MIPS® SDE 6.x Programmer's Guide

Document Number: MD00428
Revision 01.18
April 11, 2008

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043-1353

Template: nB1.03, Built with tags: 2B

Table of Contents

List of Tables

Chapter 1

Introduction

Chapter 2

Target-specific Libraries

SDE's run-time system provides an identical software interface across a range of different evaluation boards and software simulators, known here as "targets". The run-time system is provided as full source code for MTK customers, but as pre-compiled object files for most other users. Under the control of a per-target configuration file, it is built into a set of libraries specific to the chosen target. Much of the run-time code is generic and will work on any MIPS-based target, but drivers specific to a range of popular MIPS Technologies boards and simulators are included. For MTK customers, it is straight-forward to add a new target, as described in Chapter 9, "Retargeting the Toolkit" on page 89.

The supported target configurations are listed in Table 2.1. The columns are as follows:

- *Platform*: the evaluation board or software simulator.
- *CPU*: supported CPU types.
- *Base ISA*: base instruction set architecture. You can add variants like the MIPS16 ASE and the Release 2 extensions to this, see Section 2.1 "Building for ISA and CPU Variants".
- *FPU Type*: floating point hardware model. "None" implies software floating point; "64-bit" implies a 64-bit hardware FPU with the CPU's *Status.FR* bit set; and "32-bit" implies either a 32-bit FPU or a 64-bit FPU with the *FR* bit clear.
- *Endian*: CPU endianness. For a hardware target this must match the board's switch settings.
- *Connection*: how the *sde-gdb* debugger communicates with the target. "YAMON" implies a serial port connection to the YAMON Monitor; "MDI+EJTAG" is an EJTAG probe with MDI debugger interface; "MTSPMON" refers to the Linux AP/RP *pseudo-monitor.*
- *SDB*: "System Board Description", an identifier which describes this target to the SDE makefile system.

Table 2.1 Supported Target Boards and Simulators

Platform	CPUs	Base ISA	FPU Type	Endian	Debug Conn	SBD
MIPS Atlas™	4Kc, 4Km, 4Kp	MIPS32	None	BE	YAMON	ATLASLV4B
				LE		ATLASLV4L
MIPS SEAD-2™	4Kc, 4Km, 4Kp, 4KEc, 4KEm, 4KEp, 4KSc, 4KSd, M4K	MIPS32	None	LE	YAMON	SEAD32L
				BE		SEAD32B
				LE	MDI + EJTAG	SEAD32LJ
				BE		SEAD32BJ
	5Kf, 20Kc, 25Kf	MIPS32	32-bit	LE	YAMON	SEAD32FL
				BE		SEAD32FB
				LE	MDI + EJTAG	SEAD32FLJ
				BE		SEAD32FBJ
	24Kf, 24KEf, 34Kf	MIPS32 Release 2	32-bit	LE	YAMON	SEAD32F64L
				BE		SEAD32F64B
				LE	MDI + EJTAG	SEAD32F64LJ
				BE		SEAD32F64BJ
	5Kc	MIPS64	None	LE	YAMON	SEAD64L
				BE		SEAD64B
				LE	MDI + EJTAG	SEAD64LJ
				BE		SEAD64BJ
	5Kf, 20Kc, 25Kf	MIPS64	64-bit	LE	YAMON	SEAD64FL
				BE		SEAD64FB
				LE	MDI + EJTAG	SEAD64FLJ
				BE		SEAD64FBJ

Table 2.1 Supported Target Boards and Simulators (Continued)

Platform	CPUs	Base ISA	FPU Type	Endian	Debug Conn	SBD
MIPS Malta™	M4K	MIPS16e	None	LE	YAMON	MALTAM4KL
				BE		MALTAM4KB
				LE	MDI + EJTAG	MALTAM4KLJ
				BE		MALTAM4KBJ
	4Kc, 4Km, 4Kp, 4KEc, 4KEm, 4KEp, 4KSc, 4KSd, M4K, 5Kc, 24Kc, 24KEc, 34Kc, 74Kc	MIPS16e	None	LE	YAMON	MALTA16L
				BE		MALTA16B
				LE	MDI + EJTAG	MALTA16LJ
				BE		MALTA16BJ
	4Kc, 4Km, 4Kp, 4KSc, 5Kc	MIPS32	None	LE	YAMON	MALTA32L
				BE		MALTA32B
				LE	MDI + EJTAG	MALTA32LJ
				BE		MALTA32BJ
	4KEc, 4KEm, 4KEp, 4KSd, MrK, 24Kc, 24KEc, 34Kc, 74Kc	MIPS32 Release2	None	LE	YAMON	MALTA32R2L
				BE		MALTA32R2B
				LE	MDI + EJTAG	MALTA32R2LJ
				BE		MALTA32R2BJ
	5Kf, 20Kc, 25Kf	MIPS32	32-bit	LE	YAMON	MALTA32FL
				BE		MALTA32FB
				LE	MDI + EJTAG	MALTA32FLJ
				BE		MALTA32FBJ
	24Kf, 24KEf, 34Kf, 74Kf	MIPS16e	64-bit	LE	YAMON	MALTA16FL
				BE		MALTA16FB
				LE	MDI + EJTAG	MALTA16FLJ
				BE		MALTA16FBJ
		MIPS32 Release 2	64-bit	LE	YAMON	MALTA32R2FL
				BE		MALTA32R2FB
				LE	MDI + EJTAG	MALTA32R2FLJ
				BE		MALTA32R2FBJ
	34Kc, 34Kf	MIPS32 Release 2 + MT ASE	None	LE	MTSPMON	MALTA32LSP
				BE		MALTA32BSP
				LE	YAMON	MALTA32MTL
				BE		MALTA32MTB
				LE	MDI + EJTAG	MALTA32MTLJ
				BE		MALTA32MTBJ
	34Kf		64-bit	LE	YAMON	MALTA32MTFL
				BE		MALTA32MTFB
				LE	MDI + EJTAG	MALTA32MTFLJ
				BE		MALTA32MTFBJ

Table 2.1 Supported Target Boards and Simulators (Continued)

Platform	CPUs	Base ISA	FPU Type	Endian	Debug Conn	SBD
MIPS Malta™	5Kc	MIPS64	None	LE	YAMON	MALTA64L
				BE		MALTA64B
				LE	MDI + EJTAG	MALTA64LJ
				BE		MALT64BJ
	5Kf, 20Kc, 25Kf	MIPS64	64-bit	LE		MALTA64FL
				BE	YAMON	MALTA64FB
				LE		MALTAF64FLJ
				BE	MDI + EJTAG	MALTA64FBJ
MIPSSim	M4K	MIPS16e	None	LE	MDI	MSIMM4KL
				BE		MSIMM4KB
	4Kc, 4Km, 4Kp, 4KEc, 4KEm, 4KEp, 4KSc, 4KSd, M4K, 5Kc, 24Kc, 24KEc, 34Kc, 74Kc	MIPS16e	None	LE	MDI	MSIM16L
				BE		MSIM16B
	4Kc, 4Km, 4Kp, 4KSc, 5Kc	MIPS32	None	LE		MSIM32L
				BE	MDI	MSIM32B
	4KEc, 4KEm, 4KEp, 4KSd, MRK, 24Kc, 24KEc, 34Kc, 74Kc	MIPS32 Release2	None	LE		MSIM32R2L
				BE		MSIM32R2B
	5Kf, 20Kc, 25Kf	MIPS32	32-bit	LE	MDI	MSIM32FL
				BE		MSIM32FB
	24Kf, 24KEf, 34Kf, 74Kf	MIPS16e	64-bit	LE	MDI	MSIM16FL
				BE		MSIM16FB
		MIPS32 Release 2	64-bit	LE	MDI	MSIM32R2FL
				BE		MSIM32R2FB
	34Kc	MIPS32 R2 + MT ASE	None	LE	MDI	MSIM32MTL
				BE		MSIM32MTB
	34Kf		64-bit	LE	MDI	MSIM32MTFL
				BE		MSIM32MTFB
	5Kc	MIPS64	None	LE	MDI	MSIM64L
				BE		MSIM64B
	5Kf, 20Kc, 25Kf	MIPS64	64-bit	LE	MSIM	MSIM64FL
				BE		MSIM64FB

Table 2.1 Supported Target Boards and Simulators (Continued)

Platform	CPUs	Base ISA	FPU Type	Endian	Debug Conn	SBD
GNU Simulator	any	MIPS32	32-bit	LE	builtin	GSIM32L
				BE		GSIM32B
	any	MIPS16e	32-bit	LE	builtin	GSIM16EL
				BE		GSIM16EB
	any	MIPS64	64-bit	LE	builtin	GSIM64L
				BE		GSIM64B

The **SBD** column gives the short-form name of the board. This name identifies the sub-directory of `.../sde/kit` which contains the configuration files and possibly driver source code for this target. So, for example, the directory `.../sde/kit/MALTA32L` holds the target-specific information and code for MIPS Technologies' Malta board, with a MIPS32 CPU, without hardware floating point, little-endian, and debugging via a serial connection to the YAMON monitor.

To build the run-time library for one of the above targets, you simply go to its directory and run *sde-make*:

```
$ cd .../kit/MALTA32L
$ sde-makeIsde-make :
```

Having successfully built the library, you can then build any or all of the example programs. When building an example the first time, you need to specify the value of SBD on the `sde-make` command line:

```
$ cd \*[rootpath]/examples/hello
$ sde-make SBD=MALTA32L
```

This creates a file named `MALTA32L.sbd` in the working directory which records **`SDB`** and **`SDBTOP`**; further make makes will pick them up as default values. When you upgrade to a newer version of SDE, remove all generated files with:

```
$ sde-make clobber
```

Note: Specifying a different **`SDB`** value will cause the example *makefiles* to delete all object files and rebuild the program.

2.1 Building for ISA and CPU Variants

Due to the large range of processor cores and different ISAs and ASEs which are available on MIPS Technologies evaluation boards and simulators, the run-time libraries for the Malta and SEAD-2 evaluation boards and the MIPSSim simulator are configured for just a small number of base-level ISAs. If you want to build an application or benchmark that exploits a particular extended ISA or ASE, such as the MIPS32 ISA, or the SmartMIPS and MIPS16 ASE, then this is easily done when building your application by using the Makefiles' `APPISA` variable (see Section 3.2 “Example Makefiles”). Just pick the value of `SBD` which most closely matches your target board and CPU configuration, and then specify the extended ISA as follows:

```
$ cd .../sde/examples/ex5
$ sde-make SBD=MSIM32L APPISA=-mips32r2
$ sde-make SBD=MSIM32L APPISA="-mips32 -mips16"
$ sde-make SBD=MSIM32R2L APPISA="-mips32r2 -msmartmips"
$ sde-make SBD=MSIM32R2L APPISA="-mips32r2 -mdsp"
```

Similarly you can optimize the application for a specific CPU type using the **APPCPU** variable, for example:

```
$ cd .../sde/examples/dhrystone
$ sde-make SBD=MSIM32R2L APPCPU=74kc
```

Chapter 3

Example Programs

The `. . . /sde/examples` directory contains several small programs which demonstrate the use of SDE. They are each held in individual sub-directories, listed below, and they can all be built to execute in RAM under the control of a board's PROM monitor, or via an EJTAG probe, or (on some targets) blown into ROM, or run by a simulator.

All of the examples are built under the control of a common include file `. . . /sde/examples/make.mk`, which uses the board-specific parameters selected by the **`SBD`** variable to compile and link each program with the correct compiler flags and libraries.

We suggest that you first try building the examples and running them with the GNU simulator to see how they behave.

When you are happy with this, you can build the board-specific library for your target as documented in Chapter 2, “Target-specific Libraries” on page 9, and then rebuild the examples.

The remainder of this chapter describes the purpose of each example program.

3.1 Individual Examples

3.1.1 Hello World!

The program in `. . . /sde/examples/hello/hello.c` is simply everyone's first program - just to get you started!

3.1.2 TLB Exception Handling (tlbxcpt)

The example in `. . . /xde/examples/tlbxcpt` introduces SDE’s interface to low-level CPU exceptions. These are called *xcptions*, and are described in Section 7.2.1 “C-level Exceptions”. This program randomly accesses memory via the mapped KUSEG and KSEG2 regions (MIPS architecture magic words, read [Sweet99] if you don't know what they mean). On catching the resulting “TLB Miss” exceptions, it updates the TLB and returns to the faulting instruction. On completion it displays the number of TLB misses.

Note that some MIPS-Based CPUs don't have a TLB, and they will not be able to run this example.

3.1.3 Command Line Monitor (minimon)

This example provides a very simple command line monitor program, which is actually quite useful for peeking and poking devices on a new target, and can form the basis of useful command-line test harnesses. Type `help` at it for a list of commands.

One thing to note in this program is its use of POSIX *signal*-handling to catch address errors, and to test SDE’s interval timing functions (see Section 6.1 “POSIX API Environment”). In fact, the program was written and tested on a UNIX system before being ported to SDE.

This example might also be a good one with which to try out the *sde-gdb* debugger. If you reference an invalid address with the *put* or *get* commands (e.g. "**`g 1`**" will cause an address exception), then the debugger will be entered, allowing you to examine the cause of the exception.

Another useful piece of example code provided within this program is an ELF object file loader, which can load an ELF executable from a supported file-like device into memory - for example a flash ROM. See the `com_boot` function.

The ELF file loader is also capable of loading, relocating and then invoking a self-contained position-independent dynamic shared object (DSO) file. Self-contained means that the shared object must contain no undefined external references - the loader isn't yet smart enough to resolve symbols. You can try this out on a simulator target, as follows:

1. Build and run the minimon example for a simulator target, for example:

   ```
   $ sde-make SBD=MSIM32L
   ```

2. Build the example DSO as follows:

   ```
   $ sde-make SBD=MSIM32L dso
   ```

3. Load and run the minimon example on a simulator:

   ```
   $ [gdb-cmd] miniram
   (gdb) target mdi 15:1
   (gdb) load
   (gdb) run
   minimon> boot dso
   ```

3.1.4 Floating Point Test (paranoia)

The source file `.../sde/examples/paranoia/paranoia.c` is a public domain program, originally written by one of the creators of the IEEE-754 floating point standard. It is used to test many aspects of the standard: from the basic arithmetic, to the niggly rounding modes, overflow, underflow etc. We use it to test our software floating point emulation. You can use it to check that the floating point infrastructure of SDE is correctly installed and configured for your target.

3.1.5 Dhrystone Benchmark

The well known *dhrystone* benchmark (version 2.1) is in `.../sde/examples/dhrystone/dhry.c`. It serves as an example of how to port a simple integer-only benchmark. It only required configuration to use the ISO/ ANSI `clock()` function for its timing, and a minor change to disable it from attempting to write its results to a disk file.

The makefile for this example switches on high optimization (**`-O3`**).

Note that when using the MIPSSim simulator, the elapsed time for benchmarks is calculated from the simulator's cycle count, and then assuming that the simulated CPU is running at only 100\|kHz (with a 300MHz PC that will actually be close to real time, since the simulator runs at about 3000 instructions to 1) - you'll then have to scale the elapsed time to get a correct result for the expected target CPU frequency (e.g. for a 250MHz target divide the elapsed time by 2500, or multiply the benchmark result by 2500).

The GNU simulator can be used to debug benchmark programs like *dhrystone*, but it is an “instruction” simulator only. It makes no attempt to be cycle accurate, and does not simulate hardware timers or clocks, so programs will display a zero elapsed time. To get representative timings of simulated benchmark code you must use MIPSSim.

3.1.6 Whetstone Benchmark

The double-precision *whetstone* benchmark is in `.../sde/examples/whetstone/whetd.c`. It is an example of how to port a floating point benchmark. The only change was to make it use the ISO / ANSI `clock()` function to do its timing. It is built with high optimization (**`-O3 -ffast-math`**).

Note that software floating point emulation is enabled when compiling for the R3000 emulator.

For more information on the use of floating point, see Section 3.2 “Example Makefiles”.

3.1.7 Linpack Benchmark

Another well-known floating point benchmark is in directory `.../sde/examples/linpack`.

3.1.8 C++ Demo

This example builds a small C++ program: `.../sde/examples/cxxtest/tstring.cc` is a string handling test program from the GNU *libstdc++* library. If you would like to contribute a more interesting self-contained example, then please let us know!

3.1.9 Kit Test

This example, `.../sde/examples/kittest/hello.c`, is another “Hello World” program, but one which has a real purpose: it contains code that performs a simple confidence test of your target's memory system, serial port, “system interface” code, and library I/O functions.

If you are retargeting SDE to a new board, then you must make sure that this program runs before any other - basic console output must work before you stand a chance with anything more complex. In particular don't try to use the SDE remote debug stub with this example, since the debug facility uses precisely the code that you are testing here. So if your new target-specific code doesn't work well enough to run this program and talk to a serial port, then you'll need to debug it with an EJTAG probe, a logic analyser, or a pre-existing PROM monitor.

3.1.10 Flash Memory Test

The example program in `.../sde/examples/flash/flashtest.c` tests a board's Flash memory system (programming and erasing) and demonstrates use of the facilities described in Section 6.1.4 “Flash Memory Devices (/dev/flash)”.

Note that the Makefile defines `FEATURES=flashdev` to include the Flash device driver in the build, See Section 3.2 “Example Makefiles” for details.

3.1.11 PCI Bus Demo

The example program in `.../sde/examples/pci/pcitest.c` demonstrates how to setup, probe and access a board's PCI bus and PCI devices using the facilities described in Section 6.1.9 “PCI Bus Support”.

The example enumerates all devices on the bus and displays their configuration space registers symbolically. If the device has a boot ROM (and the target is running little-endian), then the ROM is accessed and its headers are decoded.

3.1.12 Decompressing Boot Loader

The example program in `.../sde/examples/zload/zload.c` is a small decompressing boot loader which could be used to load into RAM an application which is too big to fit into ROM. It also demonstrates use of the front-panel display device described in Section 6.1.5 "Alpha Display (/dev/panel)".

Note that the Makefile defines `FEATURES=paneldev` to include the front-panel display driver in the build (see Section 3.2 "Example Makefiles" for details).

The Makefile will automatically compile and link a tiny program `exec.c` into an ELF executable file and compress it. If you then run this example program on a simulator, or other target which support virtual host I/O, then it will read the compressed program, decompress it, load it into memory, and call it.

3.1.13 Linux AP/RP Communication

The example program in `.../sde/examples/rtlx/rtlx.c` demonstrates the low-level communication mechanism between a program running in the "Real-time Processor" of a multi-VPE MIPS CPU, communicating with a Linux kernel device driver running on the "Application Processor". It uses the character device files described in Section 6.1.3 "Linux AP/RP Communication (/dev/lx#)", which will only work in conjunction with one of the target board kits which support the mtspmon interface, namely: `MALTA32LSP` or `MALTA32BSP`.

3.1.14 Interrupt Example

The example program in `.../sde/examples/spxcpt/spxcpt.c` demonstrates how to install an interrupt handler on the Malta platform. It installs an interrupt handler which updates the LED display every 0.1 seconds.

3.2 Example Makefiles

Each example sub-directory contains the source of the program and a *makefile*. Each makefile defines a few variables and then includes the common file `.../sde/examples/make.mk`. This rather complicated makefile uses the board-specific parameters defined in the kit directory `.../sde/kit/$SBD/sbd.mk` to build each program with the correct combination of compiler flags and libraries to match the CPU type, endianness, floating point hardware, etc. on the selected target board.

The default action of `make.mk` is to build three versions of your program: downloadable using ROM monitor, downloadable but with its own I/O routines, and rommable. So for example the dhrystone benchmark makefile, which defines.command `PROG=dhry`, will generate (along with a number of intermediate files) the files shown in Table 3.1.

Table 3.1 Example Makefile Output Files

Filename	Purpose
`dhryram`	An executable file linked for downloading into RAM, and running with the board's PROM monitor. Some monitors can load this file directly over Ethernet.

Table 3.1 Example Makefile Output Files (Continued)

Filename	Purpose
dhryram.dl	The above executable, converted into a format suitable to transfer over a serial link to the board. The ".dl" is one of the formats supported by the sde-conv program.
dhrysa	A standalone executable file, linked for a RAM address, but once downloaded and started is independent of the PROM monitor (i.e. it includes its own UART drivers, etc).
dhrysa.dl	The standalone executable converted into download records, suitable for your PROM monitor.
dhryrom	A rommable executable file - it may relocate itself to RAM if required for debugging, or if requested by the **LAYOUT** variable (see below).
dhryrom.s3	The rommable executable, converted into Motorola S-records ready to transfer to your PROM programmer.
dhryrel	For AP/RP configurations, a relocatable version of your program which can be loaded at run-time by the operating system.

The operation of make.mk can be further controlled by setting additional variables, in one of the following ways:

1. Specify the variables on the command line, for examp[le:

```
$ sde-make SBD=MALTA32L APPISA="-mips32 -mips16e"
$ sde-make SBD=MSIM32R2L APPCPU=4ksd APPISA="-mips32r2 -msmartmips"
$ sde-make SBD=MSIM32R2L APPCPU=24kec APPISA="-mips32r2 -mdsp"
```

Note the use of quotes around the command-line values which contains spaces.

2. Edit one of the example *makefiles* only, so that just that one program is affected, and add lines which define the relevant variables, for example:

```
SBD=MALTA32R2FL
APPCPU=4ksd
APPISA=-mips32r2 -msmartmips
```

Note how, in a Makefile, values with spaces do not require quotes.

3. Add the same lines to . . . /sde/examples/make.mk so that they will apply globally to all *makefiles* which use it.

4. Set them as environment variables. For example, with Bourne shell or similar:

```
$ SBD=MALTA32R2FL; export SBD
$ APPCPU=4ksd; export APPCPU
$ APPISA="-mips32r2 -msmartmips"; export APPISA
```

or with C shell:

```
% setenv SBD MALTA32R2L
% setenv APPCPU 4ksd
% setenv APPISA "-mips32r2 -msmartmips"
```

You can have the environment variables set every time you use the software by editing a startup script.

The list of variables that you may want to change is shown in Table 3.2.

Table 3.2 User-Changeable "Make" Variables for Program Building

Variable Name	Default Value	Permissible Values	Description
ALL	rom ram sa	*any*	The default list of files to build.
APPCPU	$(CPU)		Override the default CPU type.
APPISA	$(ISA)		Override the default ISA.
ASFLAGS	$(FLAGS)	*any*	Assembler flags
CFLAGS	-02 -g	*any*	C compiler flags
CPPLAGS		*any*	C pre-processor flags (e.g., -D, -U, -A, etc.) to use when compiling the application source code.
CRT0FLAGS		*any*	Additional C pre-processor flags for customizing the crt0.o startup module.
		-DMINKIT	Don't de-initialize full POSIX run-time library. See Section 4.1.3 "Minimal C Library".
		-DSMALLXPT	Don't initialize early exception handling. See Section 4.1.3 "Minimal C Library".
		-DNOTORDTOR	No support for constructors and destructors. See Section 4.1.3 "Minimal C Library".
		-DNOFEATUREINIT	No initialization of library functions. See Section 4.1.3 "Minimal C Library".
CXXFLAGS	-02 -g	*any*	C++ compiler flags
FEATURES	A list of run-time "features", separated by spaces, which you want to include or exclude from your application. Wild-cards can be specified using the '%' character, e.g. FEATURES=pci%. To request an feature optionally, prepend a'/' character, e.g. FEATURES=/paneldev. The currently supported feature list is:		
		all	Include all optional run-time features supported on this board. To then explicitly exclude some features, append the feature names preceded by '-', e.g. FEATURES=all -pci%.
		flashdev	The /dev/flash interface. See Section 6.1.4 "Flash Memory Devices (/dev/flash)".
		paneldev	The /dev/panel interface. See Section 6.1.5 "Alpha Display (/dev/panel)".
		pci	The PCI bus scanning and initialization code. This will be included automatically if any of the PC I support functions are called by your code.
		pcilookup	Lookup table to translate known PCI vendor and device IDs to readable names. This table currently occupies 40KB and will only grow!
		unaligned	Install an unaligned address exception handler to fix up occasional unaligned accesses. But don't use this in production code, it will be very slow!
		xcptstackinfo	Stack backtrace on fatal exception (default in ROM code with remote debugging enabled).

Table 3.2 User-Changeable "Make" Variables for Program Building (Continued)

Variable Name	Default Value	Permissible Values	Description
FLOAT	no	no	Floating point is not used.
		yes	Basic floating point support is required.
		ieee	Full IEEE-754 conformance (Note that this may increase program size significantly).
LAYOUT	rom	rom	Copy only initialized data to RAM; run code from ROM.
		romcopy, ram	Copy both code and initialized data from ROM to RAM for better performance, or to set software breakpoints. This is the default if RDEBUG=imm is specified.
LDFLAGS		*any*	Additional linker flags.
LDSCRIPT		*any*	Custom linker script which overrides the standard one.
LDLIBS			Additional local libraries on which your program is dependent, and which to link with program.
LIBCC			C++ I/O stream and basic class library.
LOADLIBES		*any*	Additional standard libraries to link with your program (e.g. -lm).
NODEBUG	no	no	Produce source-level debugging information.
		yes	Do not produce source-level debugging information - unless you add **-g** to CFLAGS.
OBJS		*any*	Optional list of object files which comprise the program.
PROFILE	no	no	Do not generate or collect profiling code or data.
		yes	Generate code to collect normal *gprof* profiling data (time in each function and call graph).
		lines	Generate code to collect line-by-line *gprof* profiling data.
		feedback-generate	Generate code to collect profiling information which can be fed back to the compiler
		feedback-use	Optimize the program using data collected by running a program previously built with feedback-generate.
		gcov	Generate code to count branches, and the extra data required by the *gcov* code-coverage program.
PROG		*any*	Name of final executable file. See previous table. If you are now (or may ever be) using Windows, remember to pick file names which fit within the file extension conventions of the Windows filesystem, and ensure your file names are still unique after ignoring differences between upper and lower-case letters.
RDEBUG	no	no	Don't include standalone remote debug stub.
		yes	Include remote debug stub.
		inmed	Include stub, and cause breakpoint before calling main().
SBD	NOSBD	various	Target board name: see Chapter 2, "Target-specific Libraries" on page 9.
SDETOP	../..	and	The SDE kit and examples base directory, relative to the example directory - but you can also specify an absolute pathname.

Table 3.2 User-Changeable "Make" Variables for Program Building (Continued)

Variable Name	Default Value	Permissible Values	Description
SRCS			List of source files comprising program.
UNCACHED	no	no	Link the program to run cached.
		yes	Link the program to run uncached - for tracing with a logic analyser, for example.

You should rebuild your program from scratch whenever you change any *makefile* parameter. You can delete the old object files easily by running the command "**sde-make clean**".

You can generate a "standalone" Makefile for any example program which is customized to your selected SBD setting, which may help you to generate your own Makefile when you don't need the full multi-target flexibility of the SDE build system. Do this by running "**sde-make SDEmakefile SBD**=xxx", and then try it out by running "**sde-make -f SDEmakefile**".

Note that `.../sde/examples/make.mk` also includes the file `.../sde/kit/rules.mk`, which defines some additional compilation rules, for example to add support for the "`.sx`" file extension (which identifies assembler files that need to be passed through the C pre-processor, which is equivalent to gcc's handling of the "`.S`" extension, but compatible with Windows, which can't distinguish upper and lower-case file names).

Chapter 4

Standard Libraries

4.1 ISO / ANSI Library

SDE's library (`libc.a` and specified to the linker as **-lc**) follows the ISO Standard (ISO 9899:1990[1992]), also known as ISO 90, and formerly the ANSI X3J11 committee's standard for the programming language, It has been validated using the Plum Hall Validation Suite. The full ISO/ANSI specifications are long and careful, so this section lists only differences from the standard as described in Appendix B of *The C Programming Language* by Kernighan and Ritchie [Kern88] - yet another reason to invest in that essential volume.

Note that a number of the functions in the library assume the existence of a POSIX-like "operating system" interface, which is not included as part of the library. The notable omissions are listed below, and one possible implementation of them is contained in the embedded system kit, which can be used "as-is", or modified, or even completely replaced to suit your particular requirements.

Input and Output: <stdio.h>

All functions are supplied. However, the *stdio* functions in the library themselves call externally supplied low-level POSIX file I/O primitives. If your program is running on one of the boards supported by SDE's run-time system, then it contains "drivers" which implement the file I/O primitives. If not, or if you don't want to use our kit, then you will have to provide these routines yourself. They must have the standard POSIX semantics:

```
int open (const char *path, int flags, .../*int mode*/);
int close (int fd);
ssize_tread (int fd, void *buf, size_t n);
ssize_twrite (int fd, const void *buf, size_t n);
longlseek (int fd, long off, int whence);
int fstat (int fd, struct stat *stb);
int ioctl (int fd, unsigned long cmd, ...);
```

The *stdio* functions only support the UNIX-style line ending convention, e.g. 'n' is always written as a single line-feed character. The ISO / ANSI-specified "b" mode can be given to `fopen` etc., and this is passed to `open` as the `O_BINARY` flag bit. It is then up to the read and write "system calls" to do any translation that might be required.

Character Class Tests: <ctype.h>

All functions are supplied.

String Functions: <string.h>

All functions are supplied.

Mathematical Functions: <math.h>

All ANSI C floating-point functions are supplied (with additions from IEEE-754) in a separate maths library named `libm.a`, and specified to the linker as **`-lm`**. This library is based on code developed at the University of California,

Berkeley. We have assembler-coded some key functions (`drem`, `rint` and `sqrt`). There are two additional, non-standard functions which accept and return single-precision floating point values, namely:

```
/* single-precision square root */
floatsqrtf (float);

/* single-precision absolute */
floatfabsf (float);
```

Utility Functions: <stdlib.h>

All functions are supplied.

The *malloc* family requires an external function with which to obtain sequential, contiguous blocks of memory:

```
void *sbrk (int nbytes);
```

Note that `nbytes` may be negative if memory is being returned to the "system" from the end of the memory pool (although this is not used by the existing *malloc*). A rudimentary implementation of `sbrk` is supplied in our standard run-time system.

Diagnostics: <assert.h>

Supplied.

Variable Argument Lists: <stdarg.h>

Supplied, together with the old <*varargs.h*> version.

Non-local Jumps: <setjmp.h>

Supplied.

Signals: <signal.h>

These functions are not implemented in the C library itself, as they are operating-system dependent. The header file is present, and a simple implementation of the POSIX *signal* handling functions is provided in our standard run-time system. See Section 6.1.6 "Signal Handling".

Date and Time Functions: <time.h>

All functions are supplied, except for the hardware dependent `clock()` and `time()` functions, which are implemented in our standard run-time system, see Section 6.1.7 "Elapsed Time Measurement".

Implementation-defined Limits: <limits.h> and <float.h>

Supplied.

4.1.1 ISO C99 Library Support

Support in the SDE C library and associated header files for the new ISO C99 standard is by no means complete, but the C99 <*stdint.h* > and <*inttypes.h* > header files are provided, and the `printf()` and `scanf()` family of functions support the new C99 formatting codes. There are likely to be more C99 features appearing in future releases.

4.1.2 Thread Safety

The SDE C library can be made fully thread-safe and reentrant, using the SDEthreads API to protect shared data and manage thread local storage. This API is defined by the header file <*sdethread.h*>. Any RTOS wishing to use the SDE libraries in a thread-safe manner must implement a simple glue or "shim" layer, mapping from the SDEthreads API to its own primitives. A dummy version of the SDEthreads API, suitable for single-threaded code only, is provided in the file `...sde/kit/share/stubs.c`, and can be used as a model.

MIPS Technologies offers a number of Thread Support Packages (TSPs) which integrate popular RTOS's with SDE - contact us for the current list.

4.1.3 Minimal C Library

If program size is critical, and you do not need access to the full-blown library facilities, then you can significantly reduce the amount of the library that is linked into your program by avoiding the use of the high-level *Input and Output* functions described above. To output console messages in this case you must call only the functions `_mon_putc()`, `_mon_puts()` and `_mon_printf()` functions, which have identical interfaces to their `stdio` equivalents, except that they talk directly to the PROM monitor or your hardware; also the `_mon_printf()` function does not support floating point. For console input you can use `_mon_getc()` to read a single character at a time.

When your application is known to have limited requirements for its runtime environment, you can reduce the code size further by adding the some of the following definitions to your application *Makefile*.

- If your application does not need de-initialization features like `atexit()`, and a simplified stacktrace when unhandled exceptions occur, then use

```
RT0FLAGS += -DMINKIT
```

- If your application doesn't use exceptions, you can avoid the inclusion of exception handlers with

```
RT0FLAGS += -DSMALLX PT
```

Note that this disables handling of all sorts of exceptions, including those caused by hardware faults.

- If your application doesn't use constructors or destructors, you can disable their support with

```
LDFLAGS += -nostartfiles
RT0FLAGS += -DNO TORDTOR
```

Note that some third party libraries may rely on the availability of this feature.

- If you don't need any of the optional kit FEATURES, then you can disable the initialization code via

```
RT0FLAGS += -DNOFEATUREINIT
```

4.2 IEEE-754 Floating Point Emulation Library

SDE's floating point emulation library is named `libe.a`, and specified to the linker as **`-le`**. It implements single- and double-precision IEEE-754 floating point, but using only integer instructions. It is invoked either directly by subroutine calls from your program (if you specify the **`-msoft -float`** compiler options), or from a trap-based FPU instruction emulator (to fix up exceptional conditions, or when your code was built for a hardware FPU which is absent).

There is no external documentation for this library, other than the header file *<ieee754.h >*.

The library includes two copies of the same code, compiled with different options:

1. A pedantic emulation of the MIPS floating point unit, which is used to implement the trap-based FPU hardware emulation. This uses function names like `ieee754dp_add`.

2. A soft-float version which will be invoked by compiler-generated subroutine calls when compiled with the **`-msoft-float`** option. This version of the library has been tuned for speed by removing support for floating point exceptions, flag bits, and rounding modes other than “round to nearest”. These functions have names like `_adddf3`.

You'll find a primer on floating point and its implementation in the MIPS architecture in [Sweet99].

4.3 Multilibs

SDE can generate code for a large range of MIPS ISAs, and variants such as endianness, register size, soft/hard floating point, an so on.

In order to support this the standard libraries are supplied in many different flavors, organized into directory hierarchies below `.../sde/lib` and `...lib/gcc-lib/sde/`*`compiler-version`*.

This mechanism is known as *gcc multilibs*, and when you link your program using the *sde-gcc* front-end, it automatically determines the directories which contain the libraries that match the compiler architecture flags that you specified.

As long as you use *sde-gcc* front-end to link your program, you don't really need to know how the library directories are organized. But if for some reason you need to use the raw linker (*sde-ld*), or you're just curious, then use this command:

```
$ sde-gcc[your options] --print-multi-directory
```

That will display the directory below `.../sde/lib` which holds the libraries which match your particular group of options. There may be no directory for combinations of options which don't make sense. The set of options which effect the choice of multilib are currently: **`-EB/-EL`**, **`-mips64`**/**`-mips32r2`**/**`-mips32`**, **`-mips16`**, **`-mhard-float`**/**`-mfp64`**/–**`msoft-float`**/**`-mno-float`**, and **`-mno-data-in-code`**.

4.4 Library Source Code

Customers who purchase MIPS32® Software Toolkit receive all of the libraries as source code, as well as in pre-compiled form. Most users will never need to recompile the libraries themselves, but the option is available in case you need to modify a library function, or build debugging or profiling versions of the libraries.

To rebuild the libraries simply change directory to the root of the library source code, and run *sde-make*, like this:

```
$ cd .../sde/libsrc
$ sde-make
```

That will build the library, maths library, and floating point emulation library in sub-directories `c/OBJ`, `math/OBJ`, and `ieee/OBJ` respectively. All supported *multilib* combinations will be built.

You can also override some of the compiler options like this:

```
$ sde-make DEBUG="-O0 -g" clean all
$ sde-make DEBUG="-pg" clean all
$ sde-make DEBUG="-pg -g" clean all
```

In the first case you'll build a “debuggable” version of the libraries, in the second a profiling version, and in the third case a profiling version with line-number information.

Finally you may want to install all of your newly built libraries, replacing the pre-built libraries that were supplied as part of SDE.

```
$ sde-make DESTROOT=/home/joe/sde-\*[relno] install
```

But beware: that will overwrite all of the supplied libraries, so you might want to make a copy of the original SDE libraries first, for safe keeping, e.g.:

```
$ cd /home/joe/sde-\*[relno]/sde
$ tar cf - lib | gzip -9 >lib-orig.tgz
```

Chapter 5

MIPS® Architecture Intrinsics

The MIPS architecture includes a number of instructions and registers that can't be accessed directly by C and C++ code. SDE includes a set of *intrinsics*, which provide access to these special-purpose instructions. They are often implemented in header files, using *gcc* inline *asms* - which means that you can read, modify and reuse them for your own purposes.

This chapter describes only application-level MIPS intrinsics - for intrinsics which access a CPU's “system” facilities, see Section 7.6 “System Coprocessor (CP0) Intrinsics”.

5.1 Intrinsics for Byte Swapping

Include the header file <*sys/endian.h*> to define the inline functions listed below. On a MIPS32 Release 2 CPU, they will generate a fast, two-instruction sequence; on other MIPS ISAs, they will generate a longer sequence of shifts, ands, and ors. They are also smart enough to byte-swap constants at compile time.

```
uint32_t htobe32(uint32_t val)
```

Convert the 32-bit value `val` from “host” byte order to big-endian byte order (this will be a no-op on a big-endian CPU).

```
uint16_t htobe16(uint16_t val)
```

Convert the 16-bit value `val` to big-endian format.

```
uint32_t betoh32(uint32_t val)
```

Convert 32-bit big-endian value `val` to the “host” byte order (this will be a no-op on a big-endian CPU).

```
uint16_t betoh16(uint16_t val)
```

Convert 16-bit big-endian value `val` to the “host” byte order.

```
uint32_t htole32(uint32_t val)

uint16_t htole16(uint16_t val)

uint32_t letoh32(uint32_t val)

uint16_t letoh16(uint16_t val)
```

As above, but converting to and from little-endian.

5.2 Intrinsics for MIPS32® Architecture

The MIPS32 and MIPS64 instruction set architectures include the count-leading-zeroes and count-leading-ones instructions. SDE provides this C interface, implemented by inline *asms* on MIPS32 and MIPS64 CPUs, or as a sub-routine call on older MIPS architectures. To use these functions include the header file <*mips/mips32.h*>.

`uint32_t mips_clz(uint32_t val)`

The 32-bit argument `val` is scanned from most-significant to least-significant bit, and the number of leading zeros is returned. If no bits were set, the value 32 is returned.

`uint32_t mips_clo(uint32_t val)`

The 32-bit argument `val` is scanned from most-significant to least-significant bit, and the number of leading ones is returned. If all bits were set, the value 32 is returned.

`uint32_t mips_dclz(uint64_t val)`

The 64-bit argument `val` is scanned from most-significant to least-significant bit, and the number of leading zeros is returned. If no bits were set, the value 64 is returned.

`uint32_t mips_dclo(uint64_t val)`

The 64-bit argument `val` is scanned from most-significant to least-significant bit, and the number of leading ones is returned. If all bits were set, the value 64 is returned.

5.3 Intrinsics for MIPS32® Release 2 Architecture

The MIPS32 Release 2 ISA introduces a number of new user-level instructions. Some of them will be happily used by the compiler to optimize normal C code. But some of the byte- and bit-shuffling instructions are not available for normal C code, so these intrinsics are made available by including <*mips/mips32.h*>:

`uint32_t _mips32r2_bswapw(uint32_t int val)`

Byte swap the 32-bit value `val`, a two-instructions sequence. It is normally more efficient to use the intrinsics described in Section 5.1 “Intrinsics for Byte Swapping”.

`uint32_t _mips32r2_wsbh(uint32_t val)`

Return the result of the MIPS64 Release 2 `wsbh` instruction given `val`.

`uint32_t _mips32r2_ins(uint32_t tgt, uint32_t val, uint32_t pos, uint32_t sz)`

Return the result of a 32-bit insert bit field instruction, inserting `sz` bits of `val` into `tgt`, at bit position `pos`. Both `pos` and `sz` must be constants.

`uint32_t _mips32r2_ext(uint32_t x, uint32_t pos, uint32_t sz)`

Return the result of a 32-bit unsigned extract bit field instruction, returning `sz` bits, from bit position `pos`, of `x`. Both `pos` and `sz` must be constants.

5.4 Intrinsics for MIPS64® Release 2 Architecture

TheMIPS64 ISA inherits the MIPS32 instructions and their intrinsics, but (as one would expect) adds some 64-bit equivalents:

`uint64_t _mips64r2_bswapd(uint64_t val)`

Byte swap the 64-bit value `val`, a two-instructions sequence.

`uint64_t _mips64r2_dsbh(uint64_t val)`

Return the result of the MIPS64 Release 2 `dsbh` instruction given `val`.

`uint64_t _mips64r2_dshd(uint64_t val)`

Return the result of the MIPS64 Release 2 `dshd` instruction given `val`.

`uint64_t _mips64r2_dins(uint64_t tgt, uint64_t val, uint32_t pos, uint32_t sz)`

Return the result of a 64-bit insert bit field instruction, inserting `sz` bits of `val` into `tgt`, at bit position `pos`. Both `pos` and `sz` must be constants.

`uint64_t _mips64r2_dext(uint64_t x, uint64_t pos, uint32_t sz)`

Return the result of a 64-bit unsigned extract bit field instruction, returning `sz` bits, from bit position `pos`, of `x`. Both `pos` and `sz` must be constants.

5.5 Intrinsics for CorExtend^{CoreTrade} Extension

MIPS Technologies' Pro Series^{CoreTrade} CPU cores include the CorExtend feature, which extends the instruction set by adding a small number of user-definable instructions (UDIs). The Pro Series cores then provide an on-chip interface which allows a customer building an SoC to add just the logic to implement their chosen instructions; the interface to the CPU pipeline and its general-purpose registers is provided by the core.

The UDI instructions commonly have the standard MIPS three-operand format, where they can use two registers as source operands and one as destination. The two source registers are decoded inside the CPU core, and sent to the customer's UDI block, and so they can only be encoded in the standard position. The register number to which to write the result is selected by the UDI block, so in principle can be any CPU register or none, including one of the source registers; but it would be eccentric and unhelpful to specify a separate destination register and not use the standard MIPS format to do it. Instructions which don't use all the possible general purpose registers can recycle the register fields for other purposes.

The assembler interface to UDI provides you with choices about how you construct the instruction:

`udi IMM :`

All 24 user-definable bits of the instruction are set by integer `IMM`, including the register and opcode fields.

`udiOP IMM :`

`OP` is an integer (0 to 15) which defines the UDI opcode, and `IMM` the remaining 20 user-definable bits.

`udiOP rs,IMM :`

`OP` is the UDI opcode, `rs` the register number (read-only, or read-write), and `IMM` the remaining 15 bits.

`udiOP rs,rt,IMM :`

`rs` would conventionally be read-only, but `rt` read-only or read-write. `IMM` is the remaining 10 bits.

`udiOP rs,rt,rd,IMM :`

`rs` and `rt` would conventionally be read-only, and `rd` write-only, a conventional MIPS three-operand instruction, with `IMM` defining the remaining 5 bits.

If a register field in a UDI instruction isn't a general purpose register, but a register in the UDI block, or extra opcode bits, then use the `$n` syntax to insert a 5-bit immediate into the field, e.g. **`udi,$a0,$10,$v0,12`**.

In SDE you get an interface to the UDI instructions; you'll need to `#include <mips/udi.h>`.

The GNU compiler can optimize code around the `asm()` statements used to build this interface; and that's great. But some UDI instructions may alter internal state or registers in the UDI block which aren't visible to the compiler, making those optimizations incorrect. If your UDI instruction generates no state except for what it writes to the CPU destination register, then you can use the “safe” intrinsics, and the optimizer can work its magic.

In the description below, `OP` is the UDI opcode (0 to 15); `A` and `B` are any valid C or C++ scalar integer-valued expression, and `IMM` is a constant to fill the remaining instruction bits. The compiler allocates registers to hold the `A` and `B` source operands, and the result register.

```
/* Simple UDI instructiions are assumed to write a result to their final CPU
register operand, but may may have other side effects such as using or modifying
internal UDI registers, so they won't be optimized by the compiler. */

/* The `ri' single register intrinsic passes A in the RS field, and returns the new
RS register. IMM is the remaining 15 bits. */
typeof A mips_udi_ri (OP, A, IMM);

/* The 'rwi' two register intrinsic passes A in the RS field, and and returns the
new RT register. IMM is the remaining 10 bits. */
typeof A mips_udi_rwi (OP, A, IMM);

/* The 'rri' two register intrinsic passes A in RS, B in RT, and and returns the
new RT register. IMM is the remaining 10 bits. */
typeof A mips_udi_rri (OP, A, B, IMM);

/* The 'rrwi' three register intrinsic passes A in RS, B in RT, and returns the w/o
RD register. IMM is the remaining 5 bits. */
typeof A mips_udi_rrwi (OP, A, B, IMM);

/* Optimizable intrinsics for UDI instructions which read only the CPU source
registers and write to the destination CPU register only, and have no other side
effects, i.e. they only use and modify the supplied CPU registers. */
typeof A mips_udi_ri_safe (OP, A, IMM);
typeof A mips_udi_rwi_safe (OP, A, IMM);
typeof A mips_udi_rri_safe (OP, A, B, IMM);
typeof A mips_udi_rrwi_safe (OP, A, B, IMM);

/* The mips_udi_i() intrinsics use no register inputs, but return the value written
```

```
to the RS register (the input value is assumed discarded). */
uint32_t mips_udi_i (OP, IMM);
uint64_t mips_udi_i_64 (OP, IMM);

/* "NoValue" intrinsics for UDI instructions which don't write a result to a CPU
register, so presumably must have some other side effect, such as modifying an
internal UDI register. */
void mips_udi_nv (IMM);
void mips_udi_i_nv (OP, IMM);
void mips_udi_ri_nv (OP, A, IMM);
void mips_udi_rri_nv (OP, A, B, IMM);
```

To provide even more flexibility, the following set of intrinsics allow register fields in the UDI instructions to be set to constant 5-bit immediates (0-31), possibly to identify registers inside the UDI block, or as extra opcode bits. The `IS`, `IT` and `ID` arguments below must be constants, which will be inserted into the `rs`, `rt` and `rt` field of the instruction, as appropriate. Arguments `A` and `B` will still be computed and assigned to registers by the compiler.

UDI instructions are allowed to write to any general purpose register, not just those named in the instruction - so the destination register may be implicit in the opcode. To handle this the `GPDEST` argument allows the programmer to explicitly specify the general purpose register number that is written, and this prevents the compiler from allocating that register for other variables across the UDI instruction; if no general purpose CPU register is written, pass a GPDEST of zero.

```
/* These 4 variants of the three register operand format allow constant values to
be placed in the RS, RT fields, presumably because they name internal UDI
registers. The RD register is still allocated by the compiler. They are implicitly
"unsafe" or volatile. */
typeof A mips_udi_riri (OP, A, IT, IMM);
typeof B mips_udi_irri (OP, IS, B, IMM);
int32_t  mips_udi_iiri_32 (OP, IS, IT, IMM);
int64_t  mips_udi_iiri_64 (OP, IS, IT, IMM);

/* These 5 variants of the three register format allow constant values to be placed
in the RS, RT and RD fields, presumably because they name internal UDI registers.
In case the instruction writes to an implicit gp register, pass the register number
as GPDEST and the compiler will be told that it's been clobbered, and its value
will be returned - if no gp register is written, pass 0. They are all implicitly
unsafe, or volatile. */
typeof A mips_udi_rrii (OP, A, B, ID, IMM, GPDEST);
typeof A mips_udi_riii (OP, A, IT, ID, IMM, GPDEST);
typeof B mips_udi_irii (OP, IS, B, ID, IMM, GPDEST);
int32_t  mips_udi_iiii_32 (OP, IS, IT, ID, IMM, GPDEST);
int64_t  mips_udi_iiii_64 (OP, IS, IT, ID, IMM, GPDEST);
```

Warning: The compiler assumes that all *asm* inputs are “word sized”, i.e. that the inputs have the same precision as the underlying register size, and it may emit instructions to sign- or zero-extend any inputs which are smaller than that (e.g. char and short operands). To avoid an excessive number of these extension instructions you should try to ensure that you always pass “word sized” values to these intrinsics.

Warning 2: The GCC `asm` statement does not allow you to use aggregate values (a *struct*, *union* or *array*) as inputs or output for an `asm` - you may only pass simple scalar values. If you need to pass aggregate values to or from a UDI instruction, then you must define a union to smuggle them through. For example:

```
/* object manipulated by UDI hardware */
typedef struct {
  uint16_t imag;
```

```
  uint16_t real;
} complex_t;

/* access mechanism for UDI intrinsics */
typedef union {
  complex_t c;
  uint32_t  w;
} udicomplex_t;

/* add two complex types using three operand UDI instruction */
extern inline complex_t do_ADD (const complex_t *a, const complex_t *b)
{

  const udicomplex_t *ua = (udicomplex_t *) a;
  const udicomplex_t *ub = (udicomplex_t *) b;
  udicomplex_t uv;
  uv.w = mips_udi_rrwi_safe (ADD _OP ODE, ua->w, ub->w, 0);
  return uv.c;
}
```

5.6 Intrinsics for COP2 Extension

Some MIPS Technologies CPU cores allow an SoC builder to design a tightly-coupled coprocessor which implements theCOP2 instructions. These instructions are a part of the MIPS32 and MIPS64 ISAs reserved for use only by coprocessors. For the C interface to these instructions you must `#include <mips/cop2.h>`, which defines the following intrinsics:

```
void mips_lwc2 (C2REG, MEM);
```

Load the 32-bit word in memory referenced by `MEM` into COP2 data register C2REG (constant 0-31). The form of `MEM` is basically a 32-bit value obtained through a pointer, as in:

```
int *a;
mips_lwc2 (3, *a)
```

It's there so you can load a memory value directly into a COP2 register without loading it first into a general-purpose register.

```
void mips_swc2 (C2DREG, MEM);
```

The opposite - store COP2 data register C2REG to a memory location.

```
void mips_ldc2 (C2DREG, MEM);

void mips_sdc2 (C2DREG, MEM);
```

64-bit load/store respectively. Particularly important if your CPU has only 32-bit general purpose registers.

```
void mips_mtc2 (VAL, C2DREG, SEL);
```

Write any 32-bit expression VAL to COP2 register `C2DREG` in register bank `SEL`.

```
uint32_t mips_mfc2 (C2DREG, SEL);
```

Return the 32-bit COP2 register `C2DREG/SEL`.

```
void mips_dmtc2 (VAL, C2DREG, SEL);
```

```
uint64_t mips_dmfc2 (C2DREG, SEL);
```

64-bit versions of the above.

```
void mips_ctc2 (VAL, C2CREG);
```

Write any 32-bit C expression `VAL` to COP2 control register `C2CREG`.

```
uint32_t mips_cfc2 (C2CREG);
```

Return the 32-bit COP2 control register `C2CREG`.

```
void mips_cop2 (OP);
```

Emit arbitrary coprocessor 2 instruction with “undefined” bits set by constant integer `OP`.

```
int mips_c2t (CC);
```

Returns one if coprocessor 2 condition bit `CC` (0-7) is “true”, zero otherwise.

```
int mips_c2f (CC);
```

Returns one if coprocessor 2 condition bit `CC` is “false”, zero otherwise.

5.7 Intrinsics for SmartMIPS® ASE

MIPS Technologies' 4KSc and 4KSd CPU cores implement the SmartMIPS ASE (application specific extension) to the base MIPS32 instruction set. The bit-rotate and indexed load instructions will be used automatically by the compiler when you use the **`-msmartmips`** compiler option. The other new instructions may be used from C code by using the intrinsics defined by `#include <mips/smartmips.h>`, as follows:

```
int mips_multp (int a, int b)
```

Return the low 32-bit result of the polynomial-basis multiplication of the two, 32-bit binary polynomial arguments `a` and `b`.

```
int mips_maddp (int acc, int a, int b)
```

Return the low 32-bit result of the polynomial-basis multiplication of arguments `a` and `b`, polynomially added to `acc`. This can be used with `mips_multp` to construct a polynomial multiply-add loop which can be optimized by the compiler. For example:

```
int
maddp_arr (int *arr, int narr, int factor)
{
    int acc, i;
    acc = mips_multp (arr[0], factor);
    for (i = 1; i < narr; i++)
       acc = mips_maddp (acc, arr[i], factor);
```

```
        return acc;
   }
```

```
int mips_maddp2 (int a, int b)
```

Like `mips_maddp`, but assumes that you've already loaded the accumulator (the `LO` register) in some other way that is not visible to the compiler.

```
long long mips_multpx (int a, int b)
```

```
long long mips_maddpx (long long acc, int a, int b)
```

```
long long mips_maddp2x (int a, int b)
```

Like `mips_multp` etc, but operating on the full 64-bit multiplier result, i.e. the `HI`, `LO` register pair.

```
int mips_mfxu (void)
```

Return the extra high-order bits (bits 64 and upwards) of the multiply accumulator register (the new SmartMIPS `ACX` register). This is destructive of the accumulator, so use with care.

```
int mips_mfhu (void)
```

Return bits 32-63 of the multiply accumulator (the `HI` register). This is destructive.

```
int mips_mflhxu (int acc, int &lo)
```

Stores the low 32-bits of the multiply accumulator in `acc` into the lvalue “reference” argument `lo`, and then shifts the multiply accumulator right by 32-bits, returning the shifted accumulator. For example:

```
unsigned int
mpmadd (unsigned int *arr, unsigned int *spill, int narr, int factor)
{
  unsigned int acc = 0;
  int i, j;
  for (i = j = 0; i < narr; i += 4, j++) {
   acc += arr[i+0] * factor;
   acc += arr[i+1] * factor;
   acc += arr[i+2] * factor;
   acc += arr[i+3] * factor;
   acc = mips_mflhxu (acc, spill[j]);
  }
  return acc;
}
```

```
 long long mips_mflhxux (long long acc, int &lo)
```

Like `mips_mflhxu` etc, but operating on the full 64-bit multiplier result, i.e. the `HI`, `LO` register pair.

```
 void mips_mtlhx (int lo, int hi, int ex)
```

Moves the three 32-bit values in arguments `lo`, `hi`, and `ex` to the multiplier result registers (`LO`, `HI` and `ACX`).

```
void mips_pperm (int src, int sel)
```

Shift the 96-bit (max) extended multiplier result registers 6 bits left, and mix in 6 bits of `src`, permuted by `sel`. See the SmartMIPS **`pperm`** instruction definition for details.

5.8 Intrinsics for Paired-single/MIPS-3D® Architecture

This version of GCC includes support in the compiler for the paired-single SIMD floating point data type and instructions, and the MIPS-3D ASE. Full details of the vector data types and intrinsics can be found in the *Target Builtins* section of the [Gcc] Reference Manual.

5.9 Intrinsics for MIPS MT ASE

The new instructions introduced by the MIPS MT ASE may be accessed from code using the intrinsics defined by `#include <mips/mt.h>`, as follows:

```
unsigned int mips_mt_fork (void *addr, unsigned int pv, unsigned int cv)
```

Fork to `addr`, returning `pv` to parent and `cv` to child.

```
unsigned int mips_mt_yield (unsigned int yq)
```

Yield with qualifier `yq`, returning active signals.

```
int mips_mt_dmt (void)
```

Disable MT, returning old enable state.

```
int mips_mt_emt (void)
```

Enable MT, returning old enable state.

```
int mips_mt_dvpe (void)
```

Disable multi-VPE mode, returning old enable state.

```
int mips_mt_evpe (void)
```

Enable multi-VPE mode, returning old enable state.

Other functions in this header file provide access to the new Coprocessor 0 registers provided by the MT ASE, and to registers within other thread contexts. See Section 7.6 “System Coprocessor (CP0) Intrinsics” for a listing.

5.10 Intrinsics for MIPS DSP ASE

The MIPS DSP ASE defines a set of new instructions to improve the performance of DSP and “Media” applications.

Many of these new DSP instructions operate on Q15 or Q31 fractional data. Q31 is a 32-bit fixed-point fraction which can represent numbers between -1 and very nearly 1, and Q15 is a similar 16-bit fraction. The DSP ASE's favorite 8-bit quantity is an unsigned fraction representing numbers between 0 and 255/256.

Vectors of 4 x unsigned bytes or 2 x Q15 fractions fit into a 32-bit register, and the DSP ASE includes instructions which operate on all members of a vector at once. For detailed information about the MIPS DSP ASE (and a proper description of fractional data types), see the MIPS DSP ASE documentation [MD00374].

Addition and subtraction on fractional data are really the same as addition and subtraction with unsigned integer data, but multiplication requires a post-multiply shift to align the resulting values appropriately. The new multiply instructions in the DSP ASE that operate on fractional data provide this shift operation.

We do not (yet) have a compiler which knows about fractions. Q15 is an alias for a signed 16-bit integer (`short`), and Q31 is an alias for a signed 32-bit integer (`int`).

This document describes some new vector data types and built-in intrinsic functions available under the GNU C compiler. Each instruction in the DSP ASE has its own intrinsic, so you can write anything in C.

To tell GCC to compile for a CPU with DSP ASE support, pass the compiler the **`-mdsp`** flag.

5.10.1 Vector Data Types

Some typedefs:

```
typedef v4q7 __attribute__ ((mode(V4QI)));
typedef v2q15 __attribute__ ((mode(V2HI)));
typedef v4i8 __attribute__ ((mode(V4QI)));
typedef v2i16 __attribute__ ((mode(V2HI)));
```

`v2i16`

a vector of two 16-bit integers.

`v4i8`

a vector of four 8-bit integers.

`v4q7`

a vector of four Q7 fractions.

`v2q15`

a vector of two Q15 fractions.

You can initialize vectors like this:

```
v4i8 a = {1, 2, 3, 4};
v4i8 b;
b = (v4i8) {5, 6, 7, 8};

v2q15 a = {0x0fcb, 0x3a75};
```

Caution: When the compiler lets you see inside vectors and other packed data, you see the components in the order they occupy in memory when you store the vector. But instructions in the DSP ASE locate vector subcomponents with reference to register bit-numbers. The relationship between bit-numbers and memory addresses changes with the CPU's endianness; so initializers like this are endianness-dependent.

If you're big-endian, then at the C level you'll see the high-bit-number components first - the DSP ASE refers to these as *left* and uses an `l` (letter “l”, that is) in instruction names. If you're little-endian, then at the C level you'll see the lower-bit-numbered components first - what the DSP ASE calls *right* using an `r` in the instruction name. When little-endian, in fact, the one on the left is on the right: perhaps it's better to use a line break between the elements!

To initialize fractional values it's sometimes convenient to do this:

```
v2q15 b;
b = (v2q15) {0.1234 * 32768.0, 0.4567 * 32768.0};
```

The multiplication by `32768.0` effectively pre-shifts the decimal by 15 bits, which is just what you want for a Q15. To initialize a Q31 variable, you need a 31-bit shift, so multiply by `2147483648.0`.

You can use a union type to access vector components. Again, the relationship between the components named in your union and those seen by the DSP ASE will be endianness-dependent.

```
/* 'v4i8' Example */
typedef union
{
 v4i8 a;
 char b[4];
} v4i8_union;

v4i8 i;
char j, k, l, m;
v4i8_union temp;

/* Assume we want to extract from i. */
temp.a = i;
j = temp.b[0];
k = temp.b[1];
l = temp.b[2];
m = temp.b[3];

/* Assume we want to assign j, k, l, m to i. */
temp.b[0] = j;
temp.b[1] = k;
temp.b[2] = l;
temp.b[3] = m;
i = temp.a;

/* 'v2q15' Example */
typedef union
{
 v2q15 a;
 q15 b[2];
} v2q15_union;

v2q15 i;
q15 j, k;
v2q15_union temp;

/* Assume we want to extract from i. */
temp.a = i;
j = temp.b[0];
k = temp.b[1];

/* Assume we want to assign j, k to i. */
```

```
temp.b[0] = j;
temp.b[1] = k;
i = temp.a;
```

5.10.2 Scalar data types

```
#include <stdint.h>
typedef int32_t q31;
typedef int32_t i32;
typedef uint32_t ui32;
typedef int64_t a64;
```

`q31`

is really just an alias for a 32-bit signed integer, but an argument or return value with this type reminds you that the data is being interpreted as a Q31 fraction. Same goes for q15.

`i32`, `ui32`

are there for C purists, since there's no guarantee that a simple `int` is 32 bits.

`a64`

is an alias for `long long` (which for MIPS GCC is a 64-bit signed integer). We use it to remind you that the underlying instruction is using one of the four 64-bit accumulators defined by the DSP ASE (`$ac0`, `$ac1`, `$ac2`, `$ac3`). If you're already familiar with the MIPS architecture, note that `$ac0` comprises the bits of the `hi`/`lo` registers used in regular MIPS32 multiply/divide instructions.

Note that some parameters of builtin function have the following types:

`imm0_7`:

the parameter must be a constant in the range 0 to 7.

`imm0_15`:

the parameter must be a constant in the range 0 to 15.

`imm0_31`:

the parameter must be a constant in the range 0 to 31.

`imm0_63`:

the parameter must be a constant in the range 0 to 63.

`imm0_255`:

the parameter must be a constant in the range 0 to 255.

`imm0_1023`:

the parameter must be a constant in the range 0 to 1023.

`imm1_32`:

the parameter must be a constant in the range 1 to 32.

`imm_n32_31`:

the parameter must be a constant in the range -32 to 31.

5.10.3 Compiler Builtin Functions

The DSP ASE instruction names are full of "." (period) characters, not legal as part of C names. To make C names each period is replaced by "_" (underscore), and the assembler name prefixed with "`_builtin_mips_` ".

So the instruction called `addq.ph` becomes `_builtin_mips_addq_ph`. Note that where there are two variants of an underlying DSP instruction which accept an immediate or variable/register operand, the compiler will automatically pick the correct instruction depending on the type and size of the operand.

The instructions are listed in alphabetical order. Spaces have been introduced to separate unlike instructions, but there's no other hint as to what they do.

```
v2q15__builtin_mips_absq_s_ph (v2q15);
q31 __builtin_mips_absq_s_w (q31);

v2q15__builtin_mips_addq_ph (v2q15, v2q15);
v2q15__builtin_mips_addq_s_ph (v2q15, v2q15);
q31 __builtin_mips_addq_s_w (q31, q31);

i32 __builtin_mips_addsc (i32, i32);
i32 __builtin_mips_addwc (i32, i32);

v4i8__builtin_mips_addu_qb (v4i8, v4i8);
v4i8__builtin_mips_addu_s_qb (v4i8, v4i8);

i32 __builtin_mips_bitrev (i32);

i32 __builtin_mips_bposge32 ();

void__builtin_mips_cmp_eq_ph (v2q15, v2q15);
void__builtin_mips_cmp_le_ph (v2q15, v2q15);
void__builtin_mips_cmp_lt_ph (v2q15, v2q15);

i32 __builtin_mips_cmpgu_eq_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_le_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_lt_qb (v4i8, v4i8);

void__builtin_mips_cmpu_eq_qb (v4i8, v4i8);
void__builtin_mips_cmpu_le_qb (v4i8, v4i8);
void__builtin_mips_cmpu_lt_qb (v4i8, v4i8);

a64 __builtin_mips_dpaq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpaq_sa_l_w (a64, q31, q31);

a64 __builtin_mips_dpau_h_qbl (a64, v4i8, v4i8);
a64 __builtin_mips_dpau_h_qbr (a64, v4i8, v4i8);

a64 __builtin_mips_dpsq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsq_sa_l_w (a64, q31, q31);
```

```
a64 __builtin_mips_dpsu_h_qbl (a64, v4i8, v4i8);
a64 __builtin_mips_dpsu_h_qbr (a64, v4i8, v4i8);

i32 __builtin_mips_extp (a64, i32);
i32 __builtin_mips_extpdp (a64, i32);

i32 __builtin_mips_extr_r_w (a64, i32);
i32 __builtin_mips_extr_rs_w (a64, i32);
i32 __builtin_mips_extr_s_h (a64, i32);
i32 __builtin_mips_extr_w (a64, i32);

i32 __builtin_mips_insv (i32, i32);

i32 __builtin_mips_lbux (void *, i32);
i32 __builtin_mips_lhx (void *, i32);
i32 __builtin_mips_lwx (void *, i32);

a64 __builtin_mips_maq_s_w_phl (a64, v2q15, v2q15);
a64 __builtin_mips_maq_s_w_phr (a64, v2q15, v2q15);
a64 __builtin_mips_maq_sa_w_phl (a64, v2q15, v2q15);
a64 __builtin_mips_maq_sa_w_phr (a64, v2q15, v2q15);

i32 __builtin_mips_modsub (i32, i32);

a64 __builtin_mips_mthlip (a64, i32);

q31 __builtin_mips_muleq_s_w_phl (v2q15, v2q15);
q31 __builtin_mips_muleq_s_w_phr (v2q15, v2q15);

v2q15__builtin_mips_muleu_s_ph_qbl (v4i8, v2q15);
v2q15__builtin_mips_muleu_s_ph_qbr (v4i8, v2q15);

v2q15__builtin_mips_mulq_rs_ph (v2q15, v2q15);

a64 __builtin_mips_mulsaq_s_w_ph (a64, v2q15, v2q15);

v2q15__builtin_mips_packrl_ph (v2q15, v2q15);

v2q15__builtin_mips_pick_ph (v2q15, v2q15);
v4i8__builtin_mips_pick_qb (v4i8, v4i8);

q31 __builtin_mips_preceq_w_phl (v2q15);
q31 __builtin_mips_preceq_w_phr (v2q15);

v2q15__builtin_mips_precequ_ph_qbl (v4i8);
v2q15__builtin_mips_precequ_ph_qbla (v4i8);
v2q15__builtin_mips_precequ_ph_qbr (v4i8);
v2q15__builtin_mips_precequ_ph_qbra (v4i8);

v2q15__builtin_mips_preceu_ph_qbl (v4i8);
v2q15__builtin_mips_preceu_ph_qbla (v4i8);
v2q15__builtin_mips_preceu_ph_qbr (v4i8);
v2q15__builtin_mips_preceu_ph_qbra (v4i8);

v2q15__builtin_mips_precrq_ph_w (q31, q31);
v4i8__builtin_mips_precrq_qb_ph (v2q15, v2q15);
v2q15__builtin_mips_precrq_rs_ph_w (q31, q31);

v4i8__builtin_mips_precrqu_s_qb_ph (v2q15, v2q15);
```

```
i32 __builtin_mips_raddu_w_qb (v4i8);

i32 __builtin_mips_rddsp (imm0_63);

v2q15__builtin_mips_repl_ph (i32);
v4i8__builtin_mips_repl_qb (i32);

a64 __builtin_mips_shilo (a64, i32);

v2q15__builtin_mips_shll_ph (v2q15, i32);
v4i8__builtin_mips_shll_qb (v4i8, i32);
v2q15__builtin_mips_shll_s_ph (v2q15, i32);
q31 __builtin_mips_shll_s_w (q31, i32);

v2q15__builtin_mips_shra_ph (v2q15, i32);
v2q15__builtin_mips_shra_r_ph (v2q15, i32);
q31 __builtin_mips_shra_r_w (q31, i32);

v4i8__builtin_mips_shrl_qb (v4i8, i32);

v2q15__builtin_mips_subq_ph (v2q15, v2q15);
v2q15__builtin_mips_subq_s_ph (v2q15, v2q15);
q31 __builtin_mips_subq_s_w (q31, q31);

v4i8__builtin_mips_subu_qb (v4i8, v4i8);
v4i8__builtin_mips_subu_s_qb (v4i8, v4i8);

void__builtin_mips_wrdsp (i32, imm0_63);
```

5.10.4 Compiler Builtins for Second Revision

The second revision of the DSP ASE introduces some new instructions for which there are equivalent new builtin functions in the compiler.

```
v4q7 __builtin_mips_absq_s_qb (v4q7);

v2q15 __builtin_mips_addqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_addqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_addqh_w (q31, q31);
q31 __builtin_mips_addqh_r_w (q31, q31);

v2i16 __builtin_mips_addu_ph (v2i16, v2i16);
v2i16 __builtin_mips_addu_s_ph (v2i16, v2i16);

v4i8 __builtin_mips_adduh_qb (v4i8, v4i8);
v4i8 __builtin_mips_adduh_r_qb (v4i8, v4i8);

i32 __builtin_mips_append (i32, i32, imm0_31);
i32 __builtin_mips_balign (i32, i32, imm0_3);

i32 __builtin_mips_cmpgdu_eq_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_lt_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_le_qb (v4i8, v4i8);

a64 __builtin_mips_dpa_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dps_w_ph (a64, v2i16, v2i16);

a64 __builtin_mips_dpaqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpaqx_sa_w_ph (a64, v2q15, v2q15);
```

```
a64 __builtin_mips_dpax_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpsx_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpsqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsqx_sa_w_ph (a64, v2q15, v2q15);

a64 __builtin_mips_madd (a64, i32, i32);
a64 __builtin_mips_maddu (a64, ui32, ui32);
a64 __builtin_mips_msub (a64, i32, i32);
a64 __builtin_mips_msubu (a64, ui32, ui32);

v2i16 __builtin_mips_mul_ph (v2i16, v2i16);
v2i16 __builtin_mips_mul_s_ph (v2i16, v2i16);

q31 __builtin_mips_mulq_rs_w (q31, q31);
v2q15 __builtin_mips_mulq_s_ph (v2q15, v2q15);
q31 __builtin_mips_mulq_s_w (q31, q31);
a64 __builtin_mips_mulsa_w_ph (a64, v2i16, v2i16);

a64 __builtin_mips_mult (i32, i32);
a64 __builtin_mips_multu (ui32, ui32);

v4i8 __builtin_mips_precr_qb_ph (v2i16, v2i16);
v2i16 __builtin_mips_precr_sra_ph_w (i32, i32, imm0_31);
v2i16 __builtin_mips_precr_sra_r_ph_w (i32, i32, imm0_31);

i32 __builtin_mips_prepend (i32, i32, imm0_31);

v4i8 __builtin_mips_shra_qb (v4i8, i32);
v4i8 __builtin_mips_shra_r_qb (v4i8, i32);
v2i16 __builtin_mips_shrl_ph (v2i16, i32);

v2q15 __builtin_mips_subqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_subqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_subqh_w (q31, q31);
q31 __builtin_mips_subqh_r_w (q31, q31);

v2i16 __builtin_mips_subu_ph (v2i16, v2i16);
v2i16 __builtin_mips_subu_s_ph (v2i16, v2i16);

v4i8 __builtin_mips_subuh_qb (v4i8, v4i8);
v4i8 __builtin_mips_subuh_r_qb (v4i8, v4i8);
```

5.10.5 Intrinsics for Atomic R-M-W

SDE includes a set of atomic read-modify-write operations which provide fast, protected access to shared memory locations (but not device registers) in the face of interrupts. In the case of processors which support the `ll` and `sc` instructions, and have the appropriate external hardware, they will also be multi-processor safe. These facilities can be used to implement semaphores, mutexes, counters, etc.

To use these functions include the header file *<mips/atomic.h>*. The functions are as follows:

`uint32_t mips_atomic_bis(uint32_t *wp, uint32_t bits)`

The atomic bit “test-and-set” operation: sets those bits in `*wp` selected by non-zero bits in `bits` (e.g. `*wp |= set`), and returns the old value of `*wp`.

`uint32_t mips_atomic_bic(uint32_t *wp, uint32_t bits)`

The atomic bit “test-and-clear” operation: clears those bits in `*wp` selected by non-zero bits in `bits` (e.g. `*wp &= ~clr`), and returns the old value of `*wp`.

```
uint32_t mips_atomic_bcs(uint32_t *wp, uint32_t clr, uint32_t set)
```

A combined atomic bit “test-clear-and-set” operation: clears those bits in *wp selected by non-zero bits in `clr` and sets those selected by set (e.g. `*wp = (*wp & ~clr) | set`). Returns the old value of `*wp`.

```
uint32_t mips_atomic_swap(uint32_t *wp, uint32_t new)
```

The atomic “test-and-swap”, sets `*wp` to `new`, and returns the old value of `*wp`.

```
uint32_t mips_atomic_inc(uint32_t *wp)
```

Atomically increments `*wp`, returning its old value.

```
uint32_t mips_atomic_dec(uint32_t *wp)
```

Atomically decrements `*wp`, returning its old value.

```
uint32_t mips_atomic_add(uint32_t *wp, uint32_t val)
```

Atomically adds `val` to `*wp`, returning its old value.

```
uint32_t mips_atomic_cas(uint32_t *wp, uint32_t new, uint32_t cmp)
```

Atomic “compare-and-swap”: sets `*wp` to `new`, but only if it originally equals `cmp`. It returns the original value of `*wp`, whether or not updated.

Note that when the CPU does not include the `ll` and `sc` instructions, the operation is simulated, and will only be atomic if all interrupts are handled by the standard SDE exception handler, where there is special fixup code.

5.10.6 Intrinsics for Data Prefetch

Some MIPS-Based PUs support the `pref` instruction, which allows a programmer to optimize array processing loops (as used in many DSP algorithms) by explicitly prefetching the next block of data into the data cache before it is needed, to minimize the cache-miss latency of the following loads and stores. If it is done early enough, the data will already be in the cache by the time it is needed.

SDE includes a set of prefetch intrinsics to access these instructions. On CPUs which don't support the `pref` instruction these will be no-ops. To use the intrinsics include the header file <*mips/cpu.h*>.

```
void mips_prefetch (void *addr, int rw, int locality)
```

The value of `addr` is the address of the memory to prefetch. There are two further arguments: `rw` and `locality`. The value of `rw` is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero means that the prefetch is preparing for a read. The value `locality` must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. For example:

```
j = mips_dcache_linesize / sizeof (a[0]);
```

```
for (i = 0; i < n; i++)
{
a[i] = a[i] + b[i];
mips_prefetch (&a[i+j], 1, 1);
mips_prefetch (&b[i+j], 0, 1);
/* ... */
}
```

Data prefetch does not generate faults if `addr` is invalid, but the address expression itself must be valid. For example, a prefetch of `p->next` will not fault if `p->next` is not a valid address, but evaluation will fault if `p` is not a valid address.

Note that the `mips_prefetch` arguments match the `_builtin_prefetch` intrinsic in GCC 3.x, for which it is an alias.

`void mips_nudge (void *addr)`

The MIPS-specific “nudge” (push to memory) operation. The addressed cache line is written back to memory and invalidated.

`void mips_prepare_for_store (void *addr)`

The MIPS-specific “prepare for store” operation. If the addressed line is not already in the cache, then a line is allocated for it without reading memory (possibly flushing another line from the cache), and the line is cleared to zero. Warning: since this may zero the whole cache line, make sure that you only operate on cache line sized chunks, with cache line alignment.

Chapter 6

SDE Run-time I/O System

The SDE run-time system is a library that is built from our Embedded System Kit under the control of a board-specific configuration file. This chapter discusses the programming interfaces offered by the library.

The run-time system has two quite distinct parts: a high-level POSIX-like I/O system and environment; plus a collection of low-level CPU management and control primitives. We discuss the POSIX like system in this chapter, and the low-level CPU management in Chapter 7, “CPU Management” on page 61.

6.1 POSIX API Environment

The library, described in Section 4.1 “ISO / ANSI Library”, requires a set of low-level, UNIX-like file I/O primitives. The run-time system provides this I/O system, and a *signal* handling mechanism, both of which conform to the POSIX.1 definition. What are the benefits of this?

1. It is a well-documented, and well-known interface. See [POSIX88].

2. It shields the programmer from differences between various PROM monitor or simulator I/O systems. A program can be recompiled unchanged to run on any eval board or simulator supported by SDE.

3. A program will behave identically whether it is running in RAM, under the control of a board's monitor, or standalone in ROM.

4. It makes it very easy to port simple, self-contained programs from *[unix], Linux or other POSIX-compliant systems.

Although we refer you to [POSIX88] for documentation, the remainder of this section describes some of the details specific to this implementation. If your host system supports the POSIX interface (which is true for modern UNIX hosts, and the “Cygwin” environment on Windows) and you have the host's online “manual pages” available, then you'll find that those pages describe most of the functions listed here, and those in the SDE C library.

6.1.1 Remote File I/O

The run-time system implements a read-only POSIX file-system root, which contains a number of named special directories and devices which you can access via the standard POSIX file I/O primitives (e.g. `open`, `close`, `read`, `write`, etc). Note that you cannot create or delete directories and files in this file system, other than as documented below.

6.1.1.1 Host File Access

If your program is running on the GNU simulator, or you're using an MDI connection to your target via *gdb* (e.g. the MIPSSim simulator), then you have access to files on your host computer:

`/host/`*`path`*

refers to absolute pathname *path* on the host computer, e.g "/host/etc/passwd" refers to file /etc/passwd on the host.

/cdir/*path*

refers to file *path* on the host computer, relative to the debugger or simulator's current directory, e.g. "/cdir/Makefile" refers to file Makefile in your host's current directory.

/tmp/*path*

refers to file *path* relative to the host's /tmp directory.

Furthermore the run-time startup code performs an initial chdir() to the /cdir directory, so a simple file name without an initial '/' will refer - as you would hope - to a file in the debugger or simulator's current directory - this is handy for benchmark programs which expect to be able read and write their data files in the current directory.

6.1.2 Terminal I/O (/dev/tty)

The pseudo file-system also contains at least the following special device files (some boards may support more):

/dev/tty0

serial I/O port #0 - the first serial port.

/dev/tty1

serial I/O port #1 - the second serial port, if present.

/dev/console

the board's console, usually an alias for /dev/tty0.

/dev/tty

the "controlling terminal", also usually an alias for /dev/tty0.

All of these devices support a set of ioctl operations which implement the POSIX *termios* interface. These control: input line-editing, output processing, XON/XOFF flow control, baud rate control, "asynchronous" I/O notification, blocking/non-blocking reads, etc. When running under a PROM monitor some of the hardware control ioctl operations may have no effect, if they are not accessible via the PROM monitor's API - when running standalone or rommable code they will all be supported, because an SDE serial port driver will have full control of your UART.

Note that the "interrupt" character (default Ctrl-C) will raise a POSIX SIGINT signal, but the "quit" character (default Ctrl-\e) calls the abort() function, which will drop you into the debugger.

If you use the non-POSIX O_ASYNC flag when you open the tty device (or you use the fcntl(FASYNC) function on an open file descriptor), then the SIGIO signal will be raised when an input record is available (although note the comments on polling above).

6.1.3 Linux AP/RP Communication (/dev/lx#)

Programs which are built for targets which use the `mtspmon` "monitor" (currently `MALTA32LSP`, and `MALTA32BSP`), have access to eight character devices named `/dev/lx0-7`. These provide a basic byte stream interface between the SDE "standalone" code running on the Real-time Processor side, and the Linux device driver running on the Application Processor side.

The `rtlx` example program demonstrates the use of these devices (ee Section 6.1.3 "Linux AP/RP Communication (/dev/lx#)"). For more details on how to use this programming environment, see the Section 3.1.13 "Linux AP/RP Communication".

6.1.4 Flash Memory Devices (/dev/flash)

If your board kit includes support for Flash memory (see Chapter 9, "Retargeting the Toolkit" on page 89), there will be special device files with names in the following format:

`/dev/flash`*`N`*

Where *`N`* is the device number, starting from 0. This file provides access to the whole of the Flash memory device.

`/dev/flash`*`NP`*

Where *`N`* is the device number, and *`P`* the *partition* type. Each flash may be divided into a number of sub-partitions, as follows:

Table 6.1 Flash Memory Partition Types

Type	Description
`b`	Bootstrap (e.g. PROM monitor)
`t`	Test region (e.g. power-on test scratch area)
`e`	Non-volatile environment region
`f`	Data region #1 (e.g. flash file system)
`g`	Data region #2
`h`	Data region #3
`i`	Data region #4

To see whether your board kit supports and has detected Flash memory, build and run your application (see Section 3.1.3 "Command Line Monitor (minimon)") and use the command "`ls /dev`". You can also display the contents of the Flash using "`dump /dev/flash0`" or similar.

To include the `/dev/flash` interface in your build, you must define `FEATURES=flashdev` or `FEATURES=all` in your application Makefile (see Section 3.2 "Example Makefiles" for details). For a complete example of how to use the interface, see the example program in Section 3.1.10 "Flash Memory Test".

Each device can be opened, read and written using the standard POSIX file I/O functions (e.g. `open`, `read`, `write`, `lseek`, etc.), and therefore also the buffered *stdio* library functions (`fopen`, `fread`, `fwrite`, `fseek`, etc.). This means that you can develop an object file loader, for example, and debug it on a simulator reading from a host file, and then port the code to your target system where it can load from Flash memory. Or the Flash memory might be

used as a simple “file” in which to retain configuration data or store log output, and which can be read or written using the stdio library functions like `fscanf` or `fprintf`. A full Flash file system may be provided in future versions of SDE.

Note that although you can write to a Flash device one byte at a time, this will be very slow unless you are writing to an erased region (contains all ones).

The Flash device driver implements the following ioctls, as defined in the header file <*sys/flashio.h*>:

FLASHIOINFO

Returns the name (manufacturer and part number) of the Flash device, and its geometry, in the following structure:

```
struct flashinfo {
  char name[32];/* dev name */
  unsigned longbase;/* dev base (phys address) */
  unsigned intsize;/* dev size */
  unsigned longmapbase; /* memory mapped base (phys addr) */
  unsigned charunit;/* unit byte size (1,2,4,8 or 16) */
  unsigned intmaxssize; /* maximum sector size */
  unsigned intsoffs;/* base offset of specified sector */
  unsigned intssize;/* size of specified sector */
  int     sprot; /* specified sector is protected */
}
```

A pointer to this structure is passed as the *ioctl* parameter. If the `soffs` field is set to an offset within the device, then the returned structure will included the base offset of that sector, its size, and its protection status in the `soffs`, `ssize` and `sprot` fields respectively.

Note that when multiple Flash memory devices are organized in parallel banks, then all of the size fields will be multiplied accordingly. For example, if four byte-wide 1\|MByte devices are connected in parallel to a 32-bit data bus, then the unit size will be 4 bytes; the sector sizes will be multiplied by 4, and the total device size will be 4\|MBytes. If two banks are interleaved then the sizes will be doubled again.

FLASHIOGPART

Returns the type, offset and size of this partition within the whole device, in the following structure:

```
struct flashpart {
  int     type;  /* partition type */
  unsigned intoffs;/* base of partition */
  unsigned intsize;/* size of partition */
}
```

The type field is one of the following values:

FLASHPART_RAW

The whole device.

FLASHPART_BOOT

The boot partition (e.g. PROM monitor code).

FLASHPART_POST

Power-on self test (scratch) region.

FLASHPART_ENV

Non-volatile environment.

FLASHPART_FFS

Flash file system partition, free for data storage.

FLASHPART_UNDEF

Undefined type.

FLASHIOGFLGS

Returns the current device mode which controls how the device is read and programmed. The ioctl parameter should be a pointer to an *int*. The value contains the bitwise OR of the following bits:

FLASHFLGS_REBOOT

Reboot after next write.

FLASHFLGS_NO OPY

Don't copy programming code to RAM (normally it must be copied if your application is itself executing out of the Flash device).

FLASHFLGS_MERGE

Merge partial sector writes with existing sector data. If this flag is not set then a partial sector write will return an error if you write to a portion of unerased flash.

FLASHFLGS_ ODE

Some Flash memories must be programmed differently if they contain executable code, rather than being treated as a simple “byte stream”.

FLASHFLGS_STREAM

The default mode treats the Flash as a simple sequential byte stream.

The default value is: FLASHFLGS_MERGE | FLASHFLGS_STREAM.

FLASHIOSFLGS

Sets the current device mode which controls how the device is read and programmed. The *ioctl* parameter should be a pointer to an *int* containing the bitwise OR of the flag bits described above.

FLASHIOERASEDEV

Causes the whole Flash device to be erased. The *ioctl* parameter is ignored. Take care not to use this if your code is running in Flash!

FLASHIOERASESE T

Erases one Flash device sector. The *ioctl* parameter should be a pointer to an *unsigned int* holding an offset within the sector to be erased.

FLASHIOGPARTS

The *ioctl* parameter should be a pointer to an array of FLASHNPART *flashparts* structures, as described in FLASHIOGPART above. It will return the complete partition table for this device.

FLASHIOFLUSH 6

Forces any pending partial sector writes to written to Flash. This will happen automatically when the device is closed. The *ioctl* parameter is ignored.

6.1.5 Alpha Display (/dev/panel)

If your board kit includes support for an on-board or "front-panel" LED display, then there will be a special device file with the name "`/dev/panel`".

This device can be opened and written using the POSIX file I/O functions (e.g. `open` and `write`), and therefore also the buffered stdio library functions (`fopen`, `fprintf`, etc.). Each write to the device will by default be automatically preceded by an implicit *seek* to a fixed offset (default zero), and will thus overwrite the last message.

For an example of the use of the `/dev/panel` interface, see Section 3.1.12 "Decompressing Boot Loader".

The panel device driver also implements the following *ioctls*, as defined in the header file <*sys/panelio.h*>:

PANELIOINFO

Returns information about the display in the following structure:

```
struct panelinfo {
  unsigned chartype;/* display type */
  unsigned charflags;/* display facilities */
  unsigned charrows;/* number of rows or lines */
  unsigned charcols;/* number of columns per line */
}
```

A pointer to this structure is passed as the *ioctl* parameter. The type field will be one of:

PANELTYPE_ALPHA

Alphanumeric display

PANELTYPE_HEX

Hexadecimal display

PANELTYPE_LED

Individual LEDs

The *flags* field describes the capabilities of the display, as the bitwise OR of the following flags:

PANELFLGS_BRIGHTNESS

The display has variable brightness.

PANELFLGS_ ONTRAST

The display has variable contrast.

PANELFLGS_BLINK

The whole display can blink on and off.

PANELFLGS_FLASH

Individual characters or digits can blink.

PANELFLGS_S ROLL

The display can be scrolled if the message is longer than the display (not currently supported).

PANELFLGS_PROGRESS

The display has a bar graph or something similar, which can display the progress of a long operation.

PANELIOGMODE

Returns the current display mode in the following structure:

```
struct panelmode {
  unsigned charoptions;/* display options */
  unsigned charbrighton;/* brightness (0 to 100%) */
  unsigned charbrightoff;/* brightness (0 to 100%) */
  unsigned charcontrast;/* contrast (0 to 100%) */
  unsigned longblinkon;/* on period in ns */
  unsigned longblinkoff;/* off period in ns */
  unsigned longscrollrate;/* scroll rate in ns */
  int     scrollchars;/* scroll amount */
}
```

A pointer to this structure is passed as the *ioctl* parameter. The options field is the bitwise OR of the following bits:

PANELOPT_PAD 6

Pad short messages to the end of the display line with blanks.

PANELOPT_ CENTRE

Center short messages within each display line.

PANELOPT_WRAP

Wrap messages longer than one line onto the next line (if available), the default is to truncate the message at the end of the line.

PANELOPT_IGNLF

Line-feed (‘\en’) characters will not be treated specially; the default is to cause the following characters to start on the next display line (if available).

PANELOPT_IGNNUL

NUL characters will not be treated specially. The default is to treat them as the end of the message.

PANELOPT_ROTATE

Messages longer than one line will continually scroll/rotate. This is not yet supported.

PANELOPT_FADE

The display brightness will fade up and down, rather than simply flashing/blinking.

PANELOPT_FLASH

Characters in following writes will be flashed/faded.

PANELIOSMODE

Sets the current display mode. A pointer to the `panelmode` structure described above is passed as the *ioctl* parameter.

PANELIO LEAR

Clear the display. The *ioctl* parameter is ignored.

PANELIOPROGRESS

Update the panels' bar graph or similar to reflect progress through some long operation. The *ioctl* parameter is a pointer to an *int* with a value between 0 (min) and 100 (max).

PANELIOSCOORD

Sets the coordinate of the next output message, instead of the default <0,\|0>, from the following structure:

```
struct panelcoord {
  unsigned shortrow;
  unsigned shortcol;
}
```

A pointer to this structure is passed as the *ioctl* parameter. The value is sticky and will be used again on all following writes to the device.

6.1.6 Signal Handling

The run-time system includes an implementation of `sigaction()` and associated signal handling functions defined by [POSIX88], including `sigpending()`, `sigprocmask()`, `sigsuspend()` and `raise()`. Also included is the non-POSIX, but time-honoured `signal()` function. For an example of how these can be used, see example #3, as described in Section 3.1.3 "Command Line Monitor (minimon)".

For direct access to the lower-level CPU exceptions and interrupts see Section 7.2.1 "C-level Exceptions".

Table 6.2 lists of all the signals we use, with names as in the include file <`signal.h`>:

Table 6.2 POSIX Signal List

Name	Default Action	Description
SIGINT	Terminate program	Interrupt program (^ from terminal)
SIGILL	Terminate program	Illegal instruction
SIGTRAP	Terminate program	Debug (breakpoint) trap
SIGABRT	Terminate program	Abort() call
SIGFPE	Terminate program	Floating point exception / integer overflow
SIGKILL	Terminate program	Kill program
SIGBUS	Terminate program	Terminate program: bus error or alignment error
SIGSEGV	Terminate program	Segmentation violation (invalid address)
SIGSYS	Terminate program	System call trap
SIGALRM	Terminate program	Timer expired
SIGIO	Ignore signal	I/O is possible on a terminal
SIGVTALRM	Terminate program	Virtual time alarm (see `setitimer()` below)
SIGPROF	Terminate program	Profiling timer alarm (see `setitimer()` below)
SIGUSR1	Terminate program	User-defined signal 1
SIGUSR2	Terminate program	User-defined signal 2

6.1.7 Elapsed Time Measurement

If you need to read the current time, for performance measurement or logging, then see the standard ISO / ANSI `clock()` function, described in [Kern88], which returns the elapsed time in units of 1 microsecond; there is also the `time()` function which returns the current "wall clock" time, in units of 1 second. The <*time.h* > include file defines the following functions like this:

```
clock_t   clock (void);
time_t    time (time_t *);
```

Unlike a "real" POSIX operating system, the `clock()` function measures elapsed *real* time, not *cpu* time; in other words it **does** include time spent waiting for console input/output. When measuring performance, be careful to put calls to `clock()` around computational code only.

Alternatively you may prefer to use the POSIX `gettimeofday()` function, which returns the current "wall clock" time in both units and fractions of a second. The <*sys/time.h*> include file defines the following:

```
struct timeval {
    long  tv_sec;/* seconds */
    long  tv_usec;/* and microseconds */
};

struct timezone {
    int tz_minuteswest;/* ... of Greenwich */
    int tz_dsttime;/* type of DST correction */
};

int    gettimeofday (struct timeval *tvp, struct timezone *tzp);
```

You can pass a null timezone pointer, if you are not interested in that information.

6.1.8 Interval Timing

At the coarsest level, the `alarm(int secs)` function sets an interval timer which expires in `secs` seconds. A SIGALRM signal will be delivered when it expires.

More accurate timing facilities are modelled on those originally provided by POSIX.

The <*sys/time.h*> include file defines the following:

```
#define ITIMER_REAL0
#define ITIMER_VIRTUAL1
#define ITIMER_PROF2
#define ITIMER_USER3

int
getitimer(int which, struct itimerval *value)

int
setitimer(int which, struct itimerval *value, struct itimerval *ovalue)
```

The system provides four separate interval timers. The `getitimer()` call returns the current value for the timer specified by which in the structure at value. The `setitimer()` call sets a timer to the specified value (returning the previous value of the timer if `ovalue` is non-nil).

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
    struct timeval it_interval;/* timer interval */
    struct timeval it_value;/* current value */
    void  (*it_func)(struct timeval *, struct xcptcontext *);
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Note that interval timer values are rounded up to a multiple of 1 millisecond, and that timers are decremented in real time, i.e. no account is taken of whether a program is waiting for I/O or executing useful code.

A SIGALRM signal is delivered when the `ITIMER_REAL` timer expires.

A SIGVTALRM signal is delivered when the `ITIMER_VIRTUAL` timer expires.

A SIGUSR1 signal is delivered when the `ITIMER_USER` timer expires.

The `ITIMER_PROF` timer is used internally by the profiling system, and should not be used by applications.

The *itimerval.it_func* field is only valid for the `ITIMER_PROF` and `ITIMER_USER` timers. If non-null then the specified function is called directly at interrupt time, rather than sending a signal. The first argument passed to the function specifies the *delta* from the expected interrupt time (e.g. due to interrupt delays), and the second argument is the interrupt exception context (Section 7.2.3 "C-level Interrupts").

Three macros for manipulating time values are defined in <*sys/time.h*>; `timerclear()` sets a time value to zero; `timerisset()` tests if a time value is non-zero; and `timercmp()` compares two time values (beware that >= and <= do not work with this macro).

If the calls succeed, a value of 0 is returned. If an error occurs, the value -1 is returned.

For an example of how to use the asynchronous interval timing facilities, see the `com_itimer()` function in the example program #3, as described in Section 3.1.3 "Command Line Monitor (minimon)".

6.1.9 PCI Bus Support

On boards that have a PCI bus, and have implemented the necessary machine-dependent low-level support code, a generic interface to the PCI bus is provided to handle bus initialization, enumeration and address mapping.

Below we describe these functions in detail, and an example of their use can be found in Section 3.1.11 "PCI Bus Demo". In all cases you will need to add the following include directives to your source file:

```
#include <pci/pcivar.h>
#include <pci/pcireg.h>
```

```
void _pci_init (void)
```

Initializes the PCI bus controller and then scans the bus for devices, allocating address space for memory and I/O apertures and computing bus latency timers, etc; PCI-PCI bridges are also initialized and their buses scanned recursively. If running in RAM under control of a PROM monitor (e.g. PMON or IDT/sim), then the bus configuration is non-destructively scanned in order to determine the existing configuration. It is rarely necessary to call this function directly - it is called automatically at program initialization if any of the following PCI interface functions are used.

```
pcitag_t _pci_find (const struct pci_match *matchp, unsigned int matchnum)
```

Scans the PCI bus for the *matchnum*'th device (starting at zero) which matches the ID and lass values in the structure pointed to be *matchp*:

```
struct pci_match {
  pcireg_tclass, classmask;
  pcireg_tid, idmask;
}
```

A match succeeds when:

```
(([ O]device-class-reg & matchp->classmask) == matchp->class
 && ([ O]device-id-reg & matchp->idmask == matchp->id))
```

By using various combinations of mask value you can match all devices on the bus (`mask==id==0`), or all devices of a particular class and sub-class (e.g. `class==mass-storage` and `subclass==ide`), or a known manufacturer/device combination.

The function returns a PCI "tag" - a hardware-dependent token which represents the bus number, device number and sub-function number of the device's configuration space registers. It is passed to other functions below to gain access to other device registers and address spaces. When no matching devices are found, the function returns `~(pcitag_t)0`.

```
void _pci_break_tag (pcitag_t tag, int *busp, int *devp, int *funcp)
```

Converts the hardware-dependent tag into the individual bus, device and function number. If any *busp*, *devp* or *funcp* are null pointers, then that value is not returned.

```
void _pci_tagprintf (pcitag_t tag, const char *fmt, ...)
```

Calls the low-level `_mon_printf` function to print a diagnostic message, preceded by the string "`PCI bus` *`busno`* `slot` *`devno`*`/`*`funcno`*`:`".

```
void _pci_devinfo (pcireg_t id, pcireg_t class, char *bufp, int *supp)
```

Returns a printable form of the manufacturer, device name and type in the buffer pointed to by `bufp`, keyed on a device's ID and LASS config space registers. There is a large database of PCI devices, but it may not have yours! The final parameter `supp` should always be NULL.

```
pcireg_t _pci_conf_read32 (pcitag_t tag, int reg)

pcireg_t _pci_conf_read16 (pcitag_t tag, int reg)

pcireg_t _pci_conf_read8 (pcitag_t tag, int reg)
```

Returns the 32, 16 or 8 bit register at offset *reg* in the config space of the device selected by *tag*. If a master or target abort occurs, then the value 0xffffffff is returned, and the error is cleared.

```
void _pci_conf_write32 (pcitag_t tag, int reg, pcireg_t val)

void _pci_conf_write16 (pcitag_t tag, int reg, pcireg_t val)

void _pci_conf_write8 (pcitag_t tag, int reg, pcireg_t val)
```

Writes *val* to the 32, 16 or 8 bit register at offset *reg* in the config space of the device selected by *tag*.

```
pcireg_t _pci_statusread (void)
```

Returns the PCI host bridge's command/status register; this may be used to check for master or target aborts, and other error conditions.

```
void _pci_statuswrite (pcireg_t stat)
```

Writes *stat* to the PCI host bridge's command/status register; used to clear latched error signals.

```
int _pci_map_mem (pcitag_t tag, int reg, vm_offset_t *vap, vm_offset_t *pap)
```

Reads a PCI device's memory space base register (*reg* = 0x10 to 0x28 or 0x30) from the configuration space of the device selected by *tag*, and returns a CPU virtual address which will map to that PCI aperture in **vap*; the corresponding CPU physical address is returned in **pap*. Note that the physical PCI bus address stored in the device's base register may not correspond in a simple way to the CPU physical or virtual address. Returns 0 if all goes well, or -1if the operation fails.

```
int _pci_map_io (pcitag_t tag, int reg, vm_offset_t *vap, vm_offset_t *pap)
```

Like `_pci_map_mem`, but maps an I/O space base register.

```
int _pci_map_int (pcitag_t tag)
```

Returns the “interrupt number” for the device selected by *tag*. The value returned is zero if the device does not have an interrupt line, and negative if there is a problem finding the corresponding interrupt number.

```
vm_offset_t _pci_dmamap (vm_offset_t pa, unsigned int len)
```

Maps the CPU physical address of a region of DRAM bounded by *pa* and *pa+len* to a PCI address, which can be passed to a PCI bus master device for “DMA” purposes. Note that there may be no direct correspondence between CPU and PCI addresses.

```
vm_offset_t _pci_cpumap (vm_offset_t pcia, unsigned int len)
```

Performs the reverse of the `_pci_dmamap` transformation, and converts a PCI memory address to a CPU physical address.

```
void _pci_flush (void)
```

Ensures that any software-visible PCI host bridge read-ahead fifos are empty.

```
void _pci_wbflush (void)
```

Ensures that any software-visible PCI host bridge write buffers are flushed to PCI.

```
int _pci_cacheline_log2 (void)
```

Returns `log2()` of the PCI cacheline size which should be programmed into any device which needs to know that value.

```
int _pci_maxburst_log2 (void)
```

Returns `log2()` of the maximum PCI burst length supported by the PCI host bridge.

```
void * _isa_map_mem (vm_offset_t addr)
```

Some legacy PCI devices (e.g. VGA cards) start up with fixed mappings in a virtual ISA memory bus (the bottom 16MB of PCI memory space). This function returns a CPU virtual address pointer which maps to address *addr* within the ISA memory space.

```
void * _isa_map_io (unsigned int port)
```

Similar to `_isa_map_mam()` but for access to the virtual ISA I/O bus (the bottom 1MB of PCI I/O space); returns the CPU virtual address which maps to ISA I/O port *port*.

`vm_offset_t _isa_dmamap (vm_offset_t pa, unsigned int len)`

Like `_pci_dmamap` but for ISA DMA devices.

`vm_offset_t _isa_cpumap (vm_offset_t isaa, unsigned int len)`

Like `_pci_cpumap` but for ISA DMA devices.

Chapter 7

CPU Management

The second major component of the SDE run-time system consists of a set of support functions with which to initialize and maintain a MIPS architecture processor's caches, TLB and coprocessor registers; together with a powerful exception and interrupt handling mechanism, and support for remote source debugging of rommable code.

7.1 CPU Initialization

For rommable programs this code is invisible to your “application” program, as it is invoked automatically after a hardware reset, and before calling your `main()` function. It is described in detail in Section 8.4.1 “CPU Reset Handling”.

7.2 Exception and Interrupt Handling

SDE has sample code - MTK customers get full sources - showing how to handle exceptions and interrupts in the MIPS architecture. The code supplied is certainly usable in a simple system.

The monitor-specific code hooks SDE’s exception handling into the PROM monitor's own exception handling mechanism. This allows application programs to use the interface described here, whilst other exceptions (e.g. breakpoints) continue to be handled by the PROM monitor (e.g. the YAMON monitor).

7.2.1 C-level Exceptions

The run-time system provides a simple but powerful exception handling mechanism called *xcptions*, which are modelled on the POSIX *signal* handling mechanism described inSection 6.1.6 “Signal Handling”. To use it, include the header file <*mips/xcpt.h*> which defines these interfaces:

```
typedef int (*xcpt_t)(int, struct xcptcontext *);

struct xcptaction {
    xcpt_t xa_handler;
    unsignedxa_flags;/* unused */
};

/* install xcption handler */
int xcptaction (int xcptno, struct xcptaction *act,
          struct xcptaction *oact);

/* install xcption handler (simple version) */
xcpt_txcption (int xcptno, xcpt_t handler);
```

The `xcptaction()` function is similar to the POSIX `sigaction()` function. If `act` is non-zero, then it specifies a handler routine to be called when exception `xcptno` occurs (as defined in <*mips/xcpt.h*>). If `oact` is non-zero, then the previous handling information for that exception is returned to the caller. The function returns zero on success, or a non-zero error code if the parameters are faulty.

Once a handler is installed, it remains installed until another `xcptaction()` call is made for the same exception number. Note that the `xcptaction.xa_flags` field is currently ignored, but is intended to allow control over which registers are saved and how the exception is vectored; it should be set to zero.

The `xcption()` function provides a simpler interface, analogous to the old UNIX *signal* function. It is passed a simple function pointer, or **`XCPT_DFL`** to restore the default handler. It returns a pointer to the previous handler function, or **`XCPT_ERR`** on error.

When an exception occurs the appropriate *xcption* handler is called with two arguments:

1. the exception number;

2. a pointer to the *xcptcontext* structure which holds the processor state at the time of the exception, for example:

   ```
   int handler (int xcptno, struct xcptcontext *xcp)
   ```

   The *xcption* handler should normally return 0.

For an example showing the use of *xcptions*, see Section 3.1.2 “TLB Exception Handling (tlbxcpt)”.

7.2.1.1 Error Handling

As stated above, an *xcption* handler should normally return 0. But if it cannot handle the exception properly, or needs to asynchronously inform the application of some event, then it can return a non-zero POSIX *signal number*, as defined in Section 6.1.6 “Signal Handling”. The run-time system contains a default exception handler, which simply translates MIPS exception numbers into the appropriate POSIX signal numbers.

The application's signal handler, if installed by `sigaction()` or `signal`(), will be called before returning to the interrupted/failing instruction; if the signal handler then returns normally, execution will continue with the interrupted instruction. If no signal handler is installed, then the application will instead be terminated with a diagnostic message showing the cause of the exception, a register dump, and a stack trace. Note that SIGKILL cannot be caught, so it is guaranteed to terminate the application.

If your application has been built to run on an MDI target (e.g. the MIPSSim simulator or a CPU connected by an EJTAG probe), or it includes the SDE remote debug stub (see Section 8.4.3 “Remote Debug Stub”), then *gdb* will be activated whenever any exception handler returns a non-zero result, just before it is passed to the application’s signal handler. This lets you use *gdb* to analyze exceptional events. But when you are using a PROM monitor's remote debug facilities (e.g. YAMON), then only “uncaught” exceptions will be seen by *gdb*: if you've installed an SDE exception handler then that exception will not be reported to *gdb*, whatever its result, unless you set a breakpoint in the exception handler itself, or in the `xcpt_default` function.

```
/* diagnostics */
voidxcpt_show (struct xcptcontext *xcp);
voidxcptstacktrace (struct xcptcontext *xcp);
```

An exception handler may call `xcpt_show()` and/or `xcptstacktrace()` explicitly, to display diagnostic messages without terminating the application.

Note that all interrupts are disabled during exception processing, unless they are explicitly unmasked inside your *xcption* or *intrupt* handler.

7.2.2 RTOS Context Switch

RTOS developers and porters may find the following functions useful.

```
/* return to different xcption context */
void   xcptrestore (struct xcptcontext *xcp);

/* low-level setjmp/longjmp */
int    xcptsetjmp (xcptjmp_buf *xjb);
void   xcptlongjmp (xcptjmp_buf *xjb, int val);
```

The `xcptsetjmp()` and `xcptlongjmp()` functions are analogous to the standard C library `setjmp` and `longjmp` functions, but rather than saving and restoring the high-level POSIX signal mask, they save and restore the MIPS Coprocessor 0 *Status* Register (i.e. the interrupt mask), along with the stack pointer, program counter, and the other *callee-saved* registers. These functions can be used to implement a context save/restore for threads that have voluntarily blocked (e.g. due to a locked semaphore).

The `xcptrestore()` function allows an explicit return to a different xcption context, i.e., not the one that you are currently servicing. This can be used to implement a context switch to a thread that has been scheduled by an external event (i.e. an interrupt).

Since it is unlikely that multiple threads will be using the floating point unit simultaneously, we recommend that the floating point context switch should be lazy: enable the *Status.CU1* bit only for the current FPU owner, and then switch the FPU registers only upon receiving a *Coprocessor Unusable* (**`XCPTCPU`**) exception.

7.2.3 C-level Interrupts

On almost all MIPS processors there are 8 level-sensitive interrupt “inputs” (6 hardware and 2 software). If any become active, and they are enabled by the mask bits in the CPU's *Status* Register, then the processor generates an Interrupt (**`XCPTINTR`**) exception. Software must then examine the pending bits in the *Cause* Register to determine which of the 8 interrupts is active, prioritize them and then vector to the relevant interrupt handler.

We provide a mechanism called *intrupts* to handle this: it is very similar to the *xcption* mechanism described above, but with an additional interrupt prioritization scheme. Of course *intrupts* are just a special class of *xcption*, and is defined in header file <*mips/xcpt.h*>.

```
struct intraction {
   xcpt_t ia_handler;/* interrupt handler function */
   int    ia_arg;/* passed to interrupt handler */
   unsignedia_ipl;/* priority (1-8, 0=off) */
};

/* install intrupt handler */
int intraction (unsigned int intrno, struct intraction *act,
          struct intraction *old);

/* install intrupt handler (simple version) */
xcpt_tintrupt (unsigned int intrno, xcpt_t handler, int arg);
```

The `intraction()` function installs an *intrupt* handler, just like `xcptaction()` described above. The `intrno` argument is a number in the range 0 to 7, specifying which interrupt-pending bit in the *Cause* Register this action refers to. The `intraction.ia_arg` field specifies an arbitrary value to be passed to the *intrupt* handler, which might be used to allow a common handler to distinguish between two distinct devices.

The `intrupt()` function provides a simpler way to install an interrupt handler. It is like the `xcption()` function described above, but its `arg` parameter fulfills the same task as the `intraction.ia_arg` field.

When an interrupt occurs the appropriate intrupt handler is called with two arguments:

1. the `ia_arg` parameter;

2. a pointer to the *xcptcontext* structure which holds the processor state at the time of the interrupt, for example:

```
int handler (int arg, struct xcptcontext *xcp)
```

Like an `xcption` handler, an *intrupt* handler should normally return 0, but can return a *signal* number if it wants to send an asynchronous signal to the application. For instance a “debug button” interrupt handler could return SIGTRAP to enter the debugger.

Some boards may multiplex several interrupts onto each CPU interrupt line, and they will require a second level interrupt handler that uses an external interrupt request register to select the correct interrupt function.

Warning: Interrupt handlers should not expect to be able to safely change the *Status* Register saved in `xcp->sr` if the non-interrupt code itself modifies the *Status* Register non-atomically (e.g. using `mips_bissr()`, `spl()`, etc). Coprocessor register updates can never be atomic (though the MIPS32 Release 2 ISA does allow atomic changes of the CP0 *Status* Register, but only to the interrupt enable (IE) bit), and there is no simple way to serialize access to the *Status* Register. Contact us for advice if you need to do this.

For an example program showing the use of *intrupts*, see Section 3.1.14 “Interrupt Example”.

7.2.3.1 Interrupt Priorities

Remember that until very recently MIPS processors have not supported hardware interrupt prioritization, and it has traditionally been up to software to implement whatever priority scheme it requires. Our *intrupt* mechanism implements a fixed-priority software-based scheme, whereby each interrupt input can be assigned to one of 8 fixed *interrupt priority levels* (IPLs). This is not a one-to-one mapping: any number of interrupt inputs can be assigned the same IPL, and in any combination.

The `intraction.ia_ipl` field, passed to the `intraction()` function, explicitly specifies that interrupt's IPL. But the simpler `intrupt()` function uses a default IPL derived from the interrupt number as shown in Table 7.1.

Table 7.1 Interrupt Priorities

Input	Cause Register	IPL
h/w interrupt 5	IP7	8 <- HIGHEST
h/w interrupt 4	IP6	7
h/w interrupt 3	IP5	6
h/w interrupt 2	IP4	5
h/w interrupt 1	IP3	4
h/w interrupt 0	IP2	3
h/w interrupt 1	IP1	2
h/w interrupt 0	IP0	1
		0 <- LOWEST

In this model the CPU is notionally set to a priority level between 0 and 8 (inclusive): being set to a given priority level means that all interrupts at that IPL and below are masked out, and all above are enabled. Thus if the CPU is at priority level 0 it means that all interrupts are enabled, and if at level 8 then all are disabled. Normally your application will be running at level 0.

When an interrupt handler is called, the CPU priority is automatically set to that interrupt's IPL for the duration of the call to the handler. This prevents nested interrupts from the same device, or lower-priority devices, but allows them from higher priority devices.

Device drivers and other code will sometimes need to explicitly block out some or all interrupts in critical regions. This is done by temporarily “raising” and then “lowering” the CPU's priority level, using these functions:

```
unsigned int spl (unsigned int ipl);
unsigned int splx (unsigned int x);
```

Here `spl()` sets the CPU's priority level to `ipl`, and returns a value that can be passed later to `splx()`, to restore the old priority. Note that this return value is opaque: it is not the old priority level. This leads to the following typical usage:

```
{
    unsigned int s = spl (5);/* block out level 5 i/us and below */
    /* CRITICAL REGION HERE */
    (void) splx (s); /* return to previous priority level */
}
```

For very short critical sections only it may be faster to disable all interrupts:

```
{
    unsigned int s = _mips_intdisable ();
    /* CRITICAL REGION HERE */
    _mips_intrestore (s);
}
```

You can test for a pending interrupt while it is blocked, using

```
int intrpending (unsigned int intrno);
```

which returns 1 if the CPU h/w interrupt pending bit `intrno` is active.

7.2.3.2 Software interrupts

The MIPS *Cause* Register includes two *software interrupt* bits, which allow high-priority interrupt handlers to request a new interrupt at a low-priority, or non-interrupt code to kick-start interrupt-level processing. The following functions provide a safe way to switch these interrupts on and off:

```
void   siron (unsigned int intrno);
void   siroff (unsigned int intrno);
```

Note that `intrno` may only be 0 or 1, and the respective interrupt handlers must call `siroff()` to remove the interrupt request before they return.

7.3 Cache Maintenance

The cache management function prototypes are supplied by including <*mips/cpu.h*>. Many of these routines expect to be passed an address range to operate on, consisting of a starting *virtual address*, and a byte count.

```
void mips_size_cache (void)
```

Size the caches, setting the following global variables:

- *mips_icache_size*, *mips_icache_linesize*, *mips_icache_ways*: The size (in bytes) of the primary instruction cache; the size of each cache line, and the number of ways of set associativity.
- *mips_dcache_size*, *mips_dcache_linesize*, *mips_dcache_ways*: Ditto for the primary data cache.

```
void mips_init_cache (void)
```

Size the caches as above, and initialize them. The function MUST be called after a hardware reset and before using the caches, otherwise they may be in an inconsistent state. This is normally called by the standard reset code. Do NOT call it from application code, as it may invalidate dirty cache lines in a writeback cache, without actually writing them back to memory.

```
void mips_sync_icache (vaddr_t va, size_t n)
```

Synchronizes the I-cache with the D-cache, which is necessary when the instruction stream is modified by software (e.g. inserting software breakpoints, self-modifying code, etc).

```
void mips_clean_cache (vaddr_t va, size_t n)
```

Write back and invalidate entries matching the given address range from all caches. The most common routine to call in device drivers before starting a DMA transfer, or after dynamically modifying executable code.

```
void mips_clean_dcache (vaddr_t va, size_t n)
```

Write back and invalidate entries matching the given address range from the data caches only - separate instruction caches are unchanged.

```
void mips_clean_icache (vaddr_t va, size_t n)
```

Invalidate entries matching the given address range from the instruction caches only - separate data caches are unchanged.

```
void mips_flush_cache (void)
```

Write back and invalidate all entries from all caches. The simplest way to completely synchronize caches and memory, but not necessarily the most efficient.

```
void mips_flush_dcache (void)
```

Write back and invalidate all entries from all data caches - separate instruction caches are unchanged.

```
void mips_flush_icache (void)
```

Invalidate all entries from all instruction caches - separate data caches are unchanged.

```
void mips_lock_icache (vaddr_t va, size_t n)

void mips_lock_dcache (vaddr_t va, size_t n)

void mips_lock_scache (vaddr_t va, size_t n)
```

On CPUs which support cache locking, these functions allow you to lock regions of code or data into the primary instruction, data or secondary caches respectively. Take care not to use the global *flush* functions after locking caches, as they will invalidate (and unlock) the locked cache lines.

7.4 TLB Maintenance

The functions listed below provide for initialization and maintenance of the CPU's memory management Translation Lookaside Buffer (TLB), if present. An example showing the use of these functions can be found in Section 3.1.2 "TLB Exception Handling (tlbxcpt)". The TLB and memory management definitions are supplied by including <*mips/cpu.h*>.

```
void mips_init_tlb (void)
```

Initializes and invalidates the whole TLB.

```
unsigned int mips_tlb_size (void)
```

Returns the number of entries in the TLB.

```
void mips_tlbinval (tlbhi_t hi)
```

Probes the TLB for an entry matching `hi`, and if present invalidates it.

```
void mips_tlbinvalall (void)
```

Invalidate the entire TLB.

```
void mips_tlbri2 (tlbhi_t *phi, tlblo_t *plo0, tlblo_t *plo1, unsigned *pmsk,
int index)
```

Reads the TLB entry with specified by `index`, and returns the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* parts in `*phi`, `*plo0`, `*plo1` and `*pmsk` respectively.

```
void mips_tlbwi2 (tlbhi_t hi, tlblo_t lo0, tlblo_t lo1, unsigned msk, int
index)
```

Writes `hi`, `lo0`, `lo1` and `msk` into the TLB entry specified by `index`.

```
void mips_tlbwr2 (tlbhi_t hi, tlblo_t lo0, tlblo_t lo1, unsigned msk)
```

Writes `hi`, `lo0`, `lo1` and `msk` into the TLB entry specified by the *Random* Register.

```
int  mips_tlbprobe2 (tlbhi_t hi, tlblo_t *plo0, tlblo_t *plo1, unsigned *pmsk)
```

Probes the TLB for an entry matching `hi` and returns its index, or -1 if not found. If found, then the *EntryLo0*, *EntryLo1* and *PageMask* parts of the entry are also returned in `*plo0`, `*plo1` and `*pmsk` respectively.

```
int  mips_tlbrwr2 (tlbhi_t hi, tlblo_t lo0, tlblo_t lo1, unsigned msk)
```

Probes the TLB for an entry matching `hi` and if present rewrites that entry, otherwise updates a random entry. A safe way to update the TLB.

7.5 Hardware Watchpoints

Some MIPS architecture CPUs provide one or more hardware watchpoint registers in Coprocessor 0 (these are separate from any EJTAG hardware breakpoint registers). The watchpoint registers generate a CPU exception when software loads or stores data, or executes instructions, within a programmable address range. Different MIPS-Based CPUs implement very different watchpoint controls (number of watchpoints, type of access, physical/virtual address, address masking, and so on). To make this manageable and portable between different CPUS we have developed a generic API which is documented here. These facilities are used by the SDE remote debug stub to support *gdb*'s watchpoint facility; but you could also use them to implement profiling or debugging facilities within your own software.

To use the watchpoint API described here, include the file <*mips/watchpoint.h*>.

```
int _mips_watchpoint_init (void)
```

Initializes the watchpoint system and returns the number of hardware watchpoints available.

```
int _mips_watchpoint_howmany (void)
```

Just returns the number of hardware watchpoints, without re-initializing the sub-system.

```
int _mips_watchpoint_capabilities (int wpnum)
```

Returns the *capability* of watchpoint number *wpnum* (0 to n). Usually called after `_mips_watchpoint_init()` to collect and cache each watchpoint's capability. The capability is the bitwise OR of some or all of the values shown in Table 7.2.

Table 7.2 Hardware Watchpoint Attributes

Watchpoint	Attribute
`MIPS_WATCHPOINT_SSTEP`	Hardware single-step supported.
`MIPS_WATCHPOINT_VALUE`	Can qualify the watchpoint with the value of the data being read or written from/to memory.
`MIPS_WATCHPOINT_ASID`	Can qualify match using the virtual address-space ID (ASID).
`MIPS_WATCHPOINT_VADDR`	Matches against virtual address (if not set then matches against physical address).
`MIPS_WATCHPOINT_RANGE`	Supports an address range (arbitrarily aligned start and end address).
`MIPS_WATCHPOINT_MASK`	Supports an address mask (size must be a power-of-two, and start address aligned on a matching boundary).

Table 7.2 Hardware Watchpoint Attributes (Continued)

Watchpoint	Attribute
MIPS_WATCHPOINT_DWORD	Only supports an address match within a single 8 byte aligned double word; if an address range/mask is supported then the minimum size and alignment is 8 bytes.
MIPS_WATCHPOINT_WORD	Only supports an address match within a single 4 byte aligned word; if an address range/mask is supported then the minimum size and alignment is 4 bytes.
MIPS_WATCHPOINT_X	Instruction fetch breakpoint supported.
MIPS_WATCHPOINT_R	Data read breakpoint supported.
MIPS_WATCHPOINT_W	Data write breakpoint supported.

```
int _mips_watchpoint_set (int type, int asid, vaddr_t va, paddr_t pa, size_t
size)
```

Creates a new watchpoint where: *type* is the OR of the last three capabilities (i.e. instruction fetch, read and/or write); *asid* is the virtual address space ID (or -1 for global); *va* is the virtual address of the start of the watchpoint region; *pa* is the physical address (can be zero if virtual address matching is supported); and *size* is the size of the watchpoint region.

For CPUs which support an address mask, *addr* and *size* can be arbitrarily aligned, and the code will compute the smallest aligned region which fits around them. Beware that this could get quite loose, and cause a large number of false watchpoint hits.

The return values, shown in Table 7.3, indicate the success or failure.

Table 7.3 Watchpoint Return Codes

Watchpoint	Return Code
MIPS_WP_OK	Succeeded.
MIPS_WP_NOTSUP	This type of watchpoint is not supported, or possibly you've asked for a watchpoint region which is larger than can be supported.
MIPS_WP_INUSE	All hardware resources which support this type of watchpoint are in use.
MIPS_WP_NOMATCH	Matching watchpoint cannot be found (see `_mips_watchpoint_clear()` below).
MIPS_WP_OVERLAP	Address range would overlap the debugger's own code, data or stack.
MIPS_WP_BADADDR	If the *pa* value is zero and virtual address matching is not supported.

```
int _mips_watchpoint_clear (int type, int asid, vaddr_t va, size_t size)
```

Delete a watchpoint: the parameters must match those used when the watchpoint was created by `_mips_watchpoint_set()`. See `_mips_watchpoint_set()` for the return codes.

```
int _mips_watchpoint_set_callback (int asid, vaddr_t va, size_t len)
```

A callback function which you can (optionally) provide. When a new watchpoint is about to be added, your code has a last chance to check the computed address range to make sure that it doesn't overlap with its own code or data (which could cause recursive watchpoint traps). Should return MIPS_WP_OK or MIPS_WP_OVERLAP. If you don't provide this function then all watchpoints are allowed.

```
int _mips_watchpoint_hit (const struct xcptcontext *xcp, vaddr_t *vap, size_t
*sizep)
```

Called by your hardware watchpoint exception handler (usually the debug stub) to check whether the exception context *xcp* was a true watchpoint hit. If so the return value will be non-zero, and contain one of MIPS_WATCHPOINT_R, MIPS_WATCHPOINT_W or MIPS_WATCHPOINT_X to indicate the type of access. If in addition the bit MIPS_WATCHPOINT_INEXACT is set then this was a watchpoint exception, but it was based on a loose address mask, and this access was outside of the range originally requested by `_mips_watchpoint_set()`; your code must single-step over this instruction and then continue.

```
void _mips_watchpoint_remove (void)
```

Called by the debug stub, or your watchpoint exception handler, to disable hardware watchpoints, e.g. before single-stepping over an instruction which may trigger the watchpoint.

```
void _mips_watchpoint_insert (void)
```

Called by the debug stub, or watchpoint exception handler, to enable hardware watchpoints, e.g. after single-stepping over an instruction and before continuing execution.

```
void _mips_watchpoint_reset (void)
```

Clear all watchpoints.

7.6 System Coprocessor (CP0) Intrinsics

All MIPS-Based CPUs contain a “System Control” subsystem known as Coprocessor 0, or CP0. This is used by operating systems and other low-level software to control interrupts, exceptions, memory management, caches, etc. These intrinsics provide very low-level access to the CP0 registers from C and C++ code. Other intrinsics which give access to “user-level” instructions and registers are described in a separate chapter, see Chapter 5, “MIPS® Architecture Intrinsics” on page 29.

The header file <*mips/cpu.h*> (which in turn includes the appropriate cpu-specific header), defines the intrinsics shown in Table 7.4 and described in the following subsections. The “*” symbol represents up to five separate intrinsics.

Table 7.4 Register Access Intrinsics

*	Arguments	Operation
`get`	`()`	Return the register value.
`set`	`(unsigned val)`	Sets the register to `val`, and returns void.
`xch`	`(unsigned val)`	Sets the register to `val`, and returns the **old** register value.
`bis`	`(unsigned set)`	Bit set (*`reg`* ||= `set`): returns the **old** register value. Only defined for registers with bit-fields.
`bic`	`(unsigned clr)`	Bit clear (*`reg`* `&= ~clr`): returns the **old** register value. Only defined for registers with bit-fields.

Table 7.4 Register Access Intrinsics (Continued)

*	Arguments	Operation
bcs	(unsigned clr, unsigned set)	Bit clear and set (reg = (*reg* & ~\clr) | set): returns the **old** register value. Only defined for registers with bit-fields.

7.6.1 Common CP0 Registers

Some of the CP0 registers are common between almost all MIPS-Based CPU families, and the intrinsics to access these have the common prefix `mips_`.

Remember though that even for the common registers, the internal bit definitions are not necessarily the same across all CPU types. Make sure that you include the generic <*mips/cpu.h*>, and not <*mips/m32c0.h*>, or any of the CPU-specific header files.

N.B. The intrinsics which manipulate the Coprocessor registers do not provide atomicity in the presence of interrupts or other exceptions. This can be particularly important if you are changing the *Cause* or *Status* registers. If possible, avoid read-modify-write operations on the *Status* Register: write only constant values, or stored values manipulated only by atomic operations, unless you know that interrupts are already disabled (e.g. because you're in an exception handler). Ensure that interrupts are disabled when you update the *Cause* Register.

`mips_*sr`

(i.e. *mips_getsr*, *mips_setsr*, *mips_xchsr*, *mips_bissr*, *mips_bicsr*). Operations on the *Status* Register (CP0 register 12). See the atomicity warning above.

`mips_*cr`

Operations on the *Cause* Register (CP0 register 13). See warning above.

`mips_getcount, mips_setcount`

`mips_getcompare, mips_setcompare`

Operations on the *Count* and *Compare* Registers (CP0 registers 9 and 11). Available on most modern MIPS architecture CPUs, these implement an on-chip timer.

`mips_getprid`

Return the read-only *PrID* Register (CP0 register 15). See <*mips/prid.h*> for a list of known values.

`mips_*config`

Operations on *Config* Register (CP0 register number varies).

`mips_*ecc`

Operations on *ECC* Register (CP0 register 26), used for cache error correction on some MIPS III + CPUs.

`mips_*context`

Operations on the *Context* Register (CP0 register 4).

`mips_*pagemask`

Operations on the *PageMask* Register (CP0 register 5).

`mips_*wired`

Operations on the *Wired* Register (CP0 register 6).

`mips_*entrylo`

Operations on the *EntryLo* Register (CP0 register 2).

`mips_*entryhi`

Operations on the *EntryHi* Register (CP0 register 10).

`mips_*taglo`

`mips_*taghi`

Operations on *TagLo* and *TagHi* registers (CP0 registers 28 and 29), used for cache testing and maintenance on many MIPS architecture CPUs.

`mips_*watchlo`

`mips_*watchhi`

Operations on *WatchLo* and *WatchHi* registers (CP0 registers 18 and 19), used for hardware watchpoints on many MIPS III + CPUs.

7.6.2 CP0 Registers of MIPS32®/MIPS64® Architecture

The include files <*mips/m32c0.h*> and <*mips/m32tlb.h*> define the Coprocessor registers and memory-management unit of CPUs conforming to the MIPS32/MIPS64 specifications. They include the following functions:

`mips32_*config0`

Operations on the *Config0* Register (CP0 register 16, select 0), also available via the generic `mips_*config` functions described above.

`mips32_getconfig1`

Returns the *Config1* Register (CP0 register 16, select 1).

`mips32_getconfig2`

Returns the *Config2* Register (CP0 register 16, select 2).

`mips32_getconfig3`

Returns the *Config3* Register (CP0 register 16, select 3).

`mips32_getwatchlo(int sel)`

Return the *WatchLo* Register numbered `sel`.

`mips32_setwatchlo(int sel, unsigned int val)`

Set the *WatchLo* Register numbered `sel` to `val`.

`mips32_getwatchhi(int sel)`

Return the *WatchHi* Register numbered `sel`.

`mips32_setwatchhi(int sel, unsigned int val)`

Set the *WatchHi* Register numbered `sel` to `val`.

`mips32_*errctl`

Operations on the *ErrCtl* Register (CP0 register 26, select 0).

`mips32_*datalo`

Operations on the *DataLo* Register (CP0 register 28, select 1).

7.6.3 CP0 Registers of MIPS32®/MIPS64® Release 2 Architecture

The MIPS32 Release 2 ISA defines a few new Coprocessor 0 registers, also defined in include files <*mips/m32c0.h*>.

`mips32_*pagegrain`

Operations on the MIPS32 Release 2 *PageGrain* Register (CP0 register 5, select 1).

`mips32_*hwrena`

Operations on the MIPS32 Release 2 *HWREna* Register (CP0 register 7, select 0).

`mips32_*intctl`

Operations on the MIPS32 Release 2 *IntCtl* Register (CP0 register 12, select 1).

`mips32_*srsctl`

Operations on the MIPS32 Release 2 *SRSCtl* Register (CP0 register 12, select 2).

`mips32_*srsmap`

Operations on the MIPS32 Release 2 *SRSMap* Register (CP0 register 12, select 3).

`mips32_*ebase`

Operations on the MIPS32 Release 2 *EBase* Register (CP0 register 15, select 1).

7.6.4 Shadow Sets of MIPS32®/MIPS64® Release 2 Architecture

The MIPS32 Release 2 architecture adds support for alternative “shadow” banks of CPU general purpose registers, for use by low-latency interrupt and exception handlers. These intrinsics allow C code to read and write registers in other shadow sets, and are defined in include files <*mips/m32c0.h*>.

`uint32_t _mips32r2_xchsrspss(uint32_t set)`

Sets the *PSS* field in the *SRSCtl* Register to `set`, allowing access to that shadow set with the following intrinsics. Returns the old value of the *PSS* field.

`uint32_t _mips32r2_rdpgpr(int regno)`

Returns register number `regno` from the selected shadow set. The `regno` argument must be a constant between 0 and 31.

`void _mips32r2_wrpgpr(int regno, uint32_t val)`

Sets register number `regno` in the selected shadow set to `val`. The `regno` argument must be a constant between 0 and 31.

7.6.5 CP0 Registers of MIPS® MT ASE

The include file <*mips/mt.h*> defines the Coprocessor registers introduced by the MT ASE, and includes the following C access functions:

`mips32_*mvpcontrol`

Operations on the *MVPControl* Register (CP0 Register 0, Select 1).

`mips32_*mvpconf0`

Operations on the *MVPConf0* Register (CP0 Register 0, Select 2).

`mips32_*mvpconf1`

Operations on the *MVPConf1* Register (CP0 Register 0, Select 3).

`mips32_*vpecontrol`

Operations on the *VPEControl* Register (CP0 Register 1, Select 1).

`mips32_*vpeconf0`

Operations on the *VPEConf0* Register (CP0 Register 1, Select 2).

`mips32_*vpeconf1`

Operations on the *VPEConf1* Register (CP0 Register 1, Select 3).

`mips32_*yqmask`

Operations on the *YQMask* Register (CP0 Register 1, Select 4).

`mips32_*vpeschedule`

Operations on the *VPESchedule* Register (CP0 Register 1, Select 5).

`mips32_*vpeschefback`

Operations on the *VPEScheFback* Register (CP0 Register 1, Select 7).

`mips32_*tcstatus`

Operations on the *TCStatus* Register (CP0 Register 4, Select 1).

`mips32_*tcpc`

Operations on the *TCPC* Register (CP0 Register 4, Select 2).

`mips32_*tchalt`

Operations on the *TCHalt* Register (CP0 Register 4, Select 3).

`mips32_*tccontext`

Operations on the *TCContext* Register (CP0 Register 4, Select 4).

`mips32_*tcschedule`

Operations on the *TCSchedule* Register (CP0 Register 4, Select 5).

`mips32_*tcschefback`

Operations on the *TCScheFback* Register (CP0 Register 4, Select 6).

`mips32_*srsconf*`

Operations on the *SRSConf0-4* Registers (CP0 Register 6, Select 1-5)

The MT ASE also permits access to registers with a different thread context or virtual processor.

`mips32_mt_settarget (int vpe, int tc)`

Selects the target VPE and TC number for the following access functions.

`mips32_mt_getc0status()`

Return the CP0 *Status* Register of the selected TC/VPE.

`mips32_mt_setc0status(int val)`

Set the CP0 *Status* Register of the selected TC/VPE.

`mips32_mt_getc0cause()`

Return the CP0 *Cause* Register of the selected TC/VPE.

`mips32_mt_setc0cause(val)`

Set the CP0 *Cause* Register of the selected TC/VPE.

`mips32_mt_getc0config()`

Return the CP0 *Config* Register of the selected TC/VPE.

`mips32_mt_setc0config(val)`

Set the CP0 *Config* Register of the selected TC/VPE.

`mips32_mt_getc0config1()`

Return the CP0 *Config1* Register of the selected TC/VPE.

`mips32_mt_setc0config1(val)`

Set the CP0 *Config1* Register of the selected TC/VPE.

`mips32_mt_getc0ebase()`

Return the CP0 *EBase* Register of the selected TC/VPE.

`mips32_mt_setc0ebase(val)`

Set the CP0 *EBase* Register of the selected TC/VPE.

`mips32_mt_getsp()`

Return the stack pointer (`$29`) of the selected TC/VPE.

`mips32_mt_setsp(val)`

Set the stack pointer (`$29`) of the selected TC/VPE.

`mips32_mt_getgp()`

Return the global pointer (`$28`) of the selected TC/VPE.

`mips32_mt_setgp(val)`

Set the global pointer (`$28`) of the selected TC/VPE.

`mips32_mt_getvpecontrol()`

Return the CP0 *VPEControl* Register of the selected TC/VPE.

`mips32_mt_setvpecontrol(val)`

Set the CP0 *VPEControl* Register of the selected TC/VPE.

`mips32_mt_getvpeconf0()`

Return the CP0 *VPEConf0* Register of the selected TC/VPE.

`mips32_mt_setvpeconf0(val)`

Set the CP0 *VPEConf0* Register of the selected TC/VPE.

`mips32_mt_gettcstatus()`

Return the CP0 *TCStatus* Register of the selected TC/VPE.

`mips32_mt_settcstatus(val)`

Set the CP0 *TCStatus* Register of the selected TC/VPE.

`mips32_mt_gettcbind()`

Return the CP0 *TCBind* Register of the selected TC/VPE.

`mips32_mt_settcbind(val)`

Set the CP0 *TCBind* Register of the selected TC/VPE.

`mips32_mt_gettcrestart()`

Return the CP0 *TCRestart* Register of the selected TC/VPE.

`mips32_mt_settcrestart(val)`

Set the CP0 *TCRestart* Register of the selected TC/VPE.

`mips32_mt_settchalt(val)`

Set the CP0 TCHalt Register of the selected TC/VPE.

`mips32_mt_gettccontext()`

Return the CP0 *TCContext* Register of the selected TC/VPE.

`mips32_mt_settccontext(val)`

Set the CP0 *TCContext* Register of the selected TC/VPE.

7.7 Miscellaneous System Support

The following generic MIPS system support functions are defined in include file <*mips/cpu.h*>.

`void mips_wbflush (void)`

Drain the write buffer. All stores issued prior to the call are guaranteed to have been written to memory or device by the time the function returns. It should be called between writing to device control registers and reading their

status/data registers. On some CPUs it is also necessary to call it between successive writes to the same register, to prevent word-gathering write-buffers from swallowing some of the writes.

```
void _mips_sync (void)
```

On modern MIPS-Based CPUs this generates a `sync` instruction. This is almost but not quite the same as `mips_wbflush()` - it is a memory *barrier* which guarantees that all memory accesses preceding this instruction will be completed before any accesses which follow this instruction. It says nothing though about external state, such as interrupts - and on simpler CPUs with blocking loads it may be interpreted as a no-op.

```
uint8_t  mips_get_byte (void *addr, int *err)

uint16_t mips_get_half (void *addr, int *err)

uint32_t mips_get_word (void *addr, int *err)

uint64_t mips_get_dword (void *addr, int *err)
```

Return the byte, halfword, word, or dword at address `addr`. If the address is invalid, then `*err` may be set to a non-zero value; otherwise `*err` is unchanged. You can use these functions when accessing arbitrary memory locations outside of your program, to ensure that peculiarities of your system or CPU address map are handled correctly.

```
int mips_put_byte (void *addr, uint8_t val)

int mips_put_half (void *addr, uint16_t val)

int mips_put_word (void *addr, uint32_t val)

int mips_put_dword (void *addr, uint64_t val)
```

Store a byte, halfword, word, or dword `val` to arbitrary address `addr`. If the address is invalid, then a non-zero value may be returned, otherwise they return zero.

7.8 Floating Point Coprocessor (CP1)

The generic header file <*mips/fpa.h*> defines constants and functions for controlling the floating point coprocessor (CP1) and its register set.

```
int      fpa_enable (int fast)
```

Probes to see if CP1 is present. If so it is initialized, CP1 instructions are enabled, and 1 is returned. If it is not present, then CP1 instructions are disabled, and 0 is returned. If `fast` is non-zero then, if possible, the FPU is set to “performance mode” where IEEE-754 traps will not be taken for denormalized values, which will instead be flushed or rounded.

```
void     fpa_save (struct fpactx *ctx)
```

Save all the floating point data registers and coprocessor state into the structure pointed to by `ctx`.

```
void     fpa_restore (const struct fpactx *ctx)
```

Restore all the registers and coprocessor state from the structure pointed to by `ctx`.

```
unsigned fpa_getrid (void)
```

Returns on CP1 control register 0, the read-only floating point *RevisionID* Register.

```
fpa_*sr
```

Operations on CP1 control register 31, the floating point control and status register. See Section 7.6 "System Coprocessor (CP0) Intrinsics" for a description of '*'.

7.8.1 Coprocessor 1 Emulation

The run-time system includes a complete MIPS coprocessor\ 1 (floating point) instruction emulator. It can emulate all floating point instructions when there is no hardware FPU, or just those instructions with operands that the FPU cannot handle (e.g. denormalized values, underflow, etc). The only public interface to the module is:

```
void_cop1_init (int emulateall);
```

This function installs the appropriate exception or interrupt handler: a non-zero value for `emulateall` installs full emulation via the CoProcessor Unusable (XCPTCPU) exception, whilst a zero value installs only the floating point interrupt handler (or XCPTFPE exception handler on an R4000 CPU and above). You'll probably never need to call it yourself - it is normally invoked automatically by the standard run-time startup code, see Section 8.1.1 "Run-time Initialization".

A faster alternative to trap-based coprocessor emulation is to use the compiler's **`-msoft-float`** option,

Chapter 8

Embedded System Kit Source

This chapter introduces the source files which make up the embedded system kit. The directory `...sde/kit` contains a collection of source, assembler source and pre-compiled object files which fulfill two separate functions:

1. They form a run-time I/O system and environment for application programs, such as the examples. The programming interface provided by this system is described in Section 6.1 "POSIX API Environment".

2. They include a set of low-level primitives to initialize and manage a MIPS-Based CPU's caches, TLB, FPU, exceptions, interrupts, etc.

The programming interface provided by these components is described in Chapter 7, "CPU Management" on page 61.

The kit is set up so that you can build software, modelled on one of SDE's example programs, and by some judicious values for makefile variables, get the software to build successfully for any of a large number of different boards.

Unless you are using SDE *lite* then this is all supplied as source code, and can be adapted to other run-time environments, or perhaps just used for inspiration when porting a PROM monitor or operating system to the MIPS architecture.

The kit is built around the idea that each target has its own directory of software, and its own makefile; in the target makefile the ROM monitor (if any) and CPU type are identified, along with other options.

But first a note on the run-time I/O system.

8.1 POSIX System Interface

The run-time I/O system is modelled on the POSIX.1 specification (see [POSIX88]). It is implemented by the following files in `...sde/kit/share`; they will be either or assembler-with-cpp (`.sx`) files for supported SDE customers, or pre-compiled object files for other users:

- *crt0*: generic / C++ run-time system startup code, see below.

- *env*: the getenv / setenv functions, which interface to a board-specific non-volatile environment variable store if present.

- *flashenv.c* and *flashrom.c*: support code for a simple NAME=VALUE environment store in FLASH.

- *flashdev*: implements the `/dev/flash` special device file, as described in Section 6.1.4 "Flash Memory Devices (/dev/flash)".

- *mfs*: implements a pseudo ``memory file system'' whose structure is defined by a monitor-specific file (e.g. `pmon/pmonroot.c`).

- *nvenv*: support code for a simple environment store in non-volatile RAM.
- *posix*: implements the generic POSIX ``file I/O'' interface functions, such as open, close, read, write, ioctl, stat, etc. They pass control to device-specific functions defined by the device files in the ``memory file system'' above.
- *paneldev*: implements the `/dev/panel` special device file, described in Section 6.1.5 “Alpha Display (/dev/panel)”.
- *profil*: contains the profiling support functions which arrange to sample the program-counter at 100Hz.
- *sbrk*: is the rudimentary memory allocator required by `malloc()` et al. It dishes out consecutive, contiguous areas of memory between `_end` (the end of the program's data), and 64Kb below the stack. This hard-wired 64Kb stack size may be too small for some applications, and there is no check for the stack and memory pool colliding. You may need to change this limit!
- *signals*: is an emulation of the POSIX *signal* mechanism, which integrates with SDE’s low-level exception handling.
- *timer*: is a generic interface to whatever timing hardware a board provides. It implements three high-precision interval-timers, modelled on the BSD / SVr4 `setitimer()` interface. It also maintains the current ``elapsed'' time for use by `time()` and `clock()`. One of the interval-timers is also used by the pc-sampling profiler.
- *tty*: handles I/O to ``tty'' devices (i.e. the console), including simple line editing, baud rate setting, etc. It implements a large subset of the POSIX termios interface.

8.1.1 Run-time Initialization

The startup code in `.../sde/kit/share/crt0.sx` sets up the initial run-time environment required by C and C++ programs. Its entry-point is `__start`, which is arrived at either by a jump from the end of the standalone `romlow` code, by an eval board's PROM monitor after your code has ben downloaded to RAM, or by *gdb* via an EJTAG probe or simulator. It performs the following steps:

- Initializes the *gp* register, required for *gp-relative* addressing.
- Moves the *sp* register to the same address space (i.e. cached KSEG0 or uncached KSEG1) as the program's data has been linked for.
- Zeroes the “uninitialized” data section (*bss*).
- Initializes the POSIX I/O system and drivers, described above.
- Initializes the remote debug stub, if the **RDBG** symbol is non-zero. This may cause an immediate breakpoint if **RDBG** is greater than 1 (which is what happens if **`RDEBUG=immed`** is used in the example makefiles).
- Initializes the floating point coprocessor and/or CP1 emulator, as selected by the **#float** assertion (which is controlled by the **`FLOAT`** variable in the example makefiles).
- Starts the profiling timer if **CFLAGS** contains the **-p** flag.
- Runs the C++ global constructors, if any. It uses `atexit()` to arrange for the C++ global destructors to be called when the program exits.

- Calls `main()`.
- If `main()` returns, then it calls `exit()` with the returned value as its argument.

8.1.2 Run-time Termination

The `crt0.sx` file also contains the low-level `_exit()` function, which performs the following step:

- Calls the even lower-level `__exit()` function, defined in the monitor-specific directory. This will normally return control to the PROM monitor or *gdb*, or in a rommable program might switch off the board, or enter a tight loop.

8.2 Target-specific Code

Each target evaluation board or simulator has its own subdirectory under `.../sde/kit`. The list of supported targets is in Chapter 2, “Target-specific Libraries” on page 9.

Each target's directory contains a configuration file sbd.mk which describes the key features of the target, such as the CPU type, whether it has an FPU, the monitor type, the default download, ROM and RAM addresses, etc, etc. It also lists the files within that directory which handle board reset/initialization and devices (e.g. UART, timer, etc).

If you only want to run programs under control of an eval board's PROM monitor, then the board initialization code and UART driver can be committed, since these functions are provided to your application by the monitor. If you do need to retarget SDE to a new board, then see Chapter 9, “Retargeting the Toolkit” on page 89 for more details.

8.2.1 PCI Bus Configuration

The directory `.../sde/kit/pci/` contains generic PCI bus configuration, enumeration and access routines, which are included into the run-time system if `sbd.mk` defines **`PCI=yes`**. The functions in this directory then make use of board-specific functions to access the PCI bus controller chip; see `.../sde/kit/p6032/pci_machdep.c` for an example.

8.3 Monitor-specific Glue

Wherever possible the run-time system uses the low-level I/O facilities provided by a board's PROM monitor. It does this to:

1. Make it easier to retarget SDE to a new board which has a supported monitor.
2. Integrate more closely with the debugging facilities of the PROM monitor, so that you can use its interactive and/or remote debug facilities.
3. Make use of any remote console and file I/O facilities which it offers, while maintaining the standard POSIX and ISO / ANSI “stdio” interfaces.

Like the board-support code, each supported monitor has its own sub-directory containing a configuration file `monitor.mk`, together with the monitor interface code. The directories are shown in Table 8.1.

Table 8.1 Supported PROM Monitors

Directory	Description
`bare`	A “bare-board” interface for rommable programs, or for boards without one of the supported monitors. In this case, software from SDE takes over the board devices and exceptions completely.
`yamon`	Interface to the YAMON monitor used on MIPS Technologies' development boards.
`mdimon`	Provides facilities (including virtual console and host file I/O) for programs running on targets connected to gdb via the “MDI” interface.
`mtspmon`	Provides facilities for SDE programs running on the Real-time Processor side of a multi-threaded CPU, communicating with a Linux device driver on the Application Processor.
`gnusim`	Provides host file I/O for programs running on the GNU simulator included with SDE.
`idtsim`	Interface to the IDT/sim monitor used on boards supplied by IDT Inc.
`pmon`	Interface to the public-domain LSI PMON monitor, used on boards supplied by LSI Logic Inc. and other vendors.

More details of the files in these directories, and how to add support for a new PROM monitor, can be found inChapter 9, “Retargeting the Toolkit” on page 89.

8.4 Low-level CPU Management

The following files provide the low-level CPU initialization and control functions. In the supported, paid-for SDE version you'll find their source code in `.../sde/kit/share`; other users will find that the object code is supplied ready-built in the `.../sde/kit/free` directory, in a library file called `SBD.lib`.

- *cache.sx cache_ops.sx*: Interface layer to cache management functions, which can select at run-time between different cache architectures.
- *cp1emu.c*: A coprocessor 1 (floating point) instruction emulator, used when the coprocessor hardware is absent, or to handle those instructions which the coprocessor cannot (denormalized numbers, underflow, etc).
- *bremu.c*: is also required; it emulates branch instructions, which is a necessary part of emulating an FP instruction if they happen to be in a branch delay slot.
- *cw01cache.sx cw01cache_ops.sx*: Vendor-specific cache handling for the LSI W400x/TR411x PUs.
- *cw10cache.sx cw10cache_ops.sx*: Vendor-specific cache handling for the LSI W401x PUs.
- *cw10tlb.sx cw10tlb_ops.sx*: TLB initialization and management functions for the optional LSI W401x memory management unit.

- *dbg.c*: The remote debug stub, used when debugging standalone, rommable programs, or when a board's PROM monitor does not implement the “MIPS remote” debugging protocol. See Section 8.4.3 “Remote Debug Stub” below.
- *dbgsig.c*: Dummy h/w interrupt initialization for remote debug stub; this can be overridden.
- *dbgsup.c*: Default I/O support routines for remote debug stub.
- *ecchandler.c*: Example cache/ecc error handler for R4000 S /M processors.
- *fcache.c*: Generic Flash ROM interface for the remote debug stub, allowing breakpoints to be set in Flash.
- *intrupt.c*: Generic, prioritisable interrupt dispatcher.
- *lr30cache.sx lr30cache_ops.sx*: Cache initialization and management for the LSI LR330x0 families.
- *m32cache.sx m32cache_ops.sx*: Cache support for the MIPS32 and MIPS64 architectures.
- *m32c1.sx*: Coprocessor 0 support for the MIPS32 and MIPS64 architectures.
- *m32tlb.sx m32tlb_ops.sx*: TLB initialization and management functions for the MIPS32 and MIPS64 architectures.
- *micromon.sx*: An ultra low-level, RAM-less ROM monitor program, which can be very useful when bringing up a new MIPS-Based design.
- *mipscp0.sx*: Low-level access to the coprocessor 0 registers, provided mainly for *[mips16] code which cannot use inline *asm* statements to access these registers.
- *muldivem.c*: A software multiply and divide instruction emulator for CPU cores that don't have the hardware multiplier unit.
- *noc1.sx*: Dummy floating point coprocessor functions for PUs without an FPU.
- *notlb.sx notlb_ops.sx*: Dummy TLB functions for PUs without a TLB.
- *r3kcache.sx, r3kcache_ops.sx, r4kcache.sx, r4kcache_ops.sx, r5kcache.sx, r5kcache_ops.sx*: Cache initialization and management functions for the generic R3000, R4000 and R5000 families.
- *r54cache.sx r54cache_ops.sx*: Vendor-specific cache handling for the NE R54xx family.
- *rc32cache.sx rc32cache_ops.sx*: Vendor-specific cache handling for the IDT R 32364.
- *rm7kcache.sx rm7kcache_ops.sx*: Vendor-specific cache handling for the PM -Sierra RM7000.
- *r3kc1.sx, r4kc1.sx, r5kc1.sx*: Floating point coprocessor (CP1) initialization, register save/restore and control functions for the R3000, R4000 and R5000 families.
- *r3ktlb.sx, r3ktlb_ops.sx, r4ktlb.sx, r4ktlb_ops.sx*: TLB initialization and management functions for R3000-class and R4000-class memory management hardware.

- *romlow.sx*: The ``from reset'' initialization code, and boot exception handler. With the co-operation of board-specific functions, this receives a rommable program to the point where the normal run-time environment can be started. See Section 8.4.1 “CPU Reset Handling” below.
- *unaligned.c*: Unaligned-access exception handler and emulator.
- *watch.c*: Generic API to the CPU hardware watchpoint facilities, if available.
- *watchsup.c*: Support code for CPU hardware watchpoint facilities.
- *xcptlowb.sx*: Low-level MIPS exception handler.
- *xcptlow.sx*: Alternative low-level exception handler, for more complex environments.
- *xcptcache.s*: Example low-level R4000 “cache error” exception handler (see also `ecchandler.c`).
- *xcpt.c*: Higher-level exception support code, including default exception handler.
- *xcptshow.c*: Functions to report an exception status on the console.
- *xcptshowmin.c*: Functions to report an exception status on the console, small version.

8.4.1 CPU Reset Handling

The source file `... sde/kit/share/romlow.sx` is used only when building a standalone, rommable program, and is compiled into a board-specific object file. Unsupported users receive it in a pre-compiled object file in the board directories.

It is always linked at the beginning of the ROM, and this should be the virtual address where the CPU starts execution on a hardware reset - that is 0xbfc00000, or 0xffffffffbfc00000 on a 64-bit processor, which map to physical address 0x1fc00000.

It includes the following:

- A template showing one way to provide a monitor entry point table, should such a thing be required.
- The assembler code required to get a MIPS architecture CPU from a reset exception to the point of initializing the C/ C++ run-time environment. Part of this is target-dependent, and is accomplished by calling the board-dependent `_sbd_reset` function, which is defined in the target-specific directory.
- The code to copy the instruction and read-only data segment from ROM to RAM. This copy is done only if the.text section has *not* been linked to start at the base of the ROM, and that is usually done only if you want to be able to set breakpoints in, and single-step through standalone programs.
- The code to copy the initialized, writeable data section from ROM to RAM. The *sde-conv* program, when given the **`-p`** option, concatenates the initialized data segment to the end of the instruction and read-only data segment.
- The code to re-vector Boot Exception Vector (BEV) exceptions to the address held in kernel reserved register *k0* (*$26*). Boot exceptions are used before RAM and caches have been tested and enabled (in normal operation the CPU vectors via cached RAM space, i.e. a low KSEG0 address). If k0\|==\|zero, then it attempts to display a “catastrophic Exception” message on the system console, indicating the location and cause of the error.

The file `.../sde/kit/share/ramlow.sx` is simply a dummy version of the `romlow.sx` file, which is used when building programs to be downloaded to RAM on a target with an existing monitor.

8.4.2 Exception Handlers

The files `.../sde/kit/share/xcptlowb.sx` and `xcptlow.sx` implement two alternative forms of the lowest level of exception handling for MIPS processors. Their job is to save the current processor state in a stack frame known as an xcptcontext (defined by <*mips/xcpt.h*>), set up a fresh run-time environment, and then call a C function. When the C function returns, they restore the saved processor state and return to the interrupted program. Note that these low-level handlers neither save nor restore the floating point registers: your exception handling routines must explicitly call `fpa_save()` and `fpa_restore()` if they need to use, examine or modify any floating point registers. We recommend that exception level code should not perform floating point arithmetic!

The simplest and fastest handler is the default `xcptlowb.sx`. This handler remains on the current "application" stack, pushes a new *xcptcontext* frame, and then calls a standard handler which does further dispatching to individual exception handlers (see `xcpt.c`, described below).

More complex run-time environments may need to use the `xcptlow.sx` handler, or some hand-crafted combination of the two. The `xcptlow.sx` file implements a separate exception-level stack, which is necessary if the stack pointer might not be valid on an exception (e.g. it may point to an unmapped address in KUSEG or KSEG2). Additionally the code uses a low-level dispatch table (`xcpt_astab`) which could allow certain exceptions to be handled quickly in assembler, without the overhead of saving/restoring a complete exception context (e.g. low-latency interrupt handling).

The higher-level exception handler is in file `.../sde/kit/share/xcpt.c`, and its associated header file is <*mips/xcpt.h*>.

8.4.3 Remote Debug Stub

When EJTAG is not available, remote debugging requires that the target board runs some sort of communications protocol which allows *sde-gdb* on the host development system to control and examine the program running on the target. This usually operates over a serial line, or perhaps over Ethernet.

When a program is being run under the control of a board's PROM monitor, and that monitor implements a supported remote debug protocol (which is true for the YAMON monitor, IDT/sim and LSI PMON), then you will probably use the PROM monitor's built-in remote debug support.

But if the program is running standalone (i.e. there's no separate monitor), or if your PROM monitor does not run a *gdb* debug protocol, then your program must have the remote debugging protocol code linked into it. This is implemented by the remote debug stub in `dbg.o`; if you have source code it will be in `.../sde/kit/share/dbg.c`.

If you use the example makefiles and their standard startup code then the debug stub will be automatically linked into your program, and initialized when both:

1. You are building a *rommable* version of the program, or the selected monitor does not implement a supported *gdb* remote debug protocol, and

2. The **`BRDEBUG`** makefile variable is defined as "`yes`"or "`immed`". See Section 3.2 "Example Makefiles".

Once the debug stub has been initialized, it will then only take control if an unexpected CPU exception occurs. However if `RDEBUG=immed` was defined, then an immediate breakpoint is taken before your main program is started, to allow initial breakpoints to be set.

8.4.3.1 Hardware-specific Debug Support

The remote debug stub contains some support for catching hardware interrupts, e.g. a debug button, or a control- C (ASCII 0x03) received on the debug serial port. See the `_dbg_signals()` function in `.../sde/kit/P4000B/sbddbg.c` for an example of how to do this.

To support debugging of code in Flash memory, the debug stub performs all accesses to memory via a set of cover functions. See `_dbg_put_byte()` et al in `.../sde/kit/p4032/sbddbg.c`.

You can also use the **`DSIMULATESSTEP`** compile-time option to avoid having to rewrite a whole Flash sector on every single-step (see `.../sde/kit/share/dbg.c` for its effect).

It is also possible to integrate the debug stub with your own (perhaps interrupt driven) I/O system, by implementing your own version of the functions found in `.../sde/kit/share/dbgsup.c`.

8.4.3.2 Multi-threading Support

The remote debug stub does contain some support for debugging multiple threads/processes. See the dummy functions at the start of `.../sde/kit/share/dbg.c`. Contact us if you need to use this feature. These stubs can be overridden by a multi-threading kernel to provide thread debugging.

Chapter 9

Retargeting the Toolkit

This section is a guide to retargeting or porting SDE to a new target board or simulator, and how to check your port with the example programs. While there's nothing to stop you doing this starting from SDE *lite*, one reason for supplying the run-time source code with the supported version of SDE is to help you to get your application up and running on a new MIPS-Based design with the minimum of extra programming. This section assumes you have all the files; unsupported users will have to figure things out for themselves.

To add support for a new board you should:

1. Create a new directory in `.../sde/kit` with the name of your board (e.g. `MYBOARD`).

2. Copy into this directory all the files from the board directory `.../sde/kit/SKEL`.

3. Edit each of these files, as described by the detailed comments within them, to control your on-board devices.

In many cases you may be able to use existing, shared files for UARTS, timers, etc, which are already used on other boards. There are many different boards and chipsets already supported: it is worth scanning other board support directories for sample code or simply for inspiration. The files which you will need to create are shown in Table 9.1.

Table 9.1 Board-specific Files

File	Description
`Makefile`	Trivial file which defines the board name and includes `../kit.mk`.
`sbd.mk`	Configuration file which describes the CPU type, endianness, presence of FPU, names of object files, memory map, etc.
`sbd.h`	Header file defining board-specific devices and registers, memory map, etc.
`sbdclock.c`	The low-level code to control the on-board timer. Most modern MIPS-Based CPUs (since the R4000 CPU) have an onchip counter and can use the common `r4kclock.c` driver; some other boards have drivers for offchip timers
`sbdflashenv.c`	Support functions for storing board environment variables in Flash memory (if available).
`sbdfreq.c`	The low-level code to determine the CPU clock frequency. This is only strictly needed when using an on-chip timer, where `sbdclock.c` needs to know this value.
`sbdfrom.c`	Support code for Flash memory programming: recognizes Flash memory address region and probes for Flash device.
`sbdfrom.h`	Defines the layout and type of Flash memory device(s).
`sbdmem.c`	Describes the physical RAM layout for memory allocation; only required if it is not contiguous.

Table 9.1 Board-specific Files (Continued)

File	Description
sbdmisc.sx	Miscellaneous low-level functions like mips_wbflush().
sbdnvram.c	Support functions for storing board environment variables in non-volatile memory (if any).
sbdpanel.c	Low-level code to display simple messages on on-board or front-panel LED alphanumeric display.
sbdpci.c	Support functions for initialization of host PCI bus controller (if any) and configuration of PCI devices.
sbdreset.sx	The code to initialize the on-board memory controller and any other board-specific reset code. This is only necessary if you intend to build standalone (i.e. rommable) programs.
sbdser.sx	A simple driver for the board's UART. Again this is usually only necessary for standalone programs; other programs will use the PROM monitor's I/O routines.
sbdtime.c	For boards that have a battery-backed real-time clock this file computes the current time in seconds since 00:00:00 Jan. 1, 1970.

Fortunately, if you already have a supported PROM monitor running on the board (e.g. the YAMON monitor, PMON or IDT/sim), or are running on a supported simulator, then many of these files can be dummied out; the monitor/simulator handles the power-on initialization and console I/O for you. The only board-specific files that require real code are sbdclock.c, and possibly sbdfreq.c, which are required to implement the interval timing functions (which you will need for benchmarking and profiling).

When performing a full port, then in order to support rommable code, particular care must be taken in the sbdreset.sx and sbdser.sx files. Until the generic code in .../sde/kit/share/romlow.sx has completed its job, then memory may not be used to store variables or a stack (it may not be enabled yet, and/or may have to be cleared to initialize parity, etc). The caches and FPU will also not be initialized yet, and cannot be used. The board-reset and low-level serial I/O code must therefore be capable of operating only in registers. Also tricky is that these functions (and anything which they call) must be position-independent because, until they are relocated, they may not at first be running at their final link address: absolute jumps may not be used, only branches and *bal* for subroutine calls. If you have to load the address of a code label or read-only data label, then you must add register s8 which holds the relocation factor, e.g.

```
la  a0,reset_tab
addua0,s8
lw  t0,0(a0)
```

Having created the new files and got them to compile, you can test them with some of the example programs:

- *Micromon*: built automatically as part of the board-support kit, it can be used test the reset and serial I/O code even before a new board's memory controller is working. The ultra low-level monitor interprets a ``reverse polish'' stack-based command language allowing you to probe devices and memory - press ‘?’ for help.

- *Kittest*: should be used to check that the low-level serial I/O code as part of the full C environment.

- *Minimon*: the mini command-line monitor has a number of builtin commands which can be used to check out many of the remaining functions, as follows:

`cache`: should report the correct cache sizes.

`stat`: should display the correct memory size and CPU frequency.

`time`: should display the correct date and time, if you have a real-time clock chip.

`itimer`: checks that the timer support code is returning monotonically increasing values, and interrupting at the correct rate; it should run for exactly 120 seconds (check it with a stopwatch).

`ls /dev`: Directory listing should include "`flash0`" etc. if you have implemented Flash memory support; and "`panel`" if you have implemented front-panel display support.

`echo wow! /dev/panel`: Should display ``wow!" on your front-panel display, if implemented.

`dump /dev/flash0 0 16`: Should dump the first 16 bytes of your Flash memory, if detected.

- *Flash*: the Flash memory test/example should report each of your Flash memory devices, and run through to completion without any errors, if you have implemented Flash memory support correctly.
- *PCI*: the PCI test/example should enumerate and list all devices on your PCI bus, if you have implemented the PCI support code correctly.

9.1 Common Device Files

There are a number of files in `.../sde/kit/share` which provide support for common UART and timer chips. You may be able to use these directly for your board, by *#include*-ing them into your files, or simply use them for inspiration:

- *m82510.s*: driver for the Intel M82510 serial controller.
- *mk48t02.c*: support for the Mostek MK48T02 clock/calendar.
- *mpsc.s*: driver for the NEC uPD72001 serial controller.
- *ns16550.s*: driver for the NS16450/16550 UART.
- *r361clk.c*: interval timing support for the IDT R36100 on-chip timer.
- *r4kclock.c*: interval timing support for the on-chip timer found on most modern MIPS-Based CPUs; relies on the on-chip timer interrupt being enabled by your hardware engineer.
- *s2681.s*: driver for the Signetics SCN2681, Motorola 68681 and UMC UM26811 DUART.
- *s2681clk.c*: interval timing support using the timer on the S2681 DUARTs.
- *vacser.s*: driver for the serial-port on the VAC068 VME-bus controller.
- *z8530.s*: driver for the Z8530 DUART.

Appendix A

References

[Sweet99]
See MIPS Run, Dominic Sweetman (of MIPS Technologies), 1999, Morgan Kaufman, ISBN 1-55860-410-3.

We have to give special mention to this comprehensive guide to the MIPS Architecture and programming; firstly because one of us wrote it, and secondly because if you read it carefully enough we'll save time on support work.

[Farq94]
The MIPS Programmers Handbook, Erin Farquhar & Philip Bunce, 1994, Morgan Kaufmann, ISBN 1-55860-297-6.

Example-based programming book aimed at small MIPS-based systems.

[SGI96]
MIPSpro™ Assembly Language Programmer's Guide, Silicon Graphics Inc.

[Kane92]
MIPS RISC Architecture, Gerry Kane and Joe Heinrich, 1992, Prentice Hall, ISBN 0-13-584210-7.

Reference manual to MIPS instructions, focussed on the machine instruction level.

[Kern88]
The C Programming Language (Second Edition), Brian W. Kernighan and Dennis M. Ritchie, 1988, Prentice Hall, ISBN 0-13-110362-8.

Throw away all those cheerfully coloured fat books with big letters and lots of pictures. If you want to program in C you need this and nothing else.

[Lewine91]
POSIX Programmer's Guide, Donald Lewine, 1991, O'Reilly, ISBN 0-937175-73-0.

An introduction to and complete set of manual pages for the POSIX.1 programming interface, of which the SDE run-time system implements a generous subset.

Then there are reference works; we need to put these in, but you won’t read them unless you have to:

[POSIX88]
IEEE Standard 1003.1-1988, Institute of Electrical and Electronics Engineers Inc., 1985.

[ABI]
System V Applications Binary Interface - Revised Edition, Unix System Laboratories, Prentice Hall, ISBN 0-13-877598-2.

[MIPSABI]
System V ABI MIPS Processor Supplement, Unix System Laboratories, Prentice Hall, ISBN 0-13-880170-3.

[ELF]
Understanding ELF Object Files and Debugging Tools, Mary Lou Nohr (Editor), Prentice Hall, ISBN 0-13-091109-7.

[MD00410]
MIPS® SDE for Linux Getting Started Guide, MIPS Technologies, Inc.

The document which describes the SDE toolchain configured for native development Linux/MIPS kernels and applications.

[MD00374]
MIPS32® Architecture for Programmers Volume IV-e: MIPS® DSP Application-Specific Extension to the MIPS32® Architecture, MIPS Technologies, Inc.

[MD00378]
MIPS32® Architecture for Programmers Volume IV-f: MIPS® MT Application-Specific Extension to the MIPS32® Architecture, MIPS Technologies, Inc.

You can't (so far as we know) buy the following GNU manuals, but they're provided as part of SDE:

[Binutils]
All the object-code tools except the linker itself, which gets a separate manual [Ld].

[Conv]
The SDE-specific ELF file conversion tool (sde\-conv).

[Cpp]
The GNU C pre-processor; only for specialists.

[Gcc]
The compiler manual. Serious users should think about reading this through one time.

[Gdb]
The debugger manual. Probably for reference only.

[Gprof]
The profiler manual. Read this if you're planning to do performance analysis.

[Ld]
The linker manual. Read this if you need to go beyond the tricks used in SDE examples.

[Make]
Read this if you're keen to create makefiles even more exciting than those in the examples.

[Stabs]
Documentation on the data structures used to pass debugging information.

Appendix B

Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

This document may refer to Architecture specifications (for example, instruction set descriptions and EJTAG register definitions), and change bars in these sections indicate changes since the previous version of the relevant Architecture document.

Revision	Date	Description
1.1	October 18, 2004	• First version with this title. Based on an original document first published by Algorithmics Ltd in 1995.
1.2	October 27, 2004	• Change SDEMakefile to SDEmakefile • Warn upgraders about the new sde-insight command. • Note that inter-module optimization is not available with C++. • Add new errata.
1.3	October 27, 2003	• Updated errata and change history for 6.01.01 release.
1.4	December 14, 2004	• New for 6.01.02 release: addition of 64-bit support and N32 ABI.
1.5	March 22,2005	• New for 6.02.00 release. Addition of MIPS16, MIPS DSP, and MIPS MT ASEs. Board support kits for AMVP environment.
1.6	March 24, 2005	• Removed errata fixed in final 6.02.00 release.
1.7	March 29, 2003	• Reenabled **`-mcode-xonly`** compiler option.
1.8	April 21, 2005	• Improved MIPS16 support, including "`mips16`" and "`nomips16`" per-function attributes.
1.9	May 26, 2005	• Added MT debugging section; hardware watchpoints; extended "`set mdi asid`" command. Updated change history and errata for 6.02.02 release.
1.10	June 2, 2005	• Added AMD-64 Linux to list of supported hosts.
1.11	June 27, 2005	• Added a section listing the compiler's predefined macros. • Updated change history for 6.02.03 release.
1.12	October 3, 2005	• Updated change history and errata for 6.03.00 release.
1.13	May 12, 2006	• Updated change history and errata for 6.04.00 beta release.
1.14	May 31, 2006	• Added multi-VPE debugging section, and updated change history and errata for 6.04.00 release.
1.15	October 5, 2006	• Expanded the MT debugging section. Updated change history and errata for 6.05.00 release.
1.16	January 19, 2007	• Documented 74K core family and new DSP ASE revision 2 intrinsics. • Described new SDEthreads API and TSP support. • Added new board targets. Updated change history and errata for 6.06.00 release.
1.17	April 20, 2007	• Updated 74K core family with new name changes. Updated change history for 6.06.01 release.
1.18	April 11, 2008	•