Only enough characters need be entered to uniquely identify the previously entered command. There is also a 'history' command which allows the user to repeat a command by entering: !# where '#' is the number of the command from the history list which is displayed when the 'history' command is entered.*

Command categories

Commands accepted by the IDT/sim monitor are outlined below and are divided into eight groups, for clarity. The groups are:

- ◆ *Communication/Host interface commands*
- ◆ *Execution control commands*
- ◆ *Memory/Register and Assembly/Disassembly commands*
- ◆ *Setup and Environment commands*
- ◆ *TLB commands*
- ◆ *Trace commands*
- ◆ *Network related commands*
- ◆ *Board specific commands*

Communication/Host interface commands

Debug - GDB and IDT/c version 5.0 or later

The user does not enter any command at the IDT/sim monitor prompt to begin source level debugging using the debugger GDB which is a part of IDT/c 5.0 and later versions. GDB running on the host computer will do all the necessary work including initialization over the serial connection. Description of GDB is beyond the scope of this manual. Please refer to the documentation of IDT/C 5.0 or later.

Note: GDB works only on the 'tty0' port of the target board; the serial link to host must be at 'tty0' of the target board.

Debug remotely

```
debug|db [DEVICE]
```

Initialize remote debugging through port defined by 'DEVICE' (e.g. tty0, tty1). This command is obsolete if you are using GDB remote debugger which came with IDT/c 5.0 or later.

Debug Interrupt

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

Enable or disable the debug interrupt. The debug interrupt allows a user to interrupt an application program's execution and return to the monitor. User may specify a DEVICE or a specific interrupt line to generate the external interrupt. The monitor on many of the IDT development boards takes advantage of the fact that the DUART interrupt is connected to external interrupt line 5 on the R30xx. When the argument to the '-e' option is *cons*, *tty0* or *tty1*, the duart is programmed so that when the 'break' key is depressed an interrupt will be generated that will return control to the monitor. An example is shown below.

```
<IDT>dbgint -e cons    OR    <IDT>di -e cons
```

The '-e' specifies that the console interrupt is to be enabled. The second argument for this command may be the special name 'cons' or one of the recognized device names (tty0 or tty1).

When the user specifies 'cons', the interrupt is enabled immediately. When the user specifies a 'device name', the enabling of the external interrupt only takes place when the user enters the remote debug mode. To disable the debug interrupt use the '-d' option.

If the user enters **dbgint** without any arguments, IDT/sim will display one of the following messages:

If the debug interrupt is disabled -

```
<IDT>di
```

Debug int. disabled

```
<IDT>
```

If the debug interrupt is enabled -

```
<IDT>di
```

Debug int. enabled

Interrupt line **n** (Where **n** is 0-5)

```
<IDT>
```

If the user specifies an interrupt line number from 0-5, the monitor just enables the interrupt line specified and it is up to the user to provide a source for the interrupt (i.e. a switch).

Note to R4xxx users: if the R4xxx internal timer is enabled on your board, and if it uses a hardware interrupt line, to prevent unexpected breaks to the monitor, the **dbgint** command should not be used to enable this particular interrupt.

GreenHills debugger

```
ghd -[G|D][B<baudrate>] [DEVICE]
```

This command initializes IDT/sim to enable source level remote debugging using compiler tool-chain. from GreenHills, Inc. For further details, please study GreenHills compiler manuals.

Download Program from Host to Board

```
load|l [-a][-s] <device> (serial or parallel port based download)
```

```
load|l [-t] <FILE> (ethernet based download)
```

This command inputs Motorola S-records from the device specified on the command line. Devices supported depend on the drivers linked or installed with IDT/sim. IDT/sim contains a serial driver that supports two serial devices (tty0 and tty1), and a centronics driver. The 79S134 board also supports two more serial devices called 'ity0' and 'ity1' based on the 79S134 companion device. The 79S465 and 79S381 boards also provide the ethernet driver.

Currently, Motorola S-record is the only supported file format supported. The monitor expects to see 'S2 or S3' type records until a final 'S7' record is received from the host. Currently the serial channel defaults to 9600 baud, 8 bits, one stop bit and no parity. The RTS, CTS, DSR, and DTR hand shake signals on the UART are not used.

The command line options have the following effect:

-a : turns off handshake. Does not send ACK/NAK after receiving each s-record.

-s : silent mode. Does not echo periods (.) to console. Makes the download execute a little faster.

-t : download file using TFTP (ethernet). If the filename contains a non-alphanumeric character, the file is automatically downloaded via TFTP even if "-t" is not used. The entire path of the file including the internet address of the host needs to be specified in the load command. For example:

```
load -t 89.0.3.4:/usr/people/myhome/myfile.sre
```

This command will download a file called myfile.sre from directory /usr/people/myhome on a machine which has the internet address 89.0.3.4. If you define the environment variable \$tftp host, you do not need to specify the host internet address. If the file is in the directory /tftpboot on the host, you do not need to specify the entire path.

Typically, you can set an environment variable to remember the entire address+path+filename and simply include that variable in the load command as follows:

```
setenv f1 89.0.3.4:/usr/people/myhome/myfile.sre
l -t $f1
```

Remember that you need to issue the 'setenv' command only once as the environment variables are stored in the non-volatile RAM even if you power off your board.

Set Baud rate of tty Port

```
setbaud|sb <DEVICE>
```

This command allows the user to select the baud rate for the device specified by DEV. DEV may be either tty0 or tty1 (or ity0/ity1 in 79S134 board). This command is interactive in the sense that the user enters the command and then the monitor will display a baud rate on the next line. If this is the desired baud rate, the user must press carriage-return. If it is not, then the user must press the space bar repeatedly until the desired baud rate is displayed and then press carriage-return.

```
<IDT>setbaud tty1
19200
```

The choices of baud rates wrap around, so if you hit the space bar too many times, continue hitting it until the desired baud rate is displayed again. There is also a "no change" entry that can be selected if you wish to return without changing the baud rate.

Terminal Emulator

```
te
```

The 'te' command puts IDT/sim in a "transparent" mode and connects the console port straight through to another serial port which may be connected to a host computer's serial port or a modem. If it is connected to a modem, a remote host may be dialed up and connected to. The port connected to the host is tty1.

Once in the terminal emulation mode the escape character is ^z (control-z). To exit the terminal emulation mode, the user should enter a ^z followed by the letter 'q'. Downloading a program to the target from a remote host using the terminal emulation mode can be accomplished as follows. On the target, get into emulation mode by entering the command:

```
<IDT> te
```

This will logically connect the host to tty1 port of the box. If tty1 is physically connected to a RISC/os™ host, the user will need to login. After logging on to the host, the user may use the standard UNIX copy command (cp) to copy a program to the target. On the host enter the following command - but do not hit carriage return:

```
cp program.srec /dev/tty3
```

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

To start the download the user must 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

Run User Benchmark

```
benchmark|bm
```

The 'benchmark' command provides an easy mechanism to obtain quick estimates of execution times of user code. The 'benchmark' command measures code execution time and displays that time on the console.

To run a benchmark, first compile and link the code to be benchmarked. Convert the executable file to Motorola S-record format. Download the file to the target board using the 'load' or 'boot -n' command.

At this point one would ordinarily use the 'go' command to execute the code on the target board. Simply use the 'benchmark' command instead. The time measurement starts immediately prior to the start of user's code and stops immediately after user's call to exit(). The execution time (also referred to as elapsed time) is displayed in units of 'minutes, seconds, milliseconds, microseconds'.

Upon entering the 'benchmark|bm' command, the user is prompted for whether the stack needs to be cached or uncached for that particular execution of the benchmark. A response of "y" (for yes) or "n" (for no) is expected.

In certain implementations of timers used for benchmarking purposes, the second channel of an available UART is used to keep time. The baud rate selected for that particular channel determines the accuracy and duration of measurable time. User is offered a choice of baud rates to select for the available channel. In general, the smaller the chosen baud rate, the longer the period of time which can be measured and the lesser the accuracy.

This is typically not a problem. If the benchmark is expected to take 16 minutes, the accuracy in terms of micro or even milliseconds is not of great importance. On the other hand, if the measured time is within a few milliseconds, then even the microseconds matter significantly. Once the user selects a baud rate, the maximum limit of time which can be measured (before time counters roll-over and offer incorrect results) is displayed for user's reference.

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. That is to say, 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 to 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]
```

The 'go' command will begin execution at 'address' if entered, or at the address contained in the coprocessor zero (CPO) exception program counter (EPC). This command should be used to start the initial execution of a program downloaded to the board. The 'go' command clears the client general purpose registers, so it should not be used to continue execution once execution has been started. The '-n' option is used to set the next user PC to be executed at 'address' without starting execution. If the user then executes a 'step' command, program execution will begin at the address specified by 'address'.

GoTill

```
gotill|gt <address>
```

The 'gotill' command will continue execution from the current value of the user program counter. The program will stop execution just prior to the execution of the instruction pointed to by 'address'. This command actually installs a breakpoint at 'address'. This breakpoint will continue to remain active as if it were set by the 'brk' command. To get a listing of the currently active breakpoints the 'brk|b' command may be used.

Next (step over subroutine)

```
next|n [count]
```

The 'next' command is similar to the 'step' command, except that when a *jal* or *bal* instruction is encountered, all of the instructions of the subroutine are executed until the subroutine returns to the instruction following the *jal* or *bal*. In other words, 'next' skips over the subroutines as far as single stepping is concerned.

Single Step

step|s [count]

The 'step' command executes a single step (if count is not specified) or 'count' number of steps.

The 'step' command treats branch instructions and the following instruction in the branch delay slot as atomic and a single step executes both instructions.

Unbreakpoint

unbrk|ub <bpnumlist|ALL>

This command will remove all of the breakpoints listed in <bpnumlist>. These are the ordinal numbers of the breakpoints and can be obtained by doing a 'brk' command.

<IDT>ub 2 4 5

The above command will remove breakpoints 2, 4, and 5.

<IDT>ub ALL

This command will remove all of the currently set breakpoints.

Memory/Register and Assembly/Disassembly commands

The Memory/Register access commands handle the changing, displaying and moving of data. Each of these commands can be entered with an option to specify the data size and the range (not optional) of locations affected. The size options have the following meaning:

-d	DoubleWord access
-w	Word access
-h	Halfword access
-b	Byte access
-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
0xa0005004: 01205821  move   t3,t1
0xa0005008: 00000000  nop
0xa000500c: 1000fffc  b      0xa0005000
0xa0005010: 00000000  nop

```

The assembler only accepts native instructions. For example, to enter:

```
la    v0,0x80014000
```

the user must enter:

```
lui    v0,0x8001
ori    v0,v0,0x4000
```

It should also be noted that the radix should be explicitly specified. Shift amounts and signed and unsigned immediate values are assumed to be decimal. Target values are assumed to be hexadecimal. For the R4xxx architecture targets, the assembler accepts all MIPSII and MIPSIII architecture instructions. The list of cp0 register names that the assembler accepts follows:

◆ *MIPS-I (R30xx CPUs):*

```

index   - index register
random  - random register
tlblo   - tlb low entry
tlbhi   - tlb high entry
context - context register

```

sr - status register
 cr - cause register
 epc - exception pointer register
 config - configuration register
 badvaddr - bad virtual register
 ♦ *MIPS-II/III (R4xxx CPUs):*
 index - index register
 random - random register
 tlblo0 - tlb entry low 0
 tlblo1 - tlb entry low 1
 context - context register
 pagemask - TLB pagemask register
 wired - TLB wired register
 badvaddr - bad virtual register
 count - count register
 tlbhi - TLB entry high
 compare - count comparison register
 sr - status register
 cr - cause register
 epc - exception pointer register
 prid - processor id
 config - configuration register
 lladdr - load link address
 watchlo - watchpoint low register
 watchhi - watchpoint high register
 xcontext - extended context register
 ecc - cache ECC register
 cacheerr - cache error register
 taglo - cache tag low register
 taghi - cache tag hi register
 errpc - cache error exception pointer

The list of CP1 register names that the assembler accepts is:

f0-f31 - floating point registers
 feir - FPA implementation register
 fcsr - FPA control and status register

Cache Flush

cacheflush/cf [-i|-d|-n]

For the R30xx CPU, the cache flush command invalidates both the I-cache and the D-cache, if no option is specified. To flush only one cache, the optional argument may be entered. For example:

To flush both the I and D caches, enter:

<IDT>cf

To flush only the i cache, enter:

<IDT>cf -i

For the R4xxx CPU the cache flush command flushes and invalidates the primary instruction, primary data and secondary instruction/data caches. Normally any dirty lines in the cache will be written out before the cache line is invalidated. If the '-n' option is specified, the caches are simply flushed; any unwritten data in the cache will be discarded. If no options are given or only the '-n' option is used, all three caches will be modified.

Compare Block

compare|cp [-w|-b|-h] <RANGE> <destination>

Compares the block of memory specified by 'RANGE' to the block of memory that starts at 'destination'. Overlapping blocks will be allowed, however they probably do not make much sense. The comparison will continue until a mismatch is found. The monitor will then output the address, the 'should be' value, the destination address, and the 'is' value. The user may either enter a carriage return to continue the comparison or enter a period (.) to terminate the comparison.

Examples:

```
<IDT>cp 4000/128 5000
```

Compares 128 words of data, starting at location 0xa0004000, to a 128 word block of data starting at location 0xa0005000.

Disassemble Contents Of Memory

dis <RANGE>

Disassemble the contents of memory specified by range. If 'RANGE' consists only of a beginning address then enough locations following the beginning address are disassembled to fill one screen. To view disassembly of long pieces of code, the 'RANGE' needs to be specified only the first time. Subsequent 'dis' commands without any 'RANGE' will continue to display subsequent pages of disassembly.

If 'dis' is used immediately after a 'load' command, 'dis' without a 'RANGE' will automatically display disassembly starting at the beginning of the downloaded code. The names used for the registers depend on the register set selected (compiler/hardware) with the 'regsel' command.

Dump Cache

dc [-i|-d] [RANGE]

This command displays the instruction or data cache contents and the tag values if the cache location is valid. If no option is entered, the data cache is dumped. 'RANGE' is always assumed to be in the range of zero(0) to the size of the cache. If the addresses are larger than the cache size entered, they are treated as modulo cache-size.

The display format for the data cache is shown below. For this example cache location 0x5004 is valid and the tag value is 0x00030000 and the data is 0x12345678. Cache location 0x500c is also valid with a tag of 0x00030000, however there is an inconsistency between cache contents and main memory (cache contains 0x00000000 and main memory contains 0x55555555).

```
<IDT>dc 5000/4
```

Tag/Invalid	0x00005000	Data/00000000
Tag/0x00030000	0x00005004	Data/12345678
Tag/Invalid	0x00005008	Data/00000000
Tag/0x00030000	0x0000500c	***00000000x55555555

The display format for the instruction cache is shown below. For this example a program was executed in kseg0 starting at 0x80014000. Location 0xa0014008 was purposely written to force an inconsistency between main memory and the 'i' cache after the program was executed.

```
<IDT> dc -i 4000/8
```

Tag/0x00010000	0x00004000	0x24090001	li	t1,1
Tag/0x00010000	0x00004004	0x240a0002	li	t2,2


```

Tag/0x00010000      0x00004008  ***0x00000000  0x12345678
Tag/0x00010000      0x0000400c  0x240b0003  li    t3,3
Tag/0x00010000      0x00004010  0x012a6020  add   t4,t1,t2
Tag/0x00010000      0x00004014  0x3c028001  lui   v0,0x8001
Tag/0x00010000      0x00004018  0x34426000  ori   v0,v0,0x6000
Tag/0x00010000      0x0000401c  0xac4c0100  sw    t4,0x100(v0)

```

Dump Memory

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

The 'dump' command displays the memory specified by 'RANGE' to the display screen. The start and end addresses of the range are rounded down modulo 16 and up modulo 16 respectively (i.e. if the range requested was 24 to 53, then addresses 16 through 63 will be displayed to the screen). The memory contents are dumped in the default radix. The options have the following meaning:

- d - Doubleword access.
- w - Word access
- h - Halfword access.
- b - Byte access

The format of the memory dump is shown below. Each line will show 4 items of data either in word format (32 bits) or halfword (16 bits) format, 2 items in doubleword format (64 bits) or 8 items in byte format (8 bits). The right-hand portion of the display will show the items of data as ASCII characters. If the data is a non-printing character, an apostrophe (') will be shown in its place.

Examples:

assume: seg.=kseg1 and rad=hexadecimal.

<IDT>d 200/4

a0000200: 41424344 31003220 45464739 01020304 *ABCD1'2 EFG9''''*

<IDT>dump -h 200/4

a0000200: 4142 4344 3100 3220 4546 4739 0102 0304 *ABCD1'2 EFG9''''*

The default access type is 'word'.

Dump Registers

`dr [reg#|name|reg_group]`

This command will print out the current contents of registers. It should be noted that the register contents only have sense after running (single stepping) or after encountering a breakpoint in a client program. After IDT/sim is first started all registers are cleared, the cache is cleared and the TLB is invalidated. If the user requests a specific register from the group of CPU registers (r0-r31, hi, lo or pc) then that particular register is displayed in the default radix. If the user requests a dump of a particular coprocessor register then these registers are dumped in a special format for ease of reading.

To dump all of the coprocessor zero registers, enter the following command:

<IDT>dr cp0

All of the coprocessor zero registers may be dumped individually and will be displayed with the individual fields separated, for easy identification by the user. For example, the user could enter the following command:

<IDT>dr sr

The contents of the *status* register would be displayed in a field by field display as follows:

sr: cu bev ts pe cm pz swc isc intmsk kuo ieo kup iep kuc iec

x b b b b b b b x b b b b b b

Where: 'x' is a hex digit and 'b' is a binary (0/1) digit.

To dump all of the floating point registers in a hexadecimal format, enter the command:

```
<IDT>dr fpr
```

To dump all of the floating point registers in a single precision format, enter the command:

```
<IDT>dr fps
```

To dump all of the floating point registers in a double precision format, enter the command:

```
<DT>dr fpd
```

Fill Memory

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

The 'fill' command fills memory specified by 'RANGE' with the contents of 'value_list'. 'Value_list' is a list of 0 to 8 blank-separated values that will be repeated over and over until the amount of memory specified by RANGE is exhausted. If 'RANGE' is smaller than the number of values, only enough values, starting at the beginning of the list will be used. If 'value_list' is empty, memory specified by 'RANGE' will be cleared to zero.

Examples:

```
assume: seg.=kseg1 and rad=hexadecimal.
```

```
<IDT>fill 0x80060000-0x80060100 0x12345678 0xabcdef00
```

The above command fills memory inclusive between 0x80060000 and 0x80060100 with the repeating pattern 0x12345678,0xabcdef00.

```
<IDT>f -h 50000/256 aaaa 5555 0000 ffff
```

The above command fills memory inclusive between 0xa0050000 and 0xa0050200 with a repeating pattern 0xaaaa5555 0x0000ffff.

```
<IDT>f 50001/64 22334455
```

The above command fills memory as shown below:

```
a0050000: 00223344
a0050004: 55223344
a0050008: 55223344
:
:
a00500fc: 55223344
a0050100: 55000000
```

Fill Register

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

Puts 'value' into the register specified by 'reg#|name'. However, an exception is made for double precision floating point registers. The registers in case of double precision must be accessed as "dn" where "n" is an even number. To fill register r3 with the value 0x12345678 the user may enter either of the commands below:

```
<IDT>fr r3 0x12345678
```

```
<IDT>fr v1 0x12345678
```

The options -s and -d are used to specify either single or double precision values for the floating point registers f0-f30. When entering values into the floating point registers in the floating point format, only even numbered registers may be specified. The commands below are examples of entering values into the floating point registers:

```
<IDT>fr -s f4 1.375
```

```
<IDT>fr -d d10 3.45e10
```

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 (in the R30xx). 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 the 'x' are 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.

Substitute Memory

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

The 'sub' command allows the user to examine and change memory interactively. When the user enters the 'sub' command, the 'address' followed by the contents of memory at the address are displayed on the next line. At this point the user may enter a new value or a carriage return ('Enter' key) or a period (.).

If the user enters a new value, the current contents of memory at 'address' are replaced by the new value. The monitor then displays the next sequential address and its contents on the next line on the console and waits for the next user input.

If the user presses the 'carriage return' or 'Enter' key, the monitor will not alter the contents of memory at 'address' and will just display the next sequential address and its contents on the next line.

This sequence can be repeated over and over until the user enters a period (.) which terminates the 'sub' command and the monitor will return to the standard command prompt '<IDT>' and wait for the next command.

Examples:

assume: seg=kseg0 and rad=hexadecimal. User input is underlined.

Memory starting at 0x80005000 contains:

0x80032001 0x32402200 0x00230444 0x3309765.

<IDT>sub 4000

80005000: 80032001 12345678

80005004: 32402200 aaaa5555

80005008: 00230444 <Enter>

8000500c: 33809765.

<IDT>

The above sequence would leave the following pattern in memory starting at 0x80005000: 0x12345678 0xaaaa5555 0x00230444 0x33809765

<IDT>sub -h 5000

80005000: 1234 8765

80005002: 5678 4321

80005004: aaaa <Enter>

```
80005006: 5555 9999
80005008: 0023 <Enter>
8000500a: 0444
<IDT>
```

The above sequence would leave the following pattern in memory starting at 0x80005000:

```
0x87654321
0xaaaa9999
0x00230444
```

Write Cache Memory

```
wc [-i] [-w|-b|-h] <RANGE> [value_list]
```

Note: This command is not available in the IDT/sim for the R4xxx CPU.

Addresses are automatically set to Kseg0 and the selected cache is isolated. This command will fill the selected cache memory specified by 'RANGE' with the pattern specified by 'value_list'. The user may access either the data cache or the instruction cache. If the '-i' option is selected the instruction cache will be accessed, otherwise the data cache is accessed. If the user enters:

```
<IDT>wc 0x1000-1100
```

data cache locations 0xc000 -0x1100 will be filled with zeros. The options have the following meaning:

```
-i - Select the instruction cache.
-w - Word access.
-b - Byte access.
-h - Halfword access.
```

Writing a word to the cache while it is isolated will validate that particular entry. The default access type is 'word' and the default cache is the data cache. All addresses should be within the actual cache limit. Cache size is determined dynamically at initialization time of IDT/sim.

Set-up and Environment Commands

Checksum

```
checksum[cs [start_addr num_bytes]
```

This command calculates the checksums for each of the EPROMs on the target board and displays the results on the console. If the optional arguments are entered, the checksums for the area of memory specified are calculated. By default it is assumed that the EPROMs begin at address 0xbfc00000 and are 0x20000 bytes deep. The two forms of the command below would do the same operation.

```
<IDT>cs
<IDT>cs 0xbfc00000 0x20000
```

Help Command

```
help[?] [commandlist]
```

This command will print out the list of commands available in IDT/sim. If a command list is supplied, only the syntax for the commands in the list is displayed.

History Command

```
history|h
```

The 'history' command will display the last 16 commands entered with identifying numbers so that the user may re-execute one of those commands by entering '!#' where '#' is the command number from the list. This is a circular list in that at any time the latest 16 commands are available.

Initialize

init|i

This command is a 'warm reset' and is analogous to pressing the hardware reset switch. The 'init' command will initialize the 'bss' area and stack designated by IDT/sim. It also clears the breakpoint table, but does not clear the user memory space.

Register Set Select

regsel|rs [-c|-h]

This command allows the user to select the format of the register names. There are two formats for the register names - 'compiler' names or 'hardware' names. The 'compiler' names are: a0, t1, s0, etc. The 'hardware' names are: r0, r1,..., r31. The default selection when the monitor first powers up is 'compiler' names.

Set Default Radix

rad [-o|-d|-h]

Set the default radix to the requested base.

- o - Octal
- d - Decimal
- h - Hexadecimal

If no argument is supplied, the radix in force at the time is displayed.

Set Default Segment

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

The 'seg' command sets the default segment to the requested segment.

- 0 - Kseg0 0x80000000
- 1 - Kseg1 0xA0000000
- 2 - Kseg2 0xC0000000 (R30xx only)
- s - Kseg 0xC0000000 (R4xxx only)
- 3 - Kseg3 0xE0000000 (R4xxx only)
- u - Kuseg 0x00000000

When the user enters an address and does not specify all 8 nibbles of the 32 bit address, the address entered is 'or'-ed with the default segment value. If no argument is supplied with the 'seg' command, the segment in force at the time is displayed.

TLB Commands**TLB Dump**

tlbdump|td [RANGE]

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 R30xx CPU:

```
tlbmap|tm [-i index] [-ndgv] <vaddress> <address>
```

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.

For an R4xxx CPU:

```
tlbmap|tm [-i INX] [-v/d/g][0/1]] [-g] [-p PAGESIZE] [-c CACHEALG] VADDR PADDR [PADDR]
```

The 'tlbmap' command establishes a virtual to physical mapping in the TLB. A particular entry may be specified with the *-i* option; if no entry is specified, the entry specified by the R4xxx 'wired' register is used. The *-v*, *-d* and *-g* options allow the valid, dirty and global bits of the TLB low entries to be set.

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- Reserved
- 1- Reserved
- 2- Uncached
- 3- Cached Noncoherent
- 4- Cached Coherent Exclusive
- 5- Cached Coherent Exclusive Write
- 6- Cached Coherent Update
- 7- Reserved

In general, only the values 2 and 3 should be used. Using other values (particularly reserved ones) may cause mysterious problems with the caches.

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>
```

This command searches the TLB for translations which map to 'physaddr'. Any translations found, valid or invalid, are displayed. The default segment is not applied to 'physaddr'.

Trace Commands

The trace commands allow the user to trace memory accesses of a user program. Such things as the path of execution, writes/reads to a specific address or range of addresses, execution of a specific instruction and/or all calls may be traced. The trace occurs in a non-real-time execution mode. There is a trace mode that allows real-time execution for all but a specified range or set of ranges of program execution. For example, to accommodate programs which call ROM-based code, an address range not to be traced can be specified. Calls to the excluded address range will be executed real-time.

Trace Command

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

The 'trace' command defines the 'trace equation' and/or enables/disables tracing. The options are explained below:

- a - trace all instructions executed.
- o - turn off all previously selected trace conditions.
- e - enable tracing.
- d - disable tracing
- r RANGE - trace all reads from address range 'RANGE'
- w RANGE - trace all writes to address range 'RANGE'
- c RANGE - trace all calls to address range 'RANGE'
- i INS - trace all execution of instruction 'INS'
- m MSK - Mask value for tracing a specific instruction

Options may be specified in any combination and are interpreted in the order that they appear on the command line. An example of setting up trace conditions to trace 'all' memory accesses and 'enable' tracing is shown below:

```
<IDT>t -a -e
```

To trace all writes to a particular area of memory (for example, the user stack) the following command may be used:

```
<IDT>t -w 0x800fe000-0x800fffc -e
```

If the stack is in the area of 0x800fe000-0x800fffc, the above command will capture all writes to the stack. It also enables tracing.

The command shown below will trace all reads and writes to an area:

```
<IDT>t -r 0x800fe000-0x800fffc -w 0x800fe000-0x800fffc -e
```

To see the current trace selection the user enters the 'trace' command with no arguments (user input underlined below):

```
<IDT>t -o
```

```
<IDT>t -e -r bfe00000-bfe0003f -w 50000-60000
```

```
<IDT>t
```

```
Trace reads - bfe00000-bfe0003f
```

```
Trace writes - a0050000-a0060000
```

```
Trace enabled
```

Note that the range specification uses the default radix and segment. All of the trace commands interact with each other. The trace buffer size is set to 512 entries. The 'ts' (trace stop condition) command is used to determine when to stop tracing. The tc (trace conditionally) command will allow switching from real-time mode to trace mode automatically. An example of how to trace a specific instruction is shown below:

```
<IDT>t -i "mfc0 t0,sr"
```


The above command will trace every occurrence of the instruction 'mfc0 t0,sr'. Note that the instruction is entered with the same syntax that is used for the incremental assembler. The entered instruction must be enclosed in quote marks.

Breakpoints will not halt execution with trace enabled unless the 'trace stop' condition is set to stop on breakpoint. The trace may be disabled without erasing the trace equation. With the trace disabled, breakpoints work normally. Using the 'trace conditionally' command with breakpoints, will automatically toggle trace enable.

By specifying a mask (-m MSK) the user can trace classes of instructions. For example to trace all 'mfc0' instructions regardless of the registers involved, the following trace command could be used:

```
<IDT>t -i "mfc0 t0,sr" -m 0xffe007ff -e
```

The mask is applied to the values before the test for equality is made. It masks out the 'rt' and 'rd' values for the 'mfc0' instructions.

```
<IDT>t -i "mfc0 t0,sr" -m 0xffff07ff -e
```

The above command would cause all "move from coprocessor zero status register" instructions to be traced. Below are tables of useful masks for some of the instructions:

Store to memory/Load from memory	
Any register	0xffe0ffff
Any base	0xfc1ffff
Any offset	0xffff0000

Add,Addu,And,Nor,Or,Slv,Slt,Sltu,Sub,Subu,Xor	
Any destination	0xffff07ff
Any operands/specific destination	0xfc00ffff
Any operands/destination	0xfc0007ff

Mfcz,Mtcz,Cfcz,Ctcz Where 'z' is 0 - 3	
Any coprocessor register	0xffff07ff
Any general register	0xffe0ffff

Obviously, some combinations do not make sense. Example:

```
<IDT>t -o -d -a -r 0xa0010000/10
```

The -o disables tracing, so the -d is superfluous. The trace conditions are 'and'-ed, so the -a will trace everything including the trace conditions specified by the -r option.

Trace Stop Command

```
ts [-b|-f|-o|-r RANGE|-w RANGE|-i INS|-m MSK]
```

The 'ts' command defines the conditions necessary to halt execution of the client program and return to monitor mode so the user may examine the trace buffer. The trace buffer wraps so that when execution stops only the last 512 or less events are available. The options are explained below:

- b - Stop on occurrence of a breakpoint
- f - Stop on trace buffer full
- o - Cancel all trace stop conditions
- r RANGE - Stop on reads from address range 'RANGE'
- w RANGE - Stop on writes to address range 'RANGE'
- i INS - Stop on execution of instruction 'INS'

-m MSK - Mask value for instruction to stop tracing

Any or all options may be specified. Execution will stop on the first condition to be satisfied. Entering the 'ts' command without arguments will display the trace conditions. This is the same as the 't' command.

Examples:

```
<IDT>ts -f
```

The above command will continue tracing until the trace buffer is full.

```
<IDT> ts -f -b -i "mfc0 t1,sr" -m 0xffe0ffff
```

The above command will trace until the buffer is full or a breakpoint is encountered or a 'mfc0' instruction moves any general register to 'sr'.

Trace Conditionally Command

```
tc [-e BPNUM] [-d BPNUM]
```

This command defines the limits of tracing. By placing breakpoints at the start of a code segment and at the end and using the trace conditionally command a user may specify a section of code to trace. The user should use the 'trace' (t) command to define the items traced and the 'trace stop' (ts) command to specify when to halt execution.

Typically the client program would be started with trace disabled. When the enabling breakpoint is reached, tracing will commence and continue until the disabling breakpoint is reached. Except for the time that tracing is enabled the program will execute in real time. An example of how to use this command is shown below (user input underlined):

```
<IDT>t -a -d
```

```
<IDT>ts -f
```

```
<IDT>b 40100 40188
```

```
<IDT>b
```

```
bp 0= 0x80040100:24090000li t1,0x0
```

```
bp 1= 0x80040188:03e00008jr ra
```

```
<IDT>tc -e 0 -d 1
```

```
<IDT>t
```

Trace all

Stop on buffer full

Trace disabled

```
bp 0= 0x80040100:24090000      li      t1,0x0      Start Trace
```

```
bp 1= 0x80040188:03e00008      jr      ra      Stop Trace
```

```
<IDT>
```

The 'tc' command specifies to enable tracing when breakpoint 0 (-e 0) is reached and to disable tracing when breakpoint 1 (-d 1) is reached. Up to 16 breakpoints may be specified and the 'trace conditionally' command may be used to set enable/disable tracing on any or all of the breakpoints. It is not legal nor does it make any sense to try and enable and disable tracing on the same breakpoint.

```
<IDT>t -a -d
```

```
<IDT>ts -f
```

```
<IDT>ub all
```

```
<IDT>b 40100 40188 40200 40214
```

```
<IDT>b
```

```
bp 0= 0x80040100:24090000      li      t1,0x0
```

```
bp 1= 0x80040188:03e00008      jr      ra
```

```
bp 2= 0x80040200:240c0000      li      t4,0x4
```

```

bp 3= 0x80040214:03e00008      jr      ra
<IDT>tc -e 0 -e 2 -d 1 -d 3
<IDT>t
Trace all
Stop on buffer full
Trace disabled
bp 0= 0x80040100:24090000      li      t1,0x0      Start Trace
bp 1= 0x80040188:03e00008      jr      ra          Stop Trace
bp 2= 0x80040200:240c0000      li      t4,0x4      Start Trace
bp 3= 0x80040214:03e00008      jr      ra          Trace Stop
<IDT>

```

The above sequence specifies a couple of ranges to trace. If the enabling breakpoint is never reached then no information will be placed in the trace buffer. Even if the breakpoint is reached and after the trace is enabled no conditions for capturing trace information are satisfied, then no information will be placed in the trace buffer.

Trace Dump Command

dt

The 'dump trace' command displays the contents of the trace buffer. The format is shown below:

```

<IDT>dt
Dump Trace Buffer - 512 entries
<line#> <virtual addr> <disassembled instr> <register contents>
:           :           :           :
:           :           :           :
<line#> <virtual addr> <disassembled instr> <register contents>
<up(u) down(space) line #(l nnnn) search addr(s addr) next(n) quit(q)>

```

When the user has captured a buffer full of trace data and enters the 'dt' command, the contents of the buffer will be displayed to the console 21 lines at a time in the format shown above. The last line of the display prompts the user to enter either an 'u' to scroll up or a 'space' to scroll down, or the letter 'l' followed by a line number from 0-511, or search the trace buffer starting at line 0 for an address, or next(n) to search for the next occurrence of address, or to quit(q).

In case of specifying a line number, the contents of the trace buffer starting at the line number and continuing for 21 entries will be displayed. If 'search for address' is used and the specified address is found in the trace buffer then 10 entries before and 10 entries after are displayed with the found line marked with (***). If the specified address is not found, the next page of trace buffer contents is displayed.

Trace Exclude Command

tex [RANGE]

The trace exclude command allows the user to specify an address area, the calls to which will not be traced. The main reason for this is to prevent calls to library or ROM space from being traced. Trying to trace calls to ROM space will not work.

This command is very useful if the user is linking with IDT/sim's library functions (i.e. printf, string rts...). If the user has linked with IDT/kit or IDT/C 5.0 or later, the area addressed by the routines from 'liblnk.a' should be excluded. Tracing is done by actually single stepping and examining the instruction being executed and its operands. If the instruction is a *branch-and-link* or a *jump-and-link* the target address is calculated and tested to see if it falls in the excluded area. If it does, the branch or jump is executed real time with a breakpoint inserted at the return address, where tracing will continue normally.

Examples:

```
<IDT>tex 45110-47000
```

Any calls to the area 45110-47000 would be stepped over (similar to executing a 'next' command on the 'jal' instruction).

```
<IDT>tex
```

```
Exclude a0045110-a004700
```

The "tex" command entered without any arguments will display the currently excluded area. On power up, the default excluded area is 0xbfc00000-0xbffffc. The default segment and radix are applied to the address specification at the time the tex command is entered. Changing the segment or radix later does not effect the current exclusion area.

Trace Command Examples

Assume a client program with a kernel running in real time and starting at location 0x80020000. There is a subroutine starting at location 0x80040000 and ending at 0x80040120 that gets called infrequently. The user would like to trace this subroutine. The following command sequence could be used (user input underlined>):

```
<IDT>seg -0
```

```
<IDT>t -a
```

```
<IDT>ts -f
```

```
<IDT>b 40000 40120
```

```
<IDT>tc -e 0 -d 1
```

```
<IDT>t
```

Trace all

Stop on buffer full

Trace disabled

```
bp 0= 0x80040000:24090000      li      t1,0x0      Start Trace
```

```
bp 1= 0x80040120:03e00008      jr      ra          Stop Trace
```

```
<IDT>
```

The above sequence sets the default segment to kseg0, trace all, stop tracing on buffer full, bracket the subroutine with breakpoints at locations 0x80040000 and 0x80040120, start tracing when breakpoint zero(0) is encountered and trace all until breakpoint one(1) is encountered.

To see the trace conditions that are set, the 'trace'(t) command with no arguments is entered. With the trace conditions set, the user may start the program and it will run real time, except for the portion of time that it is executing in the range of 0x80040000- 0x80040120.

Network Related Commands

Network related commands assume that an ethernet device driver has been installed on the board and the required hardware is also present.

Download and execute binary file (boot)

```
boot [-n] [[HOST:]FILE]
```

The 'boot' command downloads and executes an executable binary file via TFTP. If the -n option is given, the file is simply downloaded ready for execution by the 'go' command.

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 [-lnqrsv] [-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 -i 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 and time

date [[[[[yy]mm]dd]hh]mm.[ss]]

The Set / display date and time command is used to display and set real time clock. With no arguments, the current date and time is simply displayed. The optional argument allows the time to be set. Each section of the string consists of two digits: yy - the year modulo 100, mm - the month (1-12), dd - the day (1-31), hh - the hour (0-23), mm - the minute (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.

Commands specific to the 79S134 evaluation board

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 in the 79S134 companion device.

gmsk

gmsk

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

isrt

isrt

This command first clears all pending interrupts from 79RC32134 in the interrupt expansion controller pending register. Next, an interrupt is triggered through timer0 rollover, and the interrupt is serviced before returning.

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.

SPI256 related commands

The following commands are described in details in the chapter "Using the system diagnostics command": *spi256*, *spi_wr*, *spi_rd*, *spi_bp*.



IDT/sim User Command Summary

Notes

Quick Reference

IDT/sim User Commands

asm <addr>

Allows the user to interactively examine and change memory, using standard assembler mnemonics.

benchmark|bm

Before issuing this command, the code to be benchmarked should be downloaded to the target board. This command returns the total elapsed time for executing the entire downloaded code, in microseconds. The time is displayed on the monitor.

brk|b [address list]

Displays all of the currently set breakpoints, if an address list is not supplied. If an address list is supplied, breakpoints are set at each of the addresses in the list.

boot [-n] [[HOST:]FILE]

Downloads a binary file via ethernet and executes it on the target board. '-n' prevents execution.

cacheflush|cf [-i|-d|-n]

Flushes both the i-cache and the d-cache, if no option is specified. If the user wants to just flush one or the other, the optional argument may be entered. In the R4xxx, '-n' prevents write-back.

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

Invokes a sub-procedure under the monitor environment. Executes a jump-and-link to the address passing up to eight arguments while still in monitor mode.

caus

Displays *cause* register. (79S134 board only)

checksum|cs [start_addr num_bytes]

Displays the checksum for each of the IDT/sim EPROMS.

clcs

Clears *cause* register. (79S134 board only)

clsr

Clears *status* register. (79S134 board only)

compare|cp [-w|-b|-h] <RANGE> <destination>

Compares the block of memory specified by RANGE to the block of memory that starts at destination.

cont|c

Continues execution of the client process from where it last halted execution.

cmpr

Timer test on 79S134 board.

creg

Sets individual fields of the SDRAM control register in the 79S134 companion device.

date [[[[yy]mm]dd]hh]mm.[ss]]

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]

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'.

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

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

gmsk

Displays interrupt mask. (79S134 only)

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: silen mode; no dots are printed on screen.

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

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 [-Inqrv] [-c COUNT] [-i WAIT] [-s SIZE] HOST

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

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. (79S134 only)

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.

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 index] [-ndgv] <vaddress> <paddress> (for R30xx)**tlbmap|tm [-i INX] [-v|d|g][0\1] [-g] [-p PAGESIZE] [-c CACHEALG] VADDR PADDR [PADDR] (for R4xxx)**

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 >.

tlbptov|tp <physaddr>

This command searches the translation buffer looking for translations which map to <physaddr>. Any translations found, valid or invalid, are displayed. The default segment is not applied to <physaddr>.

ts [-b|-f|-o|-r RANGE]-w RANGE|-i INS|-m MSK]

Defines the conditions necessary to halt execution while tracing of the client program and to return to monitor mode so the user may examine the trace buffer.

unbrk|ub <bpnumlist>

Uninstalls all of the breakpoints listed in <bpnumlist>. These are the ordinal numbers of the breakpoints and can be obtained using the 'brk' command.

unsetenv VAR

This command allows the user to delete environment variables from NVRAM.

wc [-i] [-w|-b|-h] <RANGE> [value_list]

Addresses are automatically set to Kseg0 and the selected cache is isolated. This command will fill the selected cache memory specified by RANGE with the pattern specified by value_list.



Motorola S-record Format

Notes

Each S-record is made up of 6 fields in the following format:

Field	S	type	length	address	data	checksum
Characters	1	1	2	4, 6 or 8	var	2

The specifications for each S-record field are as follows:

S - ASCII character 'S' (0123 octal). Signal the start of the S-record.

type - Record type of one of the digits 0, 1, 2, 3, 5, 7, 8, 9 with the following meaning:

0 -	Header record. Address field is 2 bytes long and zero. Data field may contain any identifying information (or be omitted completely).
1 -	Data. Address field is 2 bytes (4 chars) long.
2 -	Data. Address field is 3 bytes (6 chars) long.
3 -	Data. Address field is 4 bytes (8 chars) long. IDT SDS 2.0 uses this data format.
5 -	Address field contains count of type 1, 2 and 3 records in a group. Data field is omitted.
7 -	Terminating record - signals the end of block of type 3 records. The address field (4 bytes - 8 chars) may contain transfer address.
8 -	Terminating record - signals the end of block of type 2 records. The address field (3 bytes - 6 chars) may contain transfer address.
9 -	Terminating record - signals the end of block of type 1 records. The address field (2 bytes - 4 chars) may contain transfer address.

length - One half of the total number of characters in the address, data and checksum fields. This number is encoded as two characters representing the number in hexadecimal (one-byte quantity). Valid hexadecimal numbers are 0-9 and A-F.

address - Address at which the data portion of the record is to be stored in memory.

data - Actual data, each byte represented as a pair of hex digits in ASCII.

checksum - Computed over the length, address and data fields. All bytes (represented as hex character pairs) from these fields are added together, one's complement of the result is taken and the least significant byte represented by 2 ASCII hex digits is put into checksum field.



Register Numbers and Names

Notes

Register #	Name(s)	Use
0	r0, zero	wired to zero
1	r1, at	assembler temp
2	r2, v0	return value
3	r3, v1	return value
4	r4, a0	argument 0
5	r5, a1	argument 1
6	r6, a2	argument 2
7	r7, a3	argument 3
8	r8, t0	Temporary, not preserved across calls
9	r9, t1	"
10	r10, t2	"
11	r11, t3	"
12	r12, t4	"
13	r13, t5	"
14	r14, t6	"
15	r15, t7	"
16	r16, s0	Must be saved on entry to subroutine, if used, Must be guaranteed across calls
17	r17, s1	"
18	r18, s2	"
19	r19, s3	"
20	r20, s4	"
21	r21, s5	"
22	r22, s6	"
23	r23, s7	"
24	r24, t8	Temporary, not preserved across calls
25	r25, t9	"
26	r26, k0	kernel temporary
27	r27, k1	kernel temporary
28	r28, gp	global pointer
29	r29, sp	stack pointer
30	r30, fp/S8	frame pointer
31	r31, ra	return address

Register #	Name(s)	Use
------------	---------	-----

The following are the coprocessor-0 (CP0) registers (they are defined in the file "header/idtcpu.h"):

0	C0_INX	tlb index
1	C0_RAND	tlb random
2	C0_TLBLO	tlb entry low (R30xxE)
2	C0_BUSCTRL	bus control (R3041)
3	C0_CONFIG,cfg	hardware configuration (R3041/R3081 only)
4	C0_CTXT	tlb context
8	C0_BADVADDR	bad virtual address
9	C0_COUNT	timer counter (R3041/R32364)
10	C0_TLBHI	tlb entry high (R30xxE/R4600/R32364)
10	C0_PORTSIZE	port width control (R3041)
11	C0_COMPARE	timer compare (R3041)
12	C0_SR, sr	status register
13	C0_CAUSE,cr	exception cause
14	C0_EPC,pc	exception pc
15	C0_PRID	revision number

The following registers are only in the **R4600 and R32364**:

2	C0_TLBLO0	EntryLo0
3	C0_TLBLO1	EntryLo1
5	C0_PAGEMASK	page compare mask
6	C0_WIRED	wired boundary
16	C0_CONFIG	hardware configuration
26	C0_ECC	error check and correction
27	C0_CACHEERR	error: cache index and status
28	C0_TAGLO	lo and hi together hold pri-
29	C0_TAGHI	mary cache tag & parity
30	C0_ERRPC	parity error pc

The following registers are **only in the R4600**:

17	C0_LLADDR	recent load/linked address
18	C0_WATCHLO	watchpoint trap
19	C0_WATCHHI	watchpoint trap
20	C0_XCTXT	extended TLB context

The following register is **only in the R4640/R4650**:

0	C0_IBASE	I-base in R4640/R4650
1	C0_IBOUND	I-bound in R4640/R4650
2	C0_DBASE	D-base in R4640/R4650
3	C0_DBOUND	D-bound in R4640/R4650
17	C0_CALG	cache algorithm

Register #	Name(s)	Use
------------	---------	-----

The following registers are in the **R4640/R4650/R32364**:

18	C0_IWATCH	instruction breakpoint virtual address
19	C0_DWATCH	data breakpoint virtual address

The following registers are **only in the R32364**:

22	C0_IECP	imprecise exception PC
23	C0_DEPC	debug exception PC
24	C0_DEBUG	debug control/status register

Floating-Point Registers

The R3000 family implements 16, 64-bit registers; each register is made up of an even (least significant) and odd (most significant) 32-bit register. Any 32-bit register can be the target of a load or store; only even registers can be specified for FPA operations.

In the R4700 and R5000, the registers are identical to the R3000 when the *FR* bit of the CP0 Status register is clear. When *FR* is set, 32, 64-bit registers are enabled, any of which can be specified for FPA operations.

In the R4650, only single-precision operations are allowed. When the *FR* bit is clear, only the even registers are available, even for loads and stores. When *FR* is set, 32, 32-bit, single-precision registers are enabled, any of which can be specified for FPA operations.

Table 13.1 lists the floating-point register usage and calling convention, effective regardless of whether 16 or 32 registers are available.

Registers	Calling Convention
f0 - f3	Used to hold floating-point type function results (\$f0) and complex type function results (\$f0 has the real part, \$f2 has the imaginary part)
f4 - f11	Temporary registers, used for expression evaluation, whose values are not preserved across function calls
f12 - f15	Used to pass the first two single or double precision arguments, whose values are not preserved across function calls
f16 - f19	Temporary registers, used for expression evaluation, whose values are not preserved across function calls
f20 - f31	Saved registers, whose values must be preserved across function calls

Table 13.1 Floating Point Register Usage

Control registers

Register #	Name(s)	Use
------------	---------	-----

\$0	feir	Implementation/control Reg.
\$31	fcsr	control/status register



IDT/sim and Secondary CACHE of R5000

Notes

This is a brief description of what IDT/sim source files offer in terms of support for the secondary cache of R5000 RISController.

The file **SIM4000/MAIN.C** displays the size and line-size of the secondary cache of R5000, in the function `main()`. The variables used are: `scache_size` and `scache_linesize`. These variables are declared in the file **common/globals.c**.

The file **SIM4000/csu_idt.S** enables the secondary cache in its start-up routine.

The `#define` macros used to define argument values related to the secondary cache are in the file **header/IDTCPU.H**. Search for the word `L2_PAGESIZE`, which is the first of the `#defines`.

The structure `mem_config` which holds the current information about cache sizes of the current RISController on board, is defined in the file **header/IDTMON.H**. The functions which updates and sets these structure members are in the file **common/EXCEPT.C**. These functions are: `set_mem_conf()`, `get_mem_conf()`.

The function `mem_sys()` in the file **common/IDTMEM.S** writes and reads back memory spaces. This function recognizes the fact that when the secondary cache is enabled on the R5000 residing on the 79S500 evaluation board, bursts from EPROM space do not function properly. It, therefore, tests to see whether the secondary cache is enabled, and behaves appropriately based on what it finds.

The function `cacheflush()`, which as the name suggests, flushes the caches, is defined in the file **common/IDTCMDS1.C**. This function calls the two functions `clean_scache_nowrite_indexed()` and `clean_scache_indexed()`, which are defined in the file **common/IDTC4000.S**. This file **common/IDTC4000.S** also defines the maximum and minimum sizes of secondary cache that the configuration will allow. This is done with `#define` as follow:

```
#define SCACHE_MIN_SIZE      0x0020000    /* minimum scache: 128Kb */
#define SCACHE_MAX_SIZE      0x1000000    /* maximum scache: 16Mb */
```

The lowest level function `_size_cache()` which is called by several of the functions referenced above, is also defined in **common/IDTC4000.S**. This is the function that actually obtains the size of secondary cache upon detection of R5000. The function `_init_cache()` in this same file initializes all caches including secondary cache. The function `flush_cache()` which flushes and invalidates all caches using cache-ops, is also in this file.

