Codescape GNU Tools for MIPS

Programmer's Guide

Filename : MIPS_Toolchain_Codescape_GNU_Tools_for_MIPS_Programmers_Guide.docx

Version : 1.1.97 External Issue

Issue Date : 09 Jun 2016

Author : Imagination Technologies Limited

Contents

List of Figures

List of Tables

1. Introduction

This is a programmers' guide for the 'Codescape GNU Tools for MIPS Bare Metal' (henceforth called 'MIPS toolchain' in this manual).

The MIPS toolchain is a software engineer's cross-development system for MIPS architecture processors, intended for statically linked bare metal applications or light-weight operating systems. It is a component of the Codescape SDK, which includes toolchain, debugger, target simulators and other useful development tools (both for bare metal and Linux user-mode application development).

This guide is specifically about the bare metal version of the MIPS toolchain.

MIPS SDK Versions

There are two versions of the MIPS SDK:

Codescape MIPS SDK Essentials

Codescape GNU Tools for MIPS Bare Metal

A GNU GCC-based toolchain for compiling embedded applications,

Codescape GNU Tools for MIPS Linux

A GNU GCC-based toolchain for compiling Linux kernels.

QEMU

Pre-compiled binary version of QEMU patched to support the latest MIPS architectures.

Code examples

Simple examples of code.

Codescape MIPS SDK Professional

Codescape GNU Tools for MIPS Bare Metal

GNU GCC-based toolchains for compiling embedded applications. These are in two configurations; 'mti' for MIPS32/64 R2 – R5 architectures and 'img' for MIPS32/64 R6.

Codescape GNU Tools for MIPS Linux

A GNU GCC-based toolchain for compiling Linux kernels.

Codescape Debugger

A fully-featured graphical debugger for a variety of architectures including MIPS. This debugger includes powerful scripting languages for interacting directly with debug adaptors and targets.

IASim simulator

A simulator system providing accurate simulations of a wide range of MIPS core architectures, from very basic older cores to the very latest.

QEMU

Pre-compiled binary version of QEMU patched to support the latest MIPS architectures.

Code examples

Simple examples of code.

Codescape MIPS SDK Essentials is available for free download. Codescape MIPS SDK Professional is available under licence.

1.1. What's in Codescape GNU Tools for MIPS Bare Metal

The MIPS toolchain is built around GNU tools tuned, enhanced and packaged by Imagination Technologies together with a set of C and C++ libraries.

In this document where a name or path can be for either the 'img' or 'mti' variation, '<config>' has been used. <config> can be either 'mti' or 'elf'.

1.2. Support

Support for the MIPS Toolchain is available from a variety of sources. Informal, community-based support can be found on the forum. Registered licensees can get support via email and the Imagination Technologies Partner Portal.

Support via forum

http://forum.imgtec.com/

General information about Linux on MIPS

www.linux-mips.org

Partner portal

Support for registered licensees of MIPS technology is available via the Partner Portal; https://partnerportal.imgtec.com

2. Documentation

GNU Toolchain Documents

General GNU compiler and linker documentation can be found in <toolchainroot>/share/doc, in PDF format.

MIPS Debug Boot MIPS Example Code

Explains how to configure and use Boot-MIPS to form a custom boot solution..

Debug Adaptor Low-level Bring-up Guide

How to use DA-net debug adaptors with Codescape's scripting tools to connect and bring up new hardware targets.

EJTAG Specification

The specification of the MIPS JTAG implementation.

Codescape Debugger documentation

Imagination Technologies' debugger, Codescape Debugger, comes with a comprehensive set of documentation in online help format. There are three parts to the online help:

General Debugger Online help

Describes how to use the debugger and all dialog options.

Codescape Scripting Help

A reference guide to the Codescape Scripting language.

Codescape Console

A reference guide to 'Codescape Console', a command-line target-bring-up scripting tool, using Python-like syntax. Examples of the use of Codescape Console can also be found in the Debug Adaptor Low-level Bring-up Guide.

Debugging Tutorials

Walk-through tutorials of how to use Codescape Debugger and configuring Linux Kernels for MIPS targets.

Software User Manuals

'SUM' manuals provide a full description of the architecture of a MIPS core, from a software perspective. Critical registers and the logical architecture of a core are described in these manual.

These manuals are available to architecture licensees on request.

Codescape Debugger Flex Licence Manager Setup Guide

Codescape Debugger and IASim target simulator both use Flex LM licence server software to verify the licence. This guide explains how to configure a Flex Licence Server for those products.

3. Installation

Installation instructions for the toolchain can be found in the HTML Welcome pages.

4. Known issues

4.1. FPU denormal handler

This version of the toolkit does not include a default handler for FPU denormal exceptions. Any such exception will raise an error at runtime unless a handler is provided within an application.

5. Overview of Codescape GNU Tools for MIPS Bare Metal

This section provides a quick overview of the major components of the Codescape GNU Tools for MIPS Bare Metal. MIPS bare metal toolchains are based on FSF GCC 4.9.2.

5.1. mti and img toolchain configurations

Codescape GNU Tools for MIPS Bare Metal come in two distinct configurations, each supporting a subset of the MIPS instruction architectures:

mips-mti-elf

MIPS32R2 and MIPS64R2 through to MIPS32R5 and MIPS64R5 as well as the MicroMIPS ASE for each of those architectures.

mips-img-elf

MIPS32R6 and MIPS64R6.

The configurations include different libraries to support the appropriate architectures.

Where filenames, paths and library names have similar names, this document uses <config>, where <config> can be either ‘mti’ or ‘elf’. For example, mips-<config>-elf-gcc will be mips-img-elf-gcc for the img toolchain.

5.2. Components

Tool	Description	Executable	Version
Compiler	MIPS compiler and compiler driver which invokes the compiler and/or core binary tools.	mips-<config>-elf-gcc mips-<config>-elf-gcc-4.9.0 mips-<config>-elf-cpp mips-<config>-elf-c++ mips-<config>-elf-g++ mips-<config>-elf-gcov mips-<config>-elf-gprof	2015.01-5 (Based on FSF GCC 4.9.2)
Libraries	C libraries	Newlib	2015.01-5 (Based on 2.1.0)
Binary utilities	GNU binary utilities	mips-<config>-elf-as mips-<config>-elf-ld mips-<config>-elf-ld.bfd mips-<config>-elf-ar mips-<config>-elf-addr2line mips-<config>-elf-c++filt mips-<config>-elf-elfedit mips-<config>-elf-nm mips-<config>-elf-objcopy mips-<config>-elf-objdump mips-<config>-elf-ranlib mips-<config>-elf-	2015.01-5 (Based on 2.25)

Tool	Description	Executable	Version
		readelf mips-<config>-elf-size mips-<config>-elf-strings mips-<config>-elf-strip	
GDB	Command line debugger	mips-<config>-elf-gdb	2015.01-5 (Based on 7.7)
Small C library	Small C run-time library optimized for size by Imagination.	For each valid muti-lib combination - libc.a libg.a libm.a	1.0.0

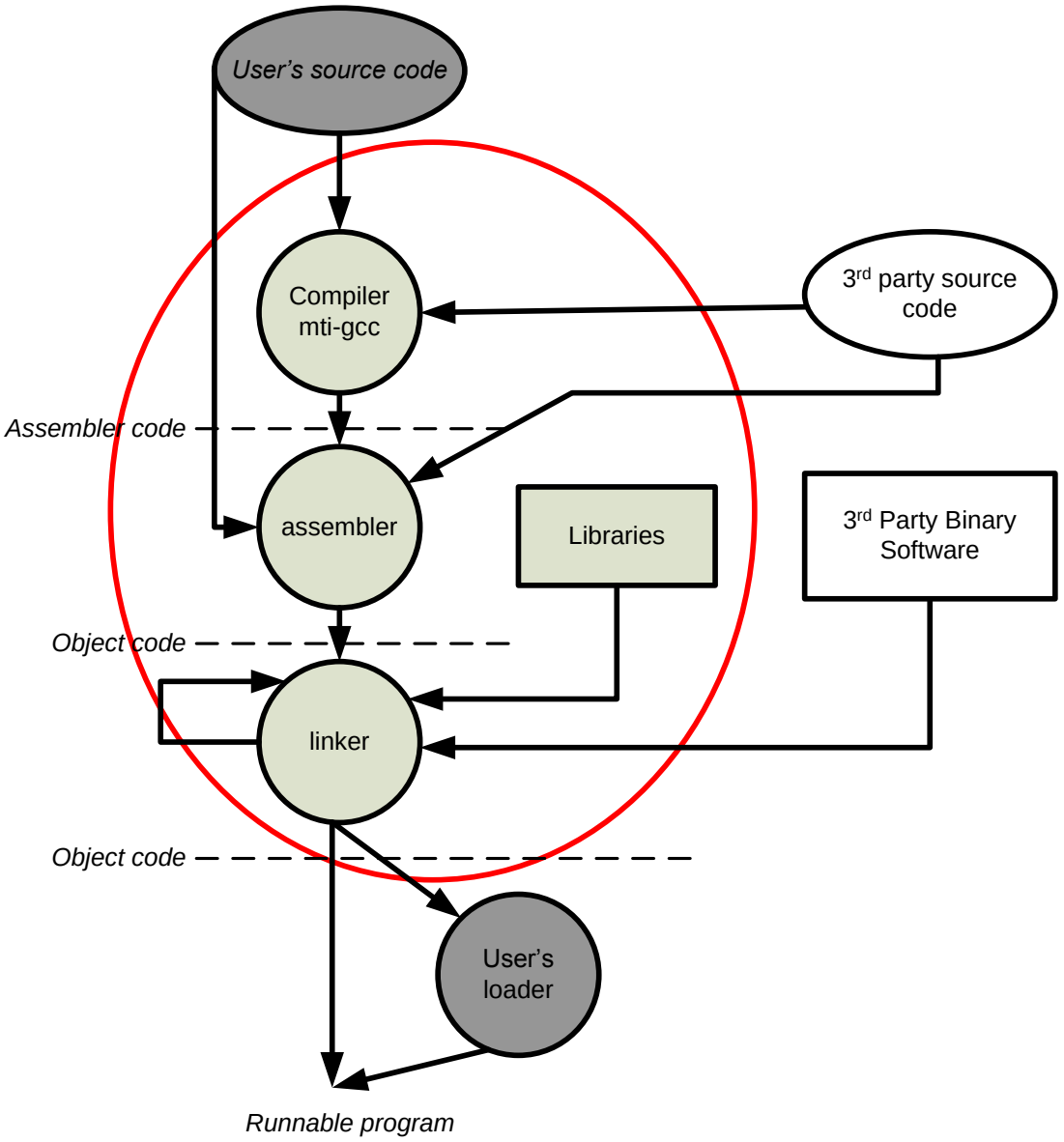

Figure 1 Toolchain components

5.3. Make

The toolkit includes standard GNU make utilities.

5.4. C Compiler

The following command-line options have been added to the compiler driver to enable new features:

Option	Description
-mmsa	Enable MSA extension.
-mclib=[newlib|small|tiny]	Select a C-library variant, defaults to newlib

5.5. C++ Compiler

We include the GNU C++ compiler (mips-<config>-elf-g++). The mips- <config>-elf-g++ compiler supports modern C++ features, and benefits from all Imagination Technologies' enhancements and optimizations to the common back-end of GCC. Use of C++ exceptions and/or run-time type identification incur a significant size overhead. If these features are not required by your code, then they can be switched off individually using the −fno−exceptions and −fno−rtti options respectively.

5.6. MIPS Assembler

The MIPS Toolchain version of the GNU assembler (mips-<config>-elf-as) is, as far as is possible, source code compatible with the ‘‘standard’’ MIPS assembler syntax. The tools are described in detail in the GNU manual [as].

5.7. Binary Utilities

The GNU binary utilities support the ELF object code format commonly used for MIPS embedded development. The tools are described in detail in the GNU manual [Binutils].

5.8. Libraries and Header Files

The MIPS Toolchain provides pre-compiled C runtime and Math libraries for a range of different MIPS ISA options; the version you need is picked automatically according to the flags you give the compiler - see ‘Multilib configurations’ on page 14. The C libraries and associated header files follow the ISO C Standard (ISO 9899:1990[1992]), also known as ISO C90, and formerly the ANSI X3J11 committee's standard for the C programming language (called the ANSI C standard in this manual). This has been validated using the Plum Hall Validation Suite.

MIPS SDK provides the following additional libraries for bare-metal environment. These libraries are available as both sources and binary components:

- Small C Library: size optimized standard C and Math libraries conforming to the ISO C standard (C99).

  SmallCLib has been specially tuned for embedded environment, where code size and run-time memory requirements are critical. SmallCLib aims to provide as much functionality as possible in the smallest amount of space. It is implemented in C with selective use of extended assembly and comes in two flavours:

  - A small flavour which is ISO C Standard (ISO 9899:1999) compliant and functionally identical to Newlib 2.0. Space saving is achieved by refactoring code to eliminate redundancy, occasionally at the expense of speed. In versions tuned for the latest MIPS ISA, new instructions are used aggressively, thereby reducing code size and in some cases improving performance. This variant is validated using the Plum Hall Validation suite.
  - A tiny flavour which diverges from both the ISO standard and Newlib behaviour in select areas. This variant is aggressively optimized for code size, in lieu of performance and features are reduced to the minimum set expected of a space-contrained embedded platform.

  Both variations are available as multiple pre-built configurations, where a small footprint library is deemed to be most relevant. However, the libraries can be built for any configuration following instructions provided with the SmallClib source release package. Small & Tiny C libraries share a common source base, derived from Newlib

Table 1 Feature comparison between libraries

	Newlib	SmallClib	TinyClib
Complex arithmetic	Yes	No	No

	Newlib	SmallClib	TinyClib
Character handling and localization	Full locale support	Full locale support	Only 7-bit ASCII (default C) locale support
Math	IEEE754 compliant. ISO C99 compliant. Additional error handling and exceptions not required by standards. Key string handling functions are tuned using hand-coded MIPS assembly.	IEEE754 compliant. ISOC99 compliant. Provides minimum error handling and exceptions as per standards. Tuned throughout using hand-coded MIPS assembly.	No support for denormals, infinity or NaN arguments. Minimum error handling required subject to above exclusions. Correct exceptions may not be generated. Tuned throughout using hand-coded MIPS assembly.
Re-entrancy	Yes	Not supported	Not supported

5.8.1. MIPS DSP Library

This library provides DSP and vector functionality relevant for multimedia applications. Library variants are optimized for various cores and leverage extensions like DSP-ASE and MSA to improve performance where possible. DSP Library sources are distributed under the MIPS Free To Use license. Refer to the MIPS DSP Library Reference and API manuals provided with the SDK for details.

5.8.2. Multilib configurations

The set of libraries provided as part of the MTI configuration of the MIPS Bare Metal toolchain are:.

Table 2 MIPS MTI Bare Metal toolchain multilibs

Libraries	Command line option
MIPS32 Release 2, Big-Endian, Hard-float	Default
MIPS64 Release 2, Big-Endian, Hard-float	-mips64r2
MIPS32 Release 2, Little-Endian, Hard-float	-EL
MIPS32 Release 2, Big-Endian, Soft-float	-msoft-float
MIPS32 Release 2, Big-Endian, Hard-float, NaN2008	-mnan=2008
MIPS32 Release 2, Big-Endian, Hard-float, NaN2008, Small Library	-mnan=2008 -mclib=small
MIPS32 Release 2, Big-Endian, Hard-float, NaN2008, Tiny Library	-mnan=2008 -mclib=tiny
MIPS32 Release 2, Little-Endian, Soft-float	-EL -msoft-float
MIPS32 Release 2, Little-Endian, Hard-float, NaN2008	-EL -mnan=2008
MIPS32 Release 2, Little-Endian, Hard-float, NaN2008, Small Library	-EL -mnan=2008 -mclib=small
MIPS32 Release 2, Little-Endian, Hard-float, NaN2008, Tiny Library	-EL -mnan=2008 -mclib=tiny

Libraries	Command line option
MIPS32 Release 2, Little-Endian, Soft-float, microMIPS	-EL -msoft-float -mmicromips
MIPS32 Release 2, Little-Endian, Hard-float, NaN2008, microMIPS	-EL -mnan=2008 -mmicromips
MIPS32 Release 2, Little-Endian, Hard-float, NaN2008, microMIPS, Small Library	-EL -mnan=2008 -mmicromips -mclib=small
MIPS32 Release 2, Little-Endian, Hard-float, NaN2008, microMIPS, Tiny Library	-EL -mnan=2008 -mmicromips -mclib=tiny
MIPS64 Release 2, Big-Endian, Hard-float, N64	-mips64r2 -mabi=64
MIPS64 Release 2, Little-Endian, Hard-float	-EL -mips64r2
MIPS64 Release 2, Big-Endian, Soft-float	-mips64r2 -msoft-float
MIPS64 Release 2, Little-Endian, Soft-float	-EL -mips64r2 -msoft-float
MIPS64 Release 2, Little-Endian, Hard-float, N64	-EL -mips64r2 -mabi=64
MIPS64 Release 2, Big-Endian, Soft-float, N64	-mips64r2 -msoft-float -mabi=64
MIPS64 Release 2, Little-Endian, Soft-float, N64	-EL -mips64r2 -msoft-float -mabi=64

The set of libraries provided as part of the IMG configuration of the MIPS Bare Metal toolchain are:.

Table 3 MIPS IMGBare Metal toolchain multilibs

Libraries	Command line option
MIPS32 revision 6, Big-Endian, Hard-float	Default
MIPS64 revision 6, Big-Endian, Hard-float	-mips64r6
MIPS32 revision 6, Little-Endian, Hard-float	-EL
MIPS32 revision 6, Big-Endian, Soft-float	-msoft-float
MIPS32 revision 6, Little-Endian, Soft-float	-EL -msoft-float
MIPS32 revision 6, Little-Endian, Single-float, Short-Double	-EL -msingle-float -fshort-double
MIPS64 revision 6, Big-Endian, Hard-float, N64	-mips64r6 -mabi=64
MIPS64 revision 6, Little-Endian, Hard-float	-EL -mips64r6
MIPS64 revision 6, Big-Endian, Soft-float	-mips64r6 -msoft-float
MIPS64 revision 6, Little-Endian, Soft-float	-EL -mips64r6 -msoft-float
MIPS64 revision 6, Little-Endian, Hard-float, N64	-EL -mips64r6 -mabi=64
MIPS64 revision 6, Big-Endian, Soft-float, N64	-mips64r6 -mabi=64 -msoft-float
MIPS64 revision 6, Little-Endian, Soft-float, N64	-EL -mips64r6 -mabi=64 -msoft-float

5.8.3. CPU Management

Along with the standard C run-time libraries, MIPS Toolchain run-time libraries provide a set of support functions. These functions can be used to maintain processor's caches, TLB and coprocessor registers. These functions are provided as libhal.a library and a set of header files under include/mips folder. The library can be linked using -lhal linker command line option and is automatically linked in when using the UHI link scripts.

Cache support functions

Cache support function prototypes are supplied by including <mips/cpu.h>.

```
void mips_size_cache (void)
```

Size caches without reinitialising and losing dirty cache lines. Sizes are returned in following global variables:

Primary instruction cache

- mips_icache_size: Size of cache
- mips_icache_linesize: Size of each cache line
- mips_icache_ways: The number of ways of set associativity

Primary instruction data

- mips_dcache_size: Size of cache
- mips_dcache_linesize: Size of each cache line
- mips_dcache_ways: The number of ways of set associativity

Secondary cache

- mips_scache_size: Size of cache
- mips_scache_linesize: Size of each cache line
- mips_scache_ways: The number of ways of set associativity

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

Synchronise icache and dcache for virtual address range

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

Writeback and invalidate address range in all caches

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

Writeback and invalidate address range in data caches

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

Writeback and invalidate address range in instruction caches

```
void mips_flush_cache (void)
```

Writeback and invalidate all caches

```
void mips_flush_dcache (void)
```

Writeback and invalidate data caches only

```
void mips_flush_icache (void)
```

Writeback and invalidate instruction cache only

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

Load and lock a block of instructions into the instruction cache

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

Load and lock a block of date into the data cache

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

Load and lock a block of instruction/data into the secondary cache

TLB support functions

TLB support function prototypes are supplied by including <mips/cpu.h>. They are used for Translation Lookaside Buffer (TLB), if present.

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

Returns the number of entries in the TLB

```
void mips_tlbinval (tlbhi_t a0)
```

Invalidate an entry

```
void mips_tlbinvalall (void)
```

The function is same as mips_init_tlb . It invalidates the all entries in TLB.

```
void mips_tlbri2(tlbhi_t *hi, tlblo_t *lo0, tlblo_t *lo1, unsigned *mask, unsigned index)
```

Reads the TLB entry with specified by index, and returns the EntryHi, EntryLo0, EntryLo1, and PageMask parts in *hi, *lo0, *lo1 and *mask respectively.

```
void mips_tlbwi2(tlbhi_t hi, tlblo_t lo0, tlblo_t lo1, unsigned mask, unsigned idx)
```

Writes hi, lo0, lo1 and mask into the TLB entry specified by idx

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

Writes hi, lo0, lo1 and mask 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.

Coprocessor 0 (CP0) Intrinsic

The Coprocessor 0 (CP0) is used to control interrupts, exceptions, memory management, caches, etc. The CP0 intrinsics provide access to the CP0 registers. The header file <mips/cpu.h> provides the intrinsic shown in table below and described in the following subsections. The “*” symbol represents up to five separate intrinsic.

Table 4 cpu.h intrinsics

*	Arguments	Operation
get	()	Returns the register value.

*	Arguments	Operation
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.
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.

```
mips_*sr
```

Operations on the Status register (CP0 register 12).

```
mips_*cr
```

Operations on the Cause register (CP0 register 13).

```
mips_getcount, mips_setcount
mips_getcompare, mips_setcompare
```

Operations on the Count and Compare registers (CP0 registers 9 and 11).

```
mips_getprid
```

Return the read-only PrID register (CP0 register 15).

```
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.

```
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).

```
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).

```
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.

```
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)

`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.

Miscellaneous functions

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

```
void _mips_sync (void)
```

This generates a sync instruction.

```
unsigned char mips get byte (void *addr, int *err)
unsigned short mips get half (void *addr, int *err)
unsigned int mips_get_word (void *addr, int *err)
unsigned long long mips_get_dword (void *addr, int *err)
```

Return the byte, half-word, word, or double-word at address addr.

```
mips_put_byte (void *addr, unsigned char val)
mips_put_half (void *addr, unsigned short val)
mips_put_word (void *addr, unsigned int val)
mips_put_dword (void *addr, unsigned long long val)
```

Store a byte, half-word, word, or double-word val to arbitrary address addr.

5.9. Codescape Debugger

Codescape MIPS SDK Professional includes Codescape Debugger, a graphical debugger.. Codescape debugger is a multi-target fully featured debugger that supports disassembly, tracing and more. See ‘Graphical Debugger’ on page 25 for a detailed feature list.

Example Programs

A collection of example programs is provided with the MIPS Toolchain. These enable you to test your build and debug environment by providing simple known MIPS-compatible code examples, makefiles and instructions on use with simulated targets.

5.10. QEMU simulator

QEMU simulator is supplied as a binary, compiled to simulate MIPS targets. This can be used to run code compiled for most MIPS instruction sets.

Source code for QEMU can be found here:

```
https://github.com/prplfoundation/qemu
```

5.11. IASimulator

IASimulator is an instruction-level core simulator for debugging bare-metal applications. It is provided with Codescape MIPS SDK Professional. Simulators for different MIPS cores can be started from within Codescape Debugger.

6. Common Usage and Recommended Practice

6.1. MIPS SIMD Architecture (MSA)

MSA extensions can be enabled by providing the following command-line option to the C compiler:

```
<prefix>-gcc -mmsa …
```

MSA instructions and vector registers are available directly for hand-written assembly. The compiler also provides C-style built-in functions for all MSA instructions in conjunction with vector-type data structures to target vector registers. The preferred coding style is to use regular C arithmetic operators on vector data types and fall back to built-ins for complicated MSA instructions which the compiler is unable to relate to vectorized C code.

Examples are provided in MD01067 - MIPS SIMD Architecture (MSA) Programmer's Guide and also on the Codescape MIPS SDK.

6.2. Linking with the small C Library

Selection of standard C library is controlled by a new command-line option to the compiler driver

```
gcc -mclib=[newlib|small|tiny] ...
```

If no -mclib option is specified, newlib will be used. At compilation stage, the mclib option automatically enables various options and optimizations relevant to small foot-print builds. At link stage, this option controls the library search path so that the desired flavour of the standard C library gets linked against. If specified, this option must be consistent across compile and link stages.

Note: Small/tiny library variants are only provided for particular build options. If mclib=[small|tiny] is specified in conjunction with an unsupported combination of multilib options, the compiler will link with the default C library(newlib), unless an alternate library search path is explicitly supplied. You can check available configurations by invoking the compiler with option '–print-multi-lib'

7. Graphical Debugger

Codescape MIPS SDK Professional includes a graphical debugger, Codescape Debugger. This is a mature, fully-featured debugger that includes multi-target, multi-platform debug functionality.

This section briefly describes some of the main features of the debugger.

7.1. Multi-target connection

Codescape Debugger can not only connect to multiple target types, it supports simultaneous connection to different targets.

Target types supported include:

- MIPS or Meta-based cores (via a Debug Adaptor)
- MIPS-based cores via Navigator Probe debug adaptor
- Simulated targets via built-in simulator
- Linux userland debug via GDB server

Cores are identified by processor ID, so the debugger can support debugging on single-target, multi-core SoCs.

7.2. RTOS debug and tracing

Codescape supports debug and trace of several common real-time operating systems, including ThreadX, FreeRTOS, MeOS and MeOS2.

7.3. Debug regions

Codescape Debugger provides many functions via ‘regions’. These are panes that can be docked, floating or stacked in a group with tabs.

The regions include

Disassembly

The Disassembly region enables you to debug code at instruction level. Assembly code is displayed optionally interleaved with source code.

Breakpoints, start, stop and stepping execution, moving the PC and viewing source can all be done from within the Disassembly region.

Source

The Source region is an integrated text editor for source code that also enables the placing of breakpoints and issuing run/stop commands.

Memory

Displays memory addresses and the contents of the addresses. Contents of addresses can be edited directly from the region.

Peripheral

The Peripheral region displays the contents of registers in your target processor that are mapped to peripheral devices. The contents of registers can be edited directly from the region.

Call Stack

Displays active function calls and recent history of function calls. You can jump directly from a function call to the corresponding source code or disassembly of the function.

Register

The Register region shows the contents of a processor’s register block and flags.

When you first open the region it shows a set of useful general registers. You can choose to show different types of register from the right-click menu, the types of register you can display depends on the target processor and the template in use. They can include DSP, Floats, Modulo addressing and System registers etc.

Watch

The debugger has two sorts of watch region; Local Watch and Watch.

Local Watch automatically displays all local variables in view from the current position of the PC. Variables are automatically added to the display as they come into the scope of a function.

Watch region displays the value of specified registers, bitfields, variables (from source code), symbols and expressions, one per row. ‘watches’ can be added to the region by dragging and dropping from other regions or directly entering expressions. A valid expression for a Watch region is any expression that evaluates to a symbol, address or register.

Cache

The debugger has two cache regions; icache and dcache.

‘icache’ displays the contents of the instruction cache on MIPS targets.

‘dcache’ displays the contents of the data cache on MIPS targets.

Script

The Script region enables you to run Python and Codescape script commands within Codescape. Combined with the Codescape scripting commands, this region enables you to develop your own debugging regions as plugins. Several plugins are supplied with Codescape Debugger and can be used as the basis for developing more functions.

Overlays

The Overlay region displays any overlays used in the thread selected for the region.

Terminal

The Terminal region allows users open a VT100-emulation terminal connecting to a target via a serial or debug channel port. It can be used as a direct replacement for hyperterminal. The benefit of the Terminal region is that it is integral to Codescape.

Profiler

The Statistical Profiler included with Codescape Debugger is an analysis tool for examining the run-time behaviour of programs. It shows where your program spends its time so you can identify inefficient sections of code.

When your program executes, the Profiler takes regular samples of the processor’s program counter (PC) at a configurable rate from 1Hz to 1kHz. From the address returned from each sample it can resolve which function your program is in at that moment in time

TLB

Codescape’s TLB region displays the contents of the Translation Lookaside Buffer for MIPS cores.

The contents can be displayed in two modes:

- Programmer’s view
- TLB/MMU register view

Switching between modes is done by right-clicking on the region and selecting ‘Change Display Mode’.

Programmer’s view

The Programmer’s view\u2019 displays the registers that can be seen from a program. These are the EntryHi, EntryLo (0 & 1) and the Page Mask. The index is also displayed.

TLB/MMU register view

The TLB/MMU register view displays the page frame number (PFN), virtual page number (VPN), corresponding physical addresses and information on the state of the TLB data.

7.4. Execution control

Codescape Debugger offers many methods of controlling the execution of programs. Some are applicable to a selected thread, some act globally to all current targets.

The following list is a summary of the execution control functions:

Reload, reset and restart

On loading a program file, you have the options to reset the target, run to a specific address or not run. Resetting a target will produce the choice of hard resetting, soft resetting and you can also opt to reload the previously-loaded program file.

Programs can be restarted at any time.

Breakpoints

Breakpoints can be set on addresses, registers, overlays, and rangest. Breakpoints can be either hardware or software.

Hardware or software can be used, and the choice is configurable per target or globally for all targets. Software breakpoints are the default choice unless you alter the target debug options.

Breakpoints can also be conditional on an expression.

Scripts can be run when a breakpoint triggers or simple printf-syntax log expressions to output values.

Run commands

Execution can be controlled by thread, target or globally (all current targets). As well as standard start, stop and single step commands, Codescape offers some special run operations:

Run to address

Allows you to specify an address or expression (which must evaluate to an address) as the point at which to stop. This is equivalent to setting a breakpoint at an address and running until the breakpoint it hit.

Run to cursor

Runs until the cursor location (in Disassembly or Source region) is reached.

Step over

This performs a single disassembly step, stepping over function calls if present.

Step out

Runs until the current function is returned.

Step run in

Runs to, then stops at, the start of each successively nested function call. That is, run until a new function is entered, or the current function is returned.

Step run out

Runs to, and stops after, each successively nested function call has completed.

Step run

Performs a single disassembly step.

Step Run Until

Single steps until the expression is true.

Set PC to cursor

Sets the address in the PC register to the current location of the cursor. This could be a line of disassembly or a line of code in a Source region

7.5. Linux application debug

Codescape Debugger can provide debug of Linux userland applications via GDB server running under Linux on the target.

Codescape connects to GDB server over a specified port, displaying GDB server as a target.

The userland application can then be loaded via Codescape's 'Load Program File' function. This will download the application binary to the Linux filesystem on your target and run it via GDB server.

7.6. Semihosting

The MIPS Toolkit and Codescape Debugger provide support for semi-hosting functions from a target via a built-in API within the toolkit. The debugger allows you to set a root directory for semi-hosting operations so that programs running on a target can use a relative path for file operations.

Any application compiled using the UHI link scripts (uhi*.ld) from the toolchain will have support for semi-hosting included.

Semi-hosting operations supported include:

- File operations such as fopen, fwrite, fread and fclose.
- stdout, stderr and printf (output is sent to the debugger for display).

A separate document, the UHI Reference Manual, provides a detailed description of the UHI architecture.

7.7. Hardware Support Packages

Hardware Support Packages (HSPs) are sets of files supplied with Codescape Debugger that provide target-specific information such as the layout of memory and register definitions.

Data contained in an HSP includes:

- Memory layout including addresses and size.
- Coprocessor register information, including addresses, offsets and size

A hardware target built on an established MIPS core will usually be able to be debugged using one of the standard HSPs. However it may be necessary to modify some details of the memory layout for specific implementations.

On connection to a target, the debugger attempts to automatically select the appropriate HSP based on the core processor ID.

7.8. Make Manager

Codescape's Make Manager provides a quick and simple way to call 'make' from within Codescape.

A suitable version of 'make' must be configured and working on your host computer in order to use Make Manager. Absolute paths to 'make' can be specified.

Multiple configurations for 'make' can be specified, with individually-specified parameters for each configuration. The location of 'make' can also be specified in the configuration.

8. Profiling

8.1. Statistical profiler

The Statistical Profiler included with Codescape Debugger is an analysis tool for examining the run-time behaviour of programs. It shows where your program spends its time so you can identify inefficient sections of code.

When your program executes, the Profiler takes regular samples of the processor's program counter (PC) at a configurable rate from 1Hz to 1kHz. From the address returned from each sample it can resolve which function your program is in at that moment in time.

Over a period of execution time, a profile of these samples is built-up which shows the approximate percentage of time spent in each function of your code. These values are used to identify hotspots where your program spends disproportionate amounts of time.

Statistical profiling is passive, which means it does not modify your code to gather performance data. Therefore your program's execution time is not affected when the Statistical Profiler is running.

8.2. RTOS Trace and Watch

The RTOS Trace region in Codescape Debugger reads, processes and displays trace data that is output from real time operating sytems (RTOS).

Trace data is written to a buffer within the RTOS. Codescape Debugger reads this buffer and displays the data in the regions. The buffer can only be read when the RTOS is stopped, so data shown in the RTOS regions will not be refreshed until the RTOS has been halted.

If the buffer is full, then the write action wraps round and overwrites data already in the buffer. The ThreadX RTOS outputs a large amount of data to the buffer, so the overwrite is likely to happen.

This trace function is available for FreeRTOS, MeOS and ThreadX. More details can be found in the Codescape Online Help, Optimization section.

8.2.1. Building RTOS for trace

Each RTOS must be configured so that the RTOS Trace functions can be used. The configurations are listed below.

ThreadX

Compile with -DTX_ENABLE_EVENT_TRACE

History across updates is maintained by utilising event timestamps.

MeOS

RTOS Trace for MeOS requires that MeOS be compiled with -DMEOS_INCLUDE_TRACE.

FreeRTOS

No specific compile requirements.

8.3. PDTrace

The PDtrace specification provides trace control and formats for both the processor-specific information captured from each pipeline within the processor and for the non-processor specific blocks, such as the CM (Coherence Manager) block in the CMP system, including the details of how the trace from multiple on-chip blocks are combined to provide a single trace stream on the chip interface pins.

The type of information that is captured in the trace stream and put into memory is controlled by CP0 control registers defined in the MIPS32 and MIPS64 architectures and by TCB (Trace Control Block) control registers defined in the PDtrace architecture.
CP0 control registers can be programmed by user applications so long as the needed hardware components and trace memory are present.

The TCB registers allow users to control tracing at the execution time of applications, using an external agent like the debugger that communicates with these control registers using a DA-net.

Trace data can be displayed in the Trace Results region in Codescape Debugger.

9. Example Programs

Example code and makefiles are provided, demonstrating how to build simple code for MIPS targets. All examples default to building for big-endian cores using an R2 instruction set.

Note: Examples are installed to the home directory of the installing user.

Environment variables

MIPS_ELF_ROOT

This should be set to the root of the installation directory for the Bare Metal Toolchain (that is, below the bin directory)

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

9.1. Individual Examples

9.1.1. Hello World!

This is the simplest program and can be used to test that your build environment is configured correctly. It is also useful for testing that you are building correctly for your target.

Example location

Linux

```
/{HOME}/imgtec/examples/baremetal/hello
```

Windows

```
\{HOME}\imgtec\examples\baremetal\hello
```

Files in the example

hello.c

Standard hello world example code.

Makefile

A simple makefile is supplied. It requires an environment variable (MIPS_ELF_ROOT) to be defined to point to the location of the MIPS toolchain. Big or little-endian ELFs can be produced from this makefile using the switch make LITEND=1. Big-endian is built by default.

9.1.2. C++ Demo

This example builds a small C++ program that performs string handling operations.

Example location

Linux

```
/{HOME}/imgtec/examples/baremetal/cxxtest
```

Windows

```
\{HOME}\imgtec\examples\baremetal\cxxtest
```

Files in the example

tststring.cc

A string handling test program from the GNU libstdc++ library.

Makefile

A simple makefile is supplied. It requires an environment variable (MIPS_ELF_ROOT) to be defined to point to the location of the MIPS toolchain. Big or little-endian ELFs can be produced from this makefile using the switch make LITEND=1. Big-endian is built by default.

Note that this makefile uses the stdc++ C++ libraries and has an additional rule for compiling C++ programs (compared to the other example makefiles).

9.1.3. Mergesort semihosting example

This example uses a simple array-sorting program to demonstrate the Unified Hosting Interface operations(UHI). The UHI library provides semihosting functions for MIPS targets.

The program uses file commands such as fopen and fwrite to read and write from text files on the debug host. These commands, along with printf, rely on the UHI library.

The code must be built using the uhi32.ld file (specified using the –T option on the command line when cross-compiling). See the included README file for information on building using this Makefile.

Example location

Linux

`/{HOME}/imgtec/examples/baremetal/mergesort`

Windows

`\{HOME}\imgtec\examples\baremetal\mergesort`

Files in the example

mergesort.c

Example source code.

mergesort.elf

Pre-built big-endian ELF.

Makefile

A simple makefile is supplied. It requires an environment variable (MIPS_ELF_ROOT) to be defined to point to the location of the MIPS toolchain. Big or little-endian ELFs can be produced from this makefile using the switch make LITEND=1. Big-endian is built by default.

Inputdata.tx

Contains text strings that are read in by the ELF.

9.1.4. MSA transposition example

Simple demo program to transpose a matrix of signed 16-bit integers.

Example location

Linux

`/{HOME}/imgtec/examples/msa/msa_transpose`

Windows

`\{HOME}\imgtec\examples\msa\msa_transpose`

Files in the example

Transpose4x8_h.c

Example source code.

Makefile

A simple makefile is supplied. It requires an environment variable (MIPS_ELF_ROOT) to be defined to point to the location of the MIPS toolchain. Big or little-endian ELFs can be produced from this makefile using the switch make LITEND=1. Big-endian is built by default.

9.1.5. MSA vector addition example

This MSA example is a simple demo program to add 2 vectors of 32-bit integers.

Example location

Linux

```
/{HOME}/imgtec/examples/msa/msa_vadd
```

Windows

```
\{HOME}\imgtec\examples\msa\msa_vadd
```

Files in the example

vadd_w.c

Source code.

Makefile

A simple makefile is supplied. It requires an environment variable (MIPS_ELF_ROOT) to be defined to point to the location of the MIPS toolchain. Big or little-endian ELFs can be produced from this makefile using the switch make LITEND=1. Big-endian is built by default.

10. Porting an ISO / ANSI C Program

This chapter is intended to help you port an existing C application or benchmark program that is compatible with the C library defined by the ISO C90 or ANSI X3J11 standard, as described in [Kern88]. Most simple, self-contained programs will port with no difficulty. The easiest approach to porting is as follows:

1. Create a new sub-directory in the .../examples directory (if you're used to an integrated environment, this subdirectory will be your ''project'') and put your source code there.
2. Copy the makefile from the most similar example and edit that
3. Edit the new makefile. Note that object files have the .o extension, not .obj or anything else.
4.
5. If you need to measure the execution time of small sections of your code, then use the clock() function, or refer to elapsed time below.
6. Make and run your program. Don't use a high loop count in benchmarks, as the simulator is not fast (hint: use ''#ifdef SIM'' to select a smaller loop count

The obvious portability considerations of byte-endianness and word size shouldn't require any explanation these days. But you should be aware of the following special considerations which apply to programs built with MIPS Toolchain's run-time system, as compared to the environment provided on a full-blown UNIX-like system.

- File I/O: other than to or from the console terminal is possible when using a probe or simulator, or on boards with network hardware and suitably equipped PROM monitors. See 'Semihosting - Unified Hosting Interface (UHI)' on page 42.
-

10.1. Common problems when converting to MIPS architecture

These remaining points are general warnings about idiosyncrasies of the MIPS architecture and its compilers, which can cause confusion when porting programs.

- Unaligned addresses will cause an ''Address Error'' exception (a SIGBUS signal). This won't affect most programs since the compiler correctly aligns structure fields unless specifically instructed otherwise. The malloc() family also aligns all requests to an 8-byte boundary (the maximum ever required by the CPU). But beware when type-casting pointers to small types into pointers to larger types (you can try using the compiler's −Wcast−align option to catch these).
- Null pointer references will cause a ''TLB Miss'' exception (a SIGSEGV signal), unless you set up a dummy TLB mapping for address 0. Memory is normally accessed through the cacheable KSEG0 or uncacheableKSEG1 address spaces, which begin at 0x80000000 and 0xa0000000 respectively.
- Use of ''short'' variables often prevalent in programs written for 16-bit or x86 processors, generates inefficient code on MIPS architecture processors, particularly if used for for loop counters and array indices. There are no MIPS instructions which operate on sub 32-bit values, and they have to be synthesised from multiple instructions. Although the compiler attempts to avoid excessive conversions, always use ''int'' for such purposes, unless you specifically need the semantics of 16-bit arithmetic.
- Character signedness : ISO and ANSI C permits char variables to be implemented as either signed or unsigned − it's compiler dependent. MIPS compilers historically made ''char'' variables default to unsigned (because it makes faster code); if your program has been developed in a context where those variables were signed, it may not work correctly on MIPS; you may get caught out by mistakes like assigning the integer result of getc() to a char variable, and then comparing that with EOF(integer −1).

  You can specify ''signed char'' explicitly for individual variables − which will make your code more portable. But if it is deeply ingrained in your application, then you can use the compiler's −fsigned−char option, which changes the default.

- Bitfield signedness. Some compilers arbitrarily treat bitfields as implicitly unsigned, but this is not the case for GCC, which uses your type definition as written. But accessing signed bitfields generates slower code, especially when using the MIPS16 ASE. You can either modify your structure definitions to add explicit ‘‘unsigned’’ type qualifiers, or change GCC’s default behaviour using its −funsigned−bitfields option.
- Small variables of 8 bytes or less are stored separately from larger variables, to allow them to be accessed more quickly. This can cause strange link-time errors if you have not declared your global variables consistently in all modules (‘‘relocation truncated’’ is the usual one.)

11. Linker Scripts and Object Files

11.1. Linker Scripts

The linker (ld) is always controlled by a script file (.ld file). The MIPS Toolchain provides linker scripts that pull in appropriate startup code and libraries. These linker scripts can be found in multilib-specific subdirectories. Linker script files are used to control how the input sections are mapped into output sections and to control the memory layout of output sections. Linker command line option -T<file> can be used to supply a linker script to the linker. The GNU Linker manual (ld.pdf) contains full details of the script language.

11.1.1. UHI linker script files

Linker script files for all three ABIs are provided in the toolchain. These scripts produce a runtime that is capable of running hosted, semi-hosted or standalone.

The scripts are named:

```
o32 ABI - uhi32.ld
n32 ABI - uhi64_n32.ld
n64 ABI - uhi64_64.ld
```

The supplied example programs make use of these linker scripts. See ‘Example Programs’ on page 31.

11.1.2. Symbols in linker script

Symbols can be defined in a linker script simply by assigning a value to it. Symbols defined in the linker script file are placed in symbol table with a global scope. Following example defines symbol __memory_size.

```
__memory_size = 0x10000;
```

In the linker script file, ‘.’ is a special symbol which represents the location counter. It can be read or written.

```
. = 0x80200000;              /* Assign 0x80200000 to location counter */
_end = .;                    /* Assign location counter to symbol _end */
```

The following table lists important symbols defined in the linker script file:

Table 5 Symbols defined in default linker script file

Symbol	Description
_ftext	Start of .text output section
_etext	End of .text output section
__MIPS_abiflags_start	Start of .MIPS.abiflags section
__MIPS_abiflags_end	End of .MIPS.abiflags section
_fdata	Start of read/write data section
_gp	Global data pointer
_edata	End of read/write data section
_fbss	Start of zero initialized read/write data section
_end	End of zero initialized read/write data section.

Symbol	Description
__stack	Start of stack (if not defined to zero).
__memory_size	Size of available memory.

Linker script file can be used to define start of stack and heap sections. Stack and heap are located at the end of .bss section (zero initialized read/write data section). Symbol __stack can be defined to a non-zero value to set the start of stack section. Following example sets start of stack to address 0x80201000.

```
__stack = 0x80201000;
```

There is no special symbol which defined start/end of heap section. Heap starts at the end of .bss section (symbol _end).

11.2. ELF Object File Format

MIPS Toolchain uses the ELF object file format, and aims to be able to interlink with most contemporary MIPS ELF versions. The format of the debug information passed from the compiler to the source-level debugger is independent; MIPS Toolchain uses DWARF.

ELF files can define multiple sections. Roughly speaking, the output of the assembler is a file containing one or more named sections; when two or more object files are linked, sections with the same name are combined; so the section ‘‘.text’’ is used for machine instructions, and by default all the instructions end up together. The compiler and the assembler generate quite a lot of different sections implicitly, and the default linker scripts built in to MIPS Toolchain know which segment (a segment is a chunk of the eventual program image) to put them in. See table below.

You can also deliberately place code or data in arbitrarily named sections if you want to take control over exactly where different chunks of your program end up in memory.

Much of the time you won’t really be aware of all these sections, but when you use one of the binary utility programs in MIPS Toolchain − nm, objdump, readelf, ld and so on − you will see those names.

Table 6 Standard ELF section names

Section name	What generates it	Where it ends up
.text	Compiler- or assembler-generated instructions	Executable code segment
.text.hot	Functions which are called frequently	
.text.unlikely	Functions which are called rarely	
.text.*	Functions when compiled with −ffunction−sections are output to uniquely named sections of this form	
−ffunction−sections are output to uniquely named sections of this form		
.gnu.linkonce.t.*	C++ methods − only one copy of each section with the same name is output to the code segment	

Section name	What generates it	Where it ends up
.init	Code to be run before main (e.g. C++ setup)	
.fini	Code to be run after _exit (e.g. C++ teardown)	
.rodata	Strings and C data declared const	Read-only data segment
.rodata.*	Constant data when compiled with −fdata−sections are output to uniquely named sections of this form	
−fdata−sections are output to uniquely named sections of this form		
.rodata.strS.A	Mergeable strings of size S and alignment A	
.rodata.cstA	Mergeable constant data of alignment A	
.gnu.linkonce.r.*	C++ ‘‘link-once’’ constant data	
.ctors	Pointers to C++ static constructors	
.dtors	Pointers to C++ static destructors	
.eh_frame_hdr		
C++ exception handling information		
.eh_frame		
.gcc_except_table		
.data	Variables >n bytes (compiled -Gn) with an initial value	Initialised data segment
.data.*	Large initialised variables compiled with −fdata−sections	
.gnu.linkonce.d.*	C++ ‘‘link-once’’ data	
.lit4	Constants (usually floating point) which the assembler decides to store in memory rather than in the instruction stream	Small initialised data segment
.lit8		
.sdata	Variables <=n bytes (compiled -Gn) with an initial value	
.sdata.*	Small variables compiled with −fdata−sections	
.gnu.linkonce.s.*	C++ ‘‘link-once’’ small data	
.sbss	Uninitialised variables <=n bytes (compiled -Gn)	Small zero-filled segment

Section name	What generates it	Where it ends up
.sbss.*	Small uninitialised variables compiled with −fdata−sections	
−fdata−sections		
.gnu.linkonce.sb.*	C++ ‘‘link-once’’ small uninitialised data	
.bss	Uninitialised larger variables	Zero-filled segment
.bss.*	Uninitialised variables compiled with −fdata−sections	
−fdata−sections.		
.gnu.linkonce.b.*	C++ ‘‘link-once’’ uninitialised data	
.debug*	DWARF debug information	Not in load image
.line		
.stab*	Stabs debug information	
.comment	#ident/.ident strings	
.gptab.*	Information section	
.reginfo	Information section	

Named ELF sections also exist to hold relocation records, symbol tables, etc, but they don’t show up in the final program at all.

11.3. Using Extra Sections

The compiler and assembler already generate a multitude of different object file sections which get linked together into (typically) three large output segments: read-only code & data, initialised data, and uninitialised (zero) data − as shown in Table 6 Standard ELF section names, above.

In some applications it may be necessary to define additional object code sections and segments which can be located at disjoint areas within the CPU’s address map. We’ll take as an example an M4K CPU core. This core has no cache, but a high-speed on-chip ISRAM (Instruction SRAM) at a fixed virtual address, say 0x0. With a CPU like this you would want to locate certain critical functions within the SPRAM region, but you would have to blow them into a PROM at a different address, which would then be copied to the ISRAM at run-time.

11.3.1. Assembler Section Definition

New sections are introduced to the assembler by the following directive:

```
.section name,"flags",@progbits[,align]
```

The section name can be any symbol, but by convention begins with a dot. The flags are a string of 0 to 4 characters selected from:

Table 16-2 Section attribute flags

Flag	Meaning
a	allocate address space
w	contains writable data
x	contains executable instructions
g	contains gp-accessible data

The optional final align parameter specifies the required section alignment, as a power-of-two. So, to introduce a code section intended for on-chip SRAM we could use the following:

```
.section .isram, "ax", @progbits, 2
```

After this initial definition has been seen by the assembler, you can then omit all but the section name, e.g.

```
‘‘.section .isram’’.
```

The assembler remembers the previous section (beware, it’s only a one-level stack!), and you can return to it using.previous directive.

11.3.2. C/C++ Section Definition

Segment switching in C or C++ is quite different; the compiler already has to keep track of sections and emits section directives as necessary. If you want to steer some particular piece of data or code into a particular named section, then GNU C provides an ‘‘attribute()’’ extension mechanism, for example:

```
 /* put variable foo into section .xdata */
  attribute  ((section (".xdata"))) int foo;

/* put function bar into section .isram */
  attribute  ((section (".isram")))
int bar ()
{
return 1;
}
```

But often what you want to do is collect a group of functions (or a group of data) into some special fixed area of the CPU memory map, so it may be more convenient to be able to decide which functions to group together at link time instead of compile time.

One way to do this is to give each function its own unique section name, and then generate a linker script which combines only the ones which we want into the hardware-significant segment. The compiler’s −ffunction−sections option outputs each function into a section whose name is straightforwardly based on the function’s name (i.e..text.fnname) − you can then manipulate the individual function sections at link time, and functions not assigned to a specific output segment will simply be merged into the global .text section.

See the −−function−ordering option in the [Gprof] manual for another way to order individual functions, based on profiling data

If the functions that you move in this way end up out of reach of the normal jal instruction (which is restricted to operating in a 256Mbyte ‘‘segment’’ of memory), then you will have to tell the compiler to use indirect jalr instructions to call these functions. See ‘Calling Remote Functions’ on page 41 for details of how to do this.

11.3.3. Linking Extra Sections

When using non-standard sections you’ll have to create your own linker script. For the ISRAM example discussed above you might expect to add something like the following lines to the default script:

```
.isram 0x0 :
{ *(.isram) }
```

These lines merge all the .isram input sections into the .isram output section, located at virtual address 0x0. The resulting executable module could then be converted into ASCII and downloaded by the board’s PROM monitor.

However, when creating a rommable program, your program will have to contain code to copy the .isram section from ROM to ISRAM itself. In this case your linker script might contain the following:

```
OVERLAY 0x0 : AT (0xbfc3c000)
{
.isram { *(.isram) }
}
```

The AT directive specifies that although the ‘‘overlay’’ is linked to be run at virtual address 0x0, it will be positioned at address 0xbfc3c000 in the load image (the load address). The load address in this case is the top

4KB of a 256KB boot PROM (base address 0xbfc00000). Your startup code must then copy the code from this known address into the ISRAM, e.g.

```
extern char  load start isram[];
extern char  load stop isram[];
/* copy from load address to run address */
isram_write (0x0,  load_start_isram,
  load_stop_isram -  load_start_isram);
```

If you have a number of C modules which contain code only intended for ISRAM (as described in the previous section), then you can name them explicitly in the script here, e.g.

```
OVERLAY 0x0 : AT (ALIGN( etext, 16) + ( edata -  fdata))
{
.isram {
*(.isram) c_isram1.o(.text) c_isram2.o(.text)
}
}
```

This example will include the .text section (i.e. code) from files c_isram1.o and c_isram2.o, and merge them into the output .isram section. The .text sections from all other object files listed on the linker command line will be handled in the normal way.

11.3.4. Linker Garbage Collection

Compiler command line options –ffunction-section and –fdata-sections can be used to create unique sections for functions and variables declared in a source file. With the --gc-sections option of the linker, unused functions and variables can be removed from the final executable image.

This process of linker ‘‘garbage collection’’ may require some manual intervention if there are sections of your code or data which are not explicitly referenced by your code, but are perhaps required by some external software, such as an operating system loader. In this case you will have to create a linker script, and mark those sections which must not be eliminated using the ‘‘KEEP’’ directive, for example ‘‘KEEP(*(.init))’’. See the linker manual [Ld] for more details.

Note that −gc−sections cannot be used when generating a relocatable output file, i.e. when using the linker’s −r flag.

11.3.5. Calling Remote Functions

Although data in additional sections can be accessed without any special precautions, care must be taken when calling functions in them. The MIPS call (jal) instruction can’t specify a full 32-bit target (MIPS instructions are only 32 bits long, and there has to be an opcode field to identify this instruction...); instead, it stores 28 bits of the target address; the high 4 bits of the target address are just those of the jal instruction. The effect is that you can only call a function in the same 256Mbyte ‘‘page’’ of memory; the linker will complain if you attempt to reach further.

There are ways around this problem:

- In C you can declare the remote function using the longcall or far attribute, e.g.:

  ```
  extern int far_away ()  attribute  ((longcall));
  ```

  or

  ```
  extern int far_away ()  attribute  ((far));
  ```

- In assembler you must explicitly take the address of the function before calling it, e.g.:

  ```
  la       t8,remfunc

  jalr     t8
  ```

- For C code where changing the source is not possible, you can compile with the −mlong−calls option. This forces the compiler to default to performing all function calls using the two-step la/jalr sequence. Note that this incurs a speed and space penalty, as ALL function calls will now require at least three instructions instead of one.

  To avoid the extra overhead when you know that certain functions can be reached with an absolute 28-bit address, you can mark such functions with the near attribute, e.g.

  ```
  extern int close_by ()  attribute  ((near));
  ```

12. MIPS Toolchain Run-time I/O System

12.1. Semihosting - Unified Hosting Interface (UHI)

A hosting interface provides a mechanism for user-mode applications to request services that require complex processing or knowledge of the current hardware. Such requests are either processed on the same target by a higher level OS (self-hosting or fully hosted) or offloaded to a remote host (semi-hosted).

The MIPS Unified Hosting Interface (UHI) aims to support both scenarios of self-hosting and semi-hosting without the need to neither modify the software when switching between the two nor require any special knowledge of the current hardware. The interface is layered to allow requests to propagate upwards through software stacks, boot monitors and debuggers.

MIPS UHI is based around the SYSCALL and SDBBP instructions along with a custom ABI and a reserved SYSCALL/SDBBP number. The SYSCALL instruction is used wherever there may be a higher level of software handling, the SDBBP instruction is used when a debugger is the only remaining option.Unified Hosting Interface libraries are included by referencing ‘uhi’ in the linker script. For example, your linker script could contain a GROUP statement like:

```
GROUP(-lc –luhi –lgcc)
```

The uhi*.ld scripts are example linker scripts which use UHI.

13. CPU Management

The second major component of the MIPS Toolchain run-time system consists of a set of support functions with which to 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 romable code.

13.1. 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_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 (for example, inserting software breakpoints, self-modifying code).

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.

13.2. TLB Maintenance

The functions listed below provide for initialization and maintenance of the CPU's memory management Translation Lookaside Buffer (TLB), if present.

The TLB and memory management definitions are supplied by including <*mips/cpu.h*>.

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.

13.3. 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.

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

Table 7 Register Access Intrinsic

*	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.
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.

13.3.1. 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.

Note: 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

(That is. *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 note 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.

13.3.2. 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).

13.3.3. 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).

13.3.4. 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.

13.3.5. 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)

Different thread context or VPE
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.

13.4. 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.

13.5. Floating Point Coprocessor (CP1)

The FPU is automatically enabled in the CRT for any hard-float application and there is therefore no need for libraries to enable or disable FP functionality.

There is no FPU emulator in the toolchain.

14. Booting MIPS

Toolchain Boot solutions

The Toolchain includes solutions for booting, included as part of the Hardware Abstraction Layer libraries. This boot solution can be optimized for performance with production code, or you can use the pre-built generic version for MIPS platforms.

See 'Boot Code' on page 78 for details.

Boot-MIPS

Also available for download is 'Boot-MIPS,' a demonstration of suitable bring-up code for MIPS-based cores. More information can be obtained from

https://community.imgtec.com/developers/mips/resources/application-notes/

Example code for Boot-MIPS is available from:

https://community.imgtec.com/downloads/boot-mips-example-boot-code-for-mips-cores-1-5-19/

15. MIPS Hardware Abstraction Layer

15.1. Introduction to HAL

This section describes the HAL in the MIPS toolchain. The HAL code forms the MIPS port of the libgloss component which is usually released alongside newlib.

15.1.1. Memory Addresses

All addresses referenced in this section relate to a 32-bit address space and are suitable for o32 or n32 ABIs. When working with the n64 ABI the addresses must be sign extended from bit-31. This often means prefixing with 0xFFFFFFFF.

15.1.2. Features

The low level support code is focussed on the features present in MIPS32, MIPS64 and MicroMIPS32 Release 2 and above including any applicable ASEs. This code is not intended to provide simplified access to all architectural features but it covers those which are commonly required to encourage re-use and stability.

15.2. Linker scripts

The HAL includes three standard linker scripts for the three supported MIPS ABIs. The HAL is fully compliant to the UHI specification and as such the linker scripts reflect the standard. The scripts are called uhi32.ld, uhi64_64.ld and uhi64_n32.ld for o32, n64 and n32 respectively and are structured similarly except for the necessary ABI differences.

The linker scripts will default to using '`_start`' as the entry point for an application. Otherwise, if the symbol '`__reset_vector`' is defined, the linker script will use 0xBFC00000 instead.

A secondary linker script, bootcode.ld is also included and is used to include the object code necessary for a bootable application. It must be used in conjunction with the other UHI linker scripts.

Symbols in the linker scripts can be overridden on command line using:

```
-Wl,--defsym,<symbol>=<value>
```

The mipshal.mk makefile fragment discussed in Section 15.3 simplifies the use of the bootcode.ld linker script.

15.3. HAL makefile fragment

A makefile fragment is included in the HAL to provide easy access to common MIPS build variants and HAL features. This can be found in `$MIPS_ELF_ROOT/share/mips/rules/mipshal.mk`.

The build variants that are accessible through the makefile fragment are listed below.

Symbol	Description
`ENDIAN`	Endian-ness of compiled programs.
`ABI`	ABI used by the compiler.
`MIPS_TOOLCHAIN`	Controls which toolchain is selected.
`ROMABLE`	Setting ROMABLE will utilize a secondary link script, "bootcode.ld" which will include all the necessary code to boot a MIPS system.
`ELF_ENTRY`	Used to define the entry point to the application. By default this is either _start or 0xbfc00000 when ROMABLE is set. This should only be defined by advanced users.

Their default options are possible configuration values are as follows:

Symbol	Default Value	Supported Values
ENDIAN	Compiler default.	“EL” (little endian) or “EB” (big endian)
ABI	Compiler default.	“32” (o32), “64” (N64), “n32” (N32)
MIPS_TOOLCHAIN	mips-mti-elf for o32 and mips-img-elf for n64	The triplet part of a toolchain, i.e. mips-mti-elf or mips-img-elf
ROMABLE	None	1
ELF_ENTRY	None	Any linker script expression

The mipshal.mk makefile fragment recognises the following symbols and will emit the required linker flags when these variables are defined.

mipshal.mk Symbol	Matching linker script symbol
MEMORY_BASE	__memory_base
MEMORY_SIZE	__memory_size
STACK	__stack
FLUSH_TO_ZERO	__flush_to_zero
FLASH_START	__flash_start
FLASH_APP_START	__flash_app_start
APP_START	__app_start
ISR_VEC_SPACE	__isr_vec_space
ISR_VECTOR_COUNT	__isr_vector_count
ENABLE_XPA	__enable_xpa

A sample makefile to set the memory size to 8M and leaving 1M for the bootloader would look like:

```
# Project name

# Project description, etc

# Optional specific memory size and start address
# MEMORY SIZE=8M
# APP_START=0x80100000

include ${MIPS ELF ROOT}/share/mips/rules/mipshal.mk

# CC, LD are set to the appropriate compiler, linker for MIPS TOOLCHAIN and
# MIPS_ELF_ROOT.  CFLAGS and LDFLAGS now contain the command flags to set
# the required ABI and any additional link time symbols to configure the
# linker scripts.

CFLAGS += ...  # Project specific compiler options
LDFLAGS += ... # Project specific linker options

< makefile targets, etc >
```

15.4. HAL Examples

The tool chain provides a number of examples of bootable code, interrupt handling code and code which makes use of various compiler and linker facilities. See 'Example Programs' page 31.

These examples and their documentation are installed in `$MIPS_ELF_ROOT/share/mips/examples`.

Example	Description
`custom_excpt`	An example of using _mips_handle_exception that traps memory reads and writes to first 4k of memory.
`fault_recovery`	Another example of _mips_handle_exception that shows how the struct gpctx can be manipulated.
`interrupt_handler`	An example showing the usage of interrupt attributes and mips_isr_sw0 to provide a handler for the software 0 interrupt.
`isr_vector_space`	An example showing the usage of ISR_VEC_SPACE to provide larger inline interrupt handlers.
`romable`	A basic bootable example suitable for placing in ROM, It includes copying application code to RAM during boot. This particular example will link in the generic library boot code.
`romable_minimal`	Similar to the romable example, this example instead focuses on producing a very small but generic bootable elf.
`romable_predef`	A more complex example which uses macros and the read only contents of configuration registers to produce a romable program that includes only the necessary code for that system. Two cores are supported: the P5600 and the I6400.

15.5. Exceptions

The toolchain provides a number of facilities for dealing with exceptions, from routines which save the CPU context to default handlers and reporting facilities.

The default exception handler can be overridden by defining the function `_mips_handle_exception`. This function is passed the CPU context at the point of the exception and an integer describing the type of exception. Its prototype can be found later in this section and in `<mips/hal.h>`.

The CPU context captures the register state, error state and relevant CP0 state at the point the exception occurred. When the exception has been serviced then the exception handler will restore the processor state using this structure, reloading registers with the corresponding values where applicable. This structure is defined in `<mips/hal.h>`.

The gpctx structure is laid out as follows:

Structure field	Description	Read/Write
r[31]	The value stored in the register (1 to 32) at which point the exception occurred. Index 0 is register 1, index 30 is register 31.	RW
epc	The program counter where processing resumes after exception has been serviced.	RW
badvaddr	Records the virtual address that caused a TLB exception or address error.	R
hi	Multiply and Divide register higher result. (Reserved in Release 6)	RW (pre Release 6) NA (Release 6)
lo	Multiply and Divide register low result. (Reserved in Release 6)	RW (pre Release 6) NA (Release 6)
link	Provides access to extended context structures relating to optional architecture state.	NA
status	Contains the operating mode, interrupt enable state and diagnostic states of the processor when the exception occurred.	R(W)
cause	Describes the cause of the exception.	R
badinstr	Records the instruction that caused the exception. This is only valid if the processor implements the CP0 BADINSTR register.	R
badpinstr	Records the branch prior to the current exception. This is only valid if the processor implements the CP0 BADPINSTR register and `cause.bd` is set.	R

Fields in this structure are reloaded back to the CPU when the exception has been serviced and normal execution resumed. Some fields are not reloaded as they correspond to read-only registers. Those fields which are marked RW can be updated and the CPU context when continuing normal execution will reflect those values. Fields marked R which have been written to will not have their value restored.

The status field will be restored by the standard exception handler but this is not guaranteed as an RTOS may manage the status register specially when overriding the handlers.

The contents of the “hi” and “lo” fields of this structure are defined only for MIPS for pre-Release 6, for Release 6 the fields are reserved.

Users should consult the MIPS Architecture Reference Manual Vol. III: MIPS64 / microMIPS64 Privileged Resource Architecture or MIPS Architecture For Programmers Volume III: The MIPS32 and

microMIPS32 Privileged Resource Architecture for further details on the *status*, *cause*, *badvaddr*, *badinstr*, *badpinstr* and *epc* values, as per the corresponding co-processor 0 registers.

The toolchain provides a facility to report exceptions through UHI semi hosting, giving the CPU context at the point where the exception occurred.

15.5.1. Extended Context

The basic exception handler saves only the general purpose CPU registers and some control registers when an exception occurs. For most exception handlers this is sufficient. In some cases it may be necessary or desirable to save and restore the contents of the floating point and MSA registers. Systems using XPA require additional support as the BadVAddr register (CP0 Register 8, Select 0) is wider than the provided exception Ihandlerhander expects.

Saving and restoring the DSP context is not currently supported.

The following table describes the “struct linkctx” which is common to all extended context structures:

Field	Description
`reg_t id`	Records the type of extended context saved. Currently used types are described below.
`struct linkctx * next`	The next context field or NULL if the last in the chain.

The values currently supported for the id field are defined in $MIPS_ELF_ROOT/include/mips/hal.h and share the common prefix LINKCTX_TYPE_. For reference:

Context type	Description
`LINKCTX_TYPE_MSA`	Context for MSA.
`LINKCTX_TYPE_FMSA`	Context for MSA with the floating point condition and status register.
`LINKCTX_TYPE_FP32`	Context for 32-bit floating point.
`LINKCTX_TYPE_FP64`	Context for 64-bit floating point.
`LINKCTX_TYPE_XPA`	Context for XPA.

To help manage multiple extended contexts, `struct gpctx` has a link field allowing the various contexts to be linked together. Additionally the following function is provided to simplify linking contexts together:

Function prototype	Description
`void _linkctx_append (struct gpctx *gp, struct linkctx *new)`	Appends new onto the end of the gp’s list of context structures.

The following functions are provided for saving and restoring these register sets.

Function prototype	Description
`reg_t _xpa_save (struct xpactx *ptr)`	Saves the XPA related context. Note: unlike the other extend context functions, no corresponding `_xpa_load` is provided as BadVAddr is a read-only register.
`reg_t _fpctx_save (struct fpctx *ptr)`	Saves the floating point context to ptr.
`reg_t _fpctx_load (struct fpctx *ptr)`	Restores the floating point context from ptr.
`reg_t _msactx_save (struct msactx *ptr)`	Saves the MSA context to ptr.

Function prototype	Description
`reg_t _msactx_load (struct msactx *ptr)`	Restores the MSA context from ptr.

The various save functions return the address of the context when no error occurs or zero if an error occurs. The load functions return the address given to them when the state is reloaded successfully or zero if an error occurs.

XPA Extended Context

When a MIPS32 core is operating with extended physical addressing enabled, the control register BadVAddr is extended by up to 32-bits. The default exception handler will not capture the upper 32-bits of BadVAddr. The `_xpa_save` function can be used to obtain the full value.

Function prototype	Description
`reg_t _xpa_save (struct xpactx *ptr)`	Saves the XPA related context. Note: unlike the other extend context functions, no corresponding _xpa_load is provided as BadVAddr is a read-only register.

As the BadVAddr is a read only register, there is no corresponding load function.

struct xpactx is laid out as follows:

Type and member name	Description
`struct linkctx link`	Type and link to next context.
`reg64_t badvaddr`	Records the virtual address that caused a TLB exception or address error.

Floating point Extended Context

The HAL provides support for saving and restoring the floating point register set for both 32-bit and 64-bit FPUs as detected by the FR bit in the status register.

The two different floating point contexts share a common member, struct fpctx:

Type and member name	Description
`struct linkctx link`	Type and link to next context.
`reg_t fcsr`	Floating point condition and status register
`reg_t reserved`	Reserved.

The 32-bit floating point context is laid out as follows:

Type and member name	Description
`struct fpctx fp`	As described above.

A union of:
Type and name
`double d[16]`
`float s[32]`
	Data from the 32 floating point registers.

Accessors for the elements are provided for use in C or C++:

Name	Description

Name	Description
`FP32CTX_DBL(struct fp32actx * ctx, int n)`	Returns the nth double element.
`FP32CTX_SGL(struct fp32actx * ctx, int n)`	Returns the nth float element.

The 64-bit floating point context is laid out as follows:

Type and member name	Description
`struct fpctx fp`	As described above.

A union of:
Type and name
`double d[32]`
`float s[64]`
	Data from the 32 floating point registers.

Accessors for the elements are provided for use in C or C++:

Name	Description
`FP64CTX_DBL(struct fp64actx * ctx, int n)`	Returns the nth double element.
`FP64CTX_SGL(struct fp64actx * ctx, int n)`	Returns the nth float element.

MSA Extended Context

The HAL provides support for saving and restoring the MSA register set with the following functions:

Function prototype	Description
`reg_t _msactx_save (struct msactx *ptr)`	Saves the MSA context in the pointed-to structure.
`reg_t _msactx_load (struct msactx *ptr)`	Restores the MSA context from the given fpctx structure.

The MSA context structure is laid out as follows:

Type and member name	Description
`struct linkctx link`	Stores link and type.
`reg_t fscr`	Floating point condition and status register.
`reg_t msacsr`	MSA condition and status register.

A union of:
Type and name
`_msareg w[32]`
`double d[64]`
`float s[128]`
	Data from the 32 MSA registers.

Accessors for the elements are provided for use in C or C++:

Name	Description
`MSAMSACTX_D(stuct msactx * ctx, int n)`	Returns the nth msa register.
`MSACTX_DBL(stuct msactx * ctx, int n)`	Returns the nth double element.
`MSACTX_SGL(stuct msactx * ctx, int n)`	Returns the nth float element.

15.5.2. Exception Handlers

The following symbols will be branched to by the vector code placed at EBASE to perform custom handling of exceptions and errors. The vector stubs will only clobber register k1 to perform the indirect branch to the symbol. The handling code must issue return from exception (ERET).

Function prototype	Description
`void _mips_tlb_refill ()`	Entry point for a tlb refill exception.
`void _mips_xtlb_refill ()`	Entry point for a xtlb refill exception.
`void _mips_cache_error ()`	Entry point for a cache error exception
`void _mips_general_exception ()`	Entry point for a general exception. The value of k1 is already saved in a gpctx structure immediately below the stack pointer.

Standard exception handling functions are provided by the HAL report unhandled exception events. All function can be overridden.

Function prototype	Language	Description
void _mips_general_exception ()	Assembly, non-standard ABI	A weak definition of `_mips_general_exception` is provided that is aliased to `__exception_save`
`void __exception_save ()`	Assembly, non-standard ABI	Performs full GP context save and calls `_mips_handle_exception`. Also restores the context if `_mips_handle_exception` returns and performs ERET.
`void _mips_handle_exception (struct gpctx *ctx, int exception)`	C – called without preserving FP register state.	A weak definition of `_mips_handle_exception` is provided that is aliased to `__exception_handle_verbose` via the linker scripts.
`void __exception_handle_verbose (struct gpctx *ctx, int exception)`	C	Handles all MIPS exceptions reporting the textual description of the cause using the 'write' function on file descriptor 1. Also performs indirection to convert UHI SYSCALL interface operations to use the UHI SDBBP interface. Floating point denormal operations are handled and flush to zero is enabled depending on the `__flush_to_zero` setting from the linker scripts. Finally any remaining unhandled exceptions are reported via the UHI exception operation and execution is optionally resumed if the UHI exception operation is successful, otherwise the __exit function is set as the exception return address with 0xff as the return value.

Function prototype	Language	Description
`void __exception_handle_quiet (struct gpctx *ctx, int exception)`	C	Identical to `__exception_handle_versbose` except that no textual output is generated.
`__exception_handle`	none	A symbol created by the linker scripts to alias whichever of `__exception_handle_quiet` or `__exception_handle_verbose` is included in the link. This symbol can be used to refer to the currently selected implementation.

An example using "_mips_handle_exception" can be found in `$MIPS_ELF_ROOT/share/mips/examples/fault_recovery`

15.6. Interrupts

The HAL supplies basic interrupt setup and handling facilities. These include initial setup of the exception base, interrupt vector spacing and interrupt mode. By default the toolchain will enable vectored interrupt mode with unique vector entry points.

Note: Interrupts are not enabled if there is an external interrupt controller (EIC) present or if vector spacing is set to non-zero and VINT in Config3 is false.

Interrupt entry points are designed to only clobber the value held in register K1. This leaves register K0 usable for persistent data within system level code.

15.6.1. Interrupt handlers

The HAL provides several layers of customization for handling interrupts. The first and simplest is a set of specific symbols which users can define to provide their own handling code without needing to worry about the setup or layout of the interrupt vector itself.

Interrupt handlers can be implemented by overriding the definition of `_mips_interrupt` for a general interrupt entry or `_mips_isr_<interrupt>` to handle a specific interrupt.

The HAL interrupt service routine will attempt to branch the corresponding `_mips_isr_<interrupt>` function. If those symbols are undefined, then the interrupt service routine will branch `_mips_interrupt` instead. If `_mips_interrupt` is not defined then the interrupt will infinitely loop. This code is located in `mips_excpt_isr.S`.

Functions for interrupt servicing

Function name	Description
`void _mips_interrupt ()`	Fallback handler when a specific handler is missing.
`void _mips_isr_sw[0-1] ()` `void _mips_isr_hw[0-5] ()`	Handlers for specific interrupts.

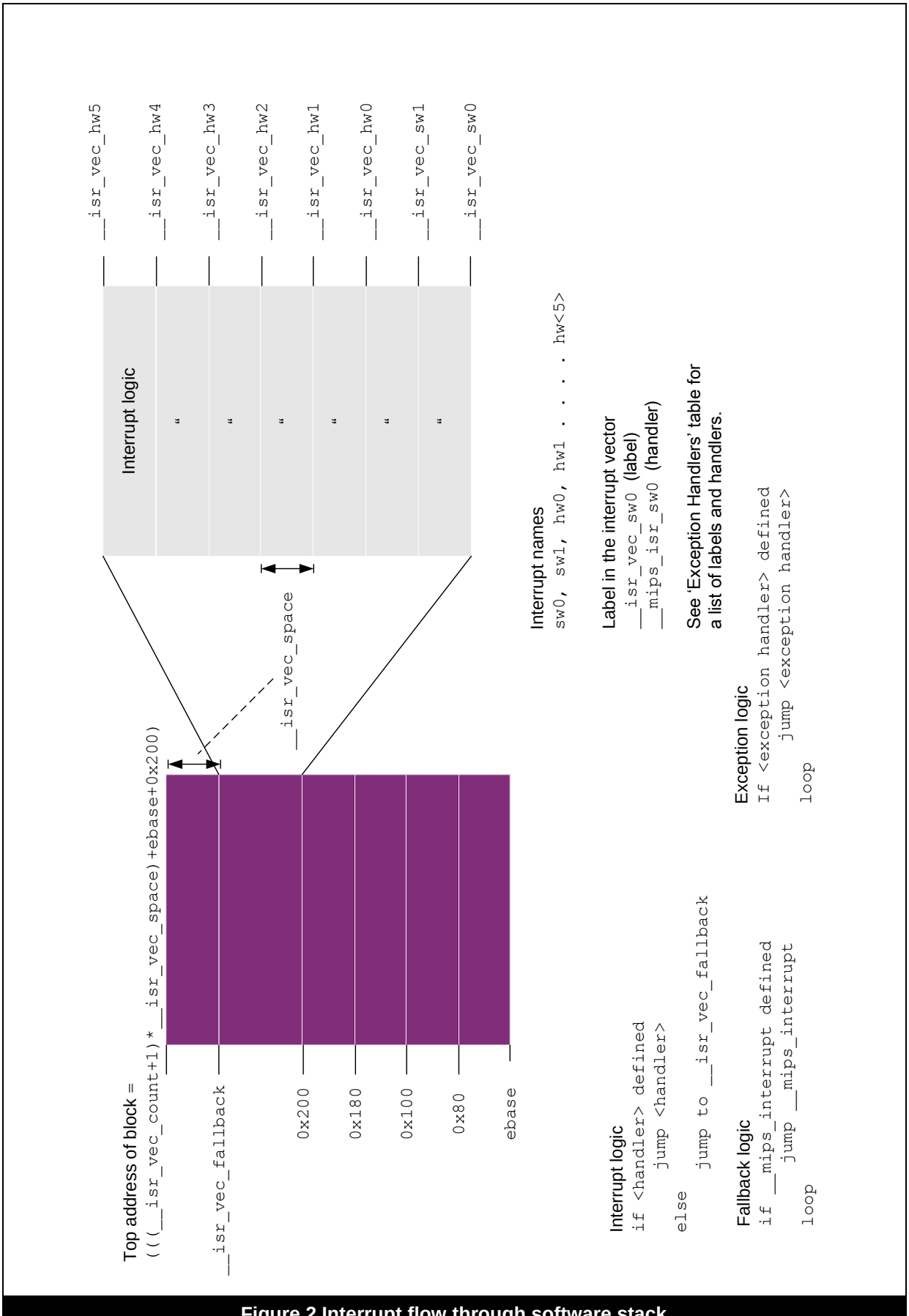

Figure 2 Interrupt flow through software stack

15.6.2. Custom interrupt vector

The HAL also supports users who wish to minimize code by supplying their own interrupt vector entry points and potentially implementing handlers directly in the vector. To create custom interrupt entry points create a module derived from `mips_excpt_isr.S` and include it in your link. If you require wider than default vector spacing then you must also set the vector spacing in the link script using the `__isr_vec_space` symbol. It is important that the module uses function sections and defines the whole vector as one function called `__isr_vec` this will allow the module to be correctly placed via the standard linker scripts.

An example demonstrating alternate vector spacing can be found in `$MIPS_ELF_ROOT/share/mips/examples/isr_vector_space`

15.6.3. Compiler support for interrupt handlers

GCC can be used to write interrupt handlers using the function attribute “interrupt”. Further attributes are available to give better control over how GCC sets the behaviour of the interrupt handler:

interrupt

An optional argument is supported for the interrupt attribute which allows the interrupt mode to be described. By default GCC assumes the external interrupt controller (EIC) mode is in use, this can be explicitly set using the argument `eic`. When interrupts are non-masked then the requested IPL is copied to the current IPL which has the effect of only enabling higher priority interrupts. To use vectored interrupt mode use the argument `vector=[sw0|sw1|hw0|hw1|hw2|hw3|hw4|hw5]`, this will change the behaviour of the non-masked interrupt support and GCC will arrange to mask all interrupts from sw0 up to and including the specified interrupt vector.

use_shadow_register_set

Assume that the handler uses a shadow register set, instead of the main general-purpose registers. An optional argument `intstack` is supported to indicate that the shadow register set contains a valid stack pointer.

keep_interrupts_masked

Keep interrupts masked for the whole function. Without this attribute, GCC tries to re-enable interrupts for as much of the function as it can.

use_debug_exception_return

Return using the deret instruction. Interrupt handlers that don’t have this attribute return using eret instead.

You can use any combination of these attributes, as shown below:

```
void   attribute   ((interrupt)) v0 ();
void   attribute   ((interrupt, use shadow register set)) v1 ();
void   attribute   ((interrupt, keep interrupts masked)) v2 ();
void __attribute__ ((interrupt, use_debug_exception_return)) v3 ();
void __attribute__ ((interrupt, use_shadow_register_set, keep_interrupts_masked)) v4 ();
void __attribute__ ((interrupt, use_shadow_register_set, use_debug_exception_return)) v5 ();
void   attribute   ((interrupt, keep interrupts masked, use debug exception return)) v6 ();
void   attribute   ((interrupt, use shadow register set, keep interrupts masked,
                                use debug exception return)) v7 ();
void __attribute__ ((interrupt("eic"))) v8 ();
void __attribute__ ((interrupt("vector=hw3"))) v9 ();
```

An example implementing _mips_isr_sw0 and _mips_interrupt via compiler generated handlers can be found in `$MIPS_ELF_ROOT/share/mips/examples/interrupt_handler`

A further example under `isr_vector_space` shows this in combination with a custom interrupt vector and an assembly coded handler.

15.7. Custom Exception Handlers

In order to implement custom exception handling an implementation of `_mips_handle_exception` should be provided by an application. This function should be written in C with the following prototype:

```
void _mips_handle_exception (struct gpctx *ctx, int exception)
```

The exception argument will be one of the exception codes listed in Appendix B, 'Exception Types' on page 85. Extreme care must be taken in this function to avoid unintended nested exceptions especially those caused by UHI semi-hosting operations. In particular, if application code does not need to process any custom SYSCALL operations then the function should begin with:

```
void  mips handle exception (struct gpctx *ctx, int exception)
{
  if (exception == EXC SYS)
  {
    /* Forward UHI SYSCALL operations to the standard handlers */
    __exception_handle (ctx, exception);
    return;
  }

  if (exception == ...)
  {
   <custom handlers>
    return;
  }

  __exception_handle (ctx, exception);
  return;
}
```

Debugging information can be placed in the handler but must only use the low level UHI semi-hosting functions like ‘write’ and ‘plog’ to output messages. Use of printf or other C library functions will result in corrupt application state depending on where the original exception occurred.

Register context at the point of the original exception can be found in the CTX structure and can be modified in place. This structure will be used to restore context on return from exception. The status field should not normally be changed and may not be restored if running inside an RTOS, consult your RTOS documentation if you need to modify the CP0 Status register during exception handling. The default configuration of the HAL will mean that no interrupts will be routed to the `_<mips_handle>_exception` function. An RTOS will hook the exception handling chain above `_<mips_handle>_exception` so some exceptions may never be seen by user code when run within an RTOS, again consult your RTOS documentation for details.

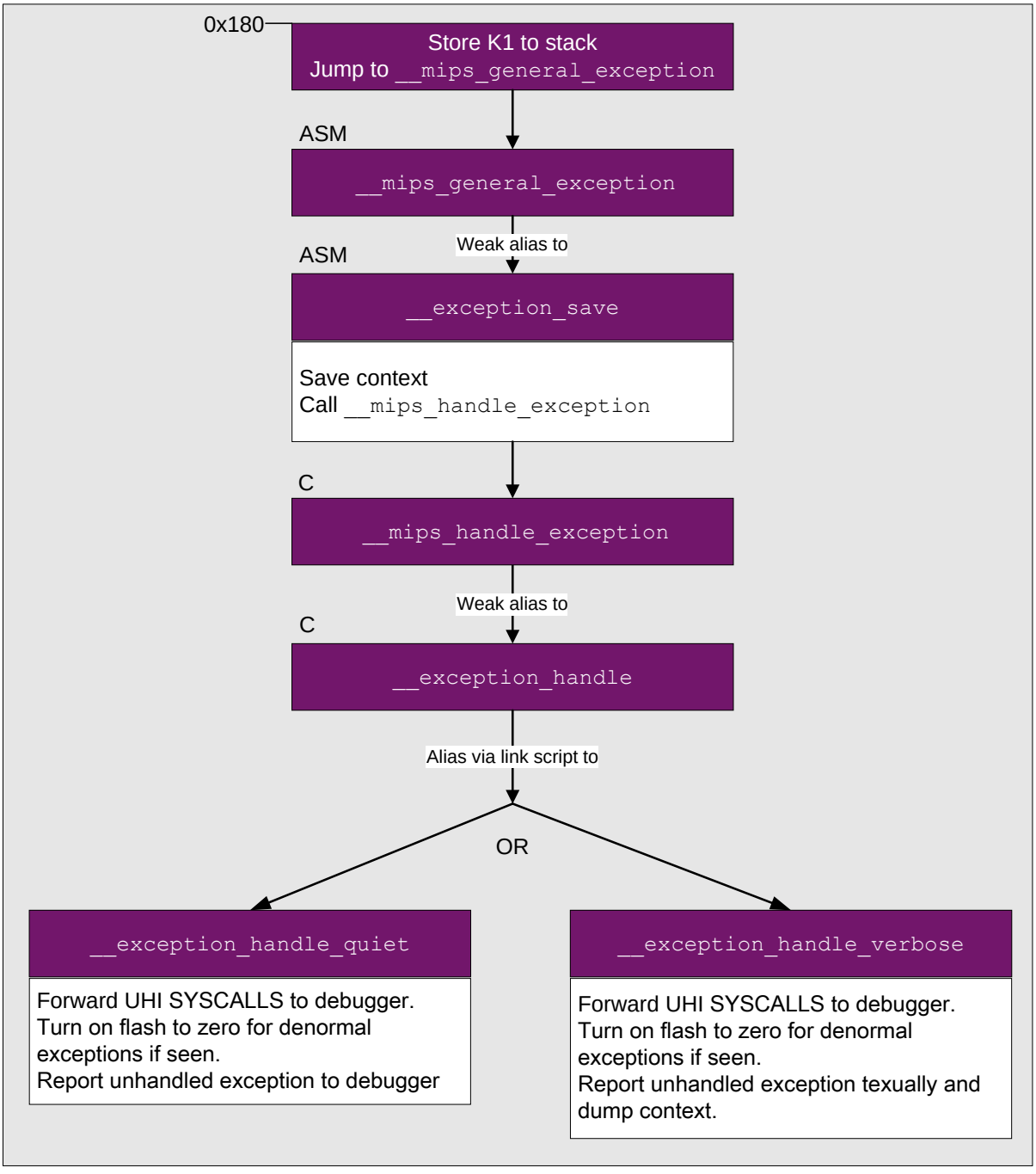

Figure 3 Normal exception handling

Custom semi-hosting handlers

Semi-hosting operations are normally forwarded to a debugger via SDBBP instructions or to a Boot Monitor if it has support for UHI. They can also be handled directly in user software if there is an application specific device or software structure for storing the data read or written by an application. To support this, all UHI semi-hosting operations will pass through `the _mips_handle_exception` function and can be hooked.

The sequence of operators to handle the 'write' function is given below as an example:

call write -> SYSCALL 1 -> - `_mips_handle_exception` -> `__exception_handle` -> `__uhi_indirect` -> SDBBP 1

A custom handler must therefore intercept a SYSCALL exception and handle a subset of UHI operations instead of forwarding them to `__exception_handle` as above. The SYSCALL instruction may be used for multiple purposes but UHI guarantees that the operand to SYSCALL will be '1' for UHI operations and also that $2 (v0) will be set to '1'. It is recommended that any other usage of SYSCALL also sets $2 to a unique value excluding '1' to make disambiguation in the handler easier. Reading the instruction that caused an exception is possible and depending on hardware support then it may be stored in the badinstr field of the CTX; note that code must check for CP0 BADINSTR support before relying on this field.

The code below relies on $2 being able to distinguish UHI SYSCALLs from any other SYSCALLs:

```
#include <mips/hal.h>
#include <mips/uhi syscalls.h>

void _mips_handle_exception (struct gpctx *ctx, int exception)
{
  if (exception == EXC SYS
      && ctx->v[0] == 1)
  {
    switch (ctx->t[9])
    {
    case   MIPS UHI WRITE:
      /* ctx->a[0] contains the file descriptor (1==stdout, 2==stderr, other=whatever)
         ctx->a[1] contains a char* buffer
         ctx->a[2] contains the size of the buffer */
      if (ctx->a[0] == 1)
      {
        <handle it>
        ctx->v[0] = <number of bytes written or -1 for error>;
        ctx->v[1] = <errno value on error>;
        return;
      }
      break;
    }
  }

  __exception_handle (ctx, exception);
  return;
}
```

Figure 4 Example custom UHI handler

When hooking semi-hosting operations care must be taken to hook related operations to ensure consistent handling. For example the hook above for 'write' using the stdout file descriptor should also include 'open', 'close', 'fstat' to indicate that stdout is available otherwise the other operations will be sent to a debugger and may well be reported as unsupported. With sufficient UHI operations hooked it is possible to become a self-hosted application and never make requests to a debugger and therefore not require a JTAG connection. As a minimum this means that stdin, stdout and stderr file descriptors must be hooked for all file operations as the C library initialisation code will call things like fstat automatically.

15.8. Application layout

A standard set of symbols are defined in the UHI linker scripts that can be used to control and locate various features such as the placement of code and the location of the exception base.

The set of symbols is as follows:

Symbol	Default value	Description
`__stack`	`0`	Default location of the top of stack. When set to zero the stack will be placed at the end of the memory returned by `_get_ram_range`
`__memory_size`	`3M`	Size of memory returned by `_get_ram_range`. When set to zero then the UHI `get_ram_range` operation is used to locate RAM instead. (The 3M value originates from a historic setting of 1M of memory starting at a base address of 0x80200000)
`__memory_base`	`0x80000000`	Base of memory returned by `_get_ram_range`.
`__app_start`	`0x80200000`	Base address for application code, leaving 2M of space for a bootloader.
`__flash_start`	`__app_start`	Base flash address which is automatically set if including boot code. The toolchain will copy application code from flash to RAM at runtime before the application starts.
`__flash_app_start`	`__flash_start + sizeof (boot code)`	Base flash address for application code. This is automatically set up to tightly pack boot code and application code in flash. To place application code without needing to copy it from ROM to RAM then set `__flash_app_start` to the same value as `__app_start`.
`__excpt_ebase`	`__app_start`	Location of the base of exception vectors. This symbol cannot be overridden.
`__isr_vector_space`	`32 (o32, n32), 64 (n64)`	Default interrupt service routine vector spacing.
`__isr_vector_count`	`9`	Number of vector entries. 9 are provided, 8 for the interrupt vectors and a 9th as a fallback mechanism. This value can be increased to reserve additional `__isr_vector_space` sized slots for interrupt vectors in EIC mode.

The HAL includes two functions that interact with the layout of an application in order to provide dynamically allocated memory support:

Function prototype	Operation
`void __get_ram_range (void ** base, void ** extent)`	Gives the base and end of the largest region of RAM. This is used internally in the support code to place the heap and stack. It uses the definition of `__memory_size` and `__memory_base`. If `memory_size` is zero then it will use the `get_ram_info` UHI operation. *base is set to the first address of RAM and *extent is set to the last address+1 of RAM.
`char* sbrk (int nbytes)`	Allocate memory from RAM. This function is used by C libraries to implement malloc and should not be called directly if C library features are also used. Memory will be allocated within the region reported by `__get_ram_range` but will also account for whether the application has been placed within the reported memory. If the application is inside the range then memory will be allocated starting from the end of the application data section as indicated by the _end symbol.

15.9. Semihosting functions

The following standard semi-hosting functions are provided for users and are implemented using the MIPS Unified Hosting Interface.

Function prototype	Description
`int close (int fd)`	Close a file descriptor, -1 on error, 0 on success.
`int fstat (int32_t file, struct stat *sbuf)`	Get file statistics for an already opened file descriptor.
`int link (const char *oldname, const char *newname)`	Rename a file, returns 0 on success or -1 on failure.
`off_t lseek (int fd, off_t offset, int whence)`	Seek with an open file handle.
`int open (const char* filename, int flags, mode_t mode)`	Open a file descriptor, -1 on error. Positive on success.
`ssize_t pread (int fd, void *buf, size_t count, off_t offset)`	File read from a given offset, returns number of bytes read or -1 on error.
`ssize_t pwrite (int fd, const void *buf, size_t count, off_t offset)`	Write to a file at a given offset.
`int read (int fd, void *buffer, size_t len)`	Read from an open file descriptor, returns number of bytes read or -1 on an error.
`int stat (const char *filename, struct stat *buf)`	Get file statistics.
`int unlink (const char *file)`	Unlink a file.
`int write (int fd, const void *buf, size_t count)`	Write to an open file descriptor, returns number of bytes read or -1 on error.

15.9.1. Specialist UHI operations

Some additional UHI specific operations are supported via non-standard support functions. The plog operation is designed as a very lightweight debug function to report status to a debugger.

Function prototype	Description
`int32_t __plog (int8_t *fmt, int32_t num)`	Print information with up to 1 integer in the format string using the plog UHI operation.
`void __exit (int exitcode)`	Emergency exit operation. This is a very low level exit that will either simply loop or return to a bootloader if supported.

15.10. Configuration settings

Several aspects of the standard support code can be configured by setting link time symbols which the code responds to.

Symbol name	Default value	Description
`__use_excpt_boot`	1	Controls whether UHI operation handling should make use of the exception handler installed by a boot monitor. See below for supported values
`__flush_to_zero`	1	Set if subnormal floating point values should be flushed to zero in hardware for FPU and MSA operations.

`__use_excpt_boot` can have one of three values:

Value	Meaning
0	Do not use exception handler present in boot.
1	Use exception handler present in boot if Status[BEV] is 0 at startup.
2	Always use exception handler present in boot.

15.11. Cache management

The HAL provides numerous functions for operating on CPU primary, secondary and tertiary level caches. These functions are accessible though the `<mips/cpu.h>` header file.

Note: Cache initialization is provided only within the boot code.

Function prototype	Description
`void mips_size_cache ()`	Sets the following uint32_t global variables: `mips[ids]cache_size`, the size in bytes of the primary instruction, primary data and secondary cache. `mips[ids]cache_linesize`, the size in bytes of each line in the primary instruction, primary data and secondary cache. `mips[ids]cache_ways`, the number of ways of the primary instruction, primary data and secondary cache. This is called automatically by the other cache operations.
`void mips_sync_icache (vaddr_t va, size_t n)`	Synchronizes the contents of the instruction cache with the data cache when the instruction stream is modified.
`void mips_clean_cache (vaddr_t va, size_t n)`	Write back and invalidate cache entries matching the given address range from all caches.
`void mips_clean_dcache (vaddr_t va, size_t n)`	Write back and invalidate cache entries matching the given address range from all data caches, separate instruction caches are unchanged.
`void mips_clean_icache (vaddr_t va, size_t n)`	Write back and invalidate cache entries matching the given address range from all instruction and joint caches, separate data caches are unchanged.
`void mips_clean_dcache_nowrite (vaddr_t va, unsigned int size)`	Invalidate with no write back a virtual address range in all data caches. Only safe if region is totally cache line aligned.
`void mips_flush_cache ()`	Write back and invalidate all cache entries from all caches.
`void mips_flush_dcache ()`	Write back and invalidate all cache entries from all data caches.
`void mips_flush_icache ()`	Write back and invalidate all cache entries from instruction and joint caches.
`void mips_lock_icache (vaddr_t va, size_t n)`	Lock a given range to an instruction cache if supported.
`void mips_lock_dcache (vaddr_t va, size_t n)`	Lock a given range to a data cache if supported.
`void mips_lock_scache (vaddr_t va, size_t n)`	Lock a given range to a secondary cache if supported.

The locks on caches lines will be cleared with the associated flush operations. Self-modifying code will require some variant of `mips_sync_icache`, `mips_clean_icache`, `mips_flush_icache` or `mips_flush_cache` to ensure that the instruction stream reflects the changes made.

15.11.1. CM3 - L2 cache

The HAL provides support for the L2 cache provided by Coherency Manager Version 3. As the CM3 is an optional feature the code to support this is not included by default. Attempts to use the default cache management code or boot code when L2 cache is external to the core will cause a UHI boot exception (L2 configuration is external to the core).

The HAL code for the CM3 is part of the supplied 'libcm3.a' library, add this to the link using -lcm3 when targeting a core with CM3.

15.12. CP0 support macros/functions

Co-processor 0 provides control registers to configure interrupts, exceptions, TLBs and caches. The HAL provides a number of functions to simplify access to these registers and are provided by the `<mips/cpu.h>` header file.

Users should consult the MIPS Architecture Reference Manual Vol. III: MIPS64 / microMIPS64 Privileged Resource Architecture or MIPS Architecture For Programmers Volume III: The MIPS32 and microMIPS32 Privileged Resource Architecture for further details on the registers referenced here.

The HAL provides simple get/set/exchange operations and bit wise set, clear, set and clear operations.

Some control registers are 64-bit on MIPS64 and some remain 32-bit. The MIPS32 functions must be used to access any 32-bit control register on MIPS64 architectures.

Note: These functions are not atomic with respect to interrupts or exceptions.

15.12.1. MIPS32 registers

A general set of functions are provided to access header defined CP0 registers. These functions take a register name to access any co-processor 0 register. The header file <mips/m32c0.h> defines these macros and textual names for many of the architectural specified registers, see Appendix A for a full list.

Function	Description
`mips32_get_c0 (<RegName>)`	Returns the value of `<RegName>`.
`mips32_set_c0(<RegName>, val)`	Set `<RegName>`.
`mips32_xch_c0(<RegName>, val)`	Exchange the old value of `<RegName>`with `val` and return the old value.
`mips32_bc_c0(<RegName>, clr)`	Clear the corresponding bits in `<RegName>` that are set in `clr`.
`mips32_bs_c0(<RegName>, set)`	Set the corresponding bits in `<RegName>` that are set in `set`.
`mips32_bcs_c0(<RegName>, clr, set)`	Set and clear corresponding bits in `<RegName>` using `set` and `clr` masks.

A helper macro "`MIPS_C0_REGNAME`" must be used to specify `<RegName>` for register and select combinations that are implementation dependant, e.g. Performance Counter 3, for the above functions.

Macro prototype	Description
`opaque MIPS_C0_REGNAME(int register_number, int select)`	Returns an implementation defined value for use with `mips32_<operation>_c0` and `mips64_<operation>_c0` macros.

For example:

```
// Read performance counter 3
count = mips32_get_c0(MIPS_C0_REGNAME(25, 7));

// Alternatively in a header file after the inclusion of <mips/m32c0.h>
#define C0_PERFCNT3 MIPS_C0_REGNAME(25, 7)
```

15.12.2. MIPS64 registers

The MIPS64 architecture changes the width of some of the registers that are used in the macros described in Section 15.12.1. To support their usage on MIPS64, additional macros are supplied in `<mips/m64c0.h>`.

Function	Description
`mips64_get_c0 (<RegName>)`	Returns the value of `<register>`
`mips64_set_c0(<RegName>, val)`	Set `<register>`
`mips64_xch_c0(<RegName>, val)`	Exchange the old value of `<register>` with the parameter and return the old value
`mips64_bc_c0(<RegName>, clr)`	Clear the corresponding bits in `<register>` that are set in `clr`.
`mips64_bs_c0(<RegName>, set)`	Set the corresponding bits in `<register>` that are set in `set`
`mips64_bcs_c0(<RegName>, clr, set)`	Set and clear corresponding bits in `<register>` using `set` and `clr` masks.

The header file `<mips/m64c0.h>` defines these macros and the helper macro MIPS_C0_REGNAME.

15.13. TLB support

The HAL provides support for the initialization and maintenance of the CPU's memory management Translation Lookaside Buffer (TLB). For 32-bit TLBs, the memory management functions are supplied by including <mips/m32tlb.h>. For 64-bit TLBs found on either MIPS64 or MIPS32 with XPA support enabled, the functions are supplied by including <mips/m64tlb.h>.

Users should consult relevant Privileged Resource Architecture document and the Software User's Manual for their processor when using the TLB as some aspects of TLB operation are implementation specific.

15.13.1. 32-bit Physical Memory

The 32-bit API is described below:

Function prototype	Description
`void mips_init_tlb (void)`	Initialises and invalidates the whole TLB.
`unsigned int mips_tlb_size (void)`	Returns the number of entries in the TLB.
`void mips_tlbinval (tlbhi_r hi)`	Probes the TLB for an entry matching hi, and if present invalidates it.
`void mips_tlbinvalall (void)`	Invalidate the whole TLB.
`void mips_tlbri2 (tlbhi_t *phi, tlblo_t *plo0, tlblo_t *plo1, unsigned int *pmsk, int index)`	Reads the TLB entry with specified `byindex`,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 int pmsk, 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 int pmsk)`	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 int *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 int pmsk)`	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.

`tlbhi_t, tlblo_t` and `pmsk` are 32-bits wide and their usage for the relevant EntryHi, EntryLo0 and EntryLo1 and PageMask fields should be consulted in the MIPS Architecture For Programmers Volume III: The MIPS32 and microMIPS32 Privileged Resource Architecture. Users will also need to implement the TLB refill exception handler and general exception handlers for TLB refill events if using virtual memory management features.

15.13.2. 64-bit Physical Memory

The 64-bit API is described below:

Function prototype	Description
`unsigned int m64_tlb_size (void)`	Returns the number of entries in the TLB.
`void m64_tlbinval (tlbhi64_r hi)`	Probes the TLB for an entry matchinghi,and if present invalidates it.
`void m64_tlbinvalall (void)`	Invalidate the whole TLB.
`void m64_tlbri2 (tlbhi64_t *phi, tlblo64_t *plo0, tlblo64_t *plo1, unsigned long long *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 m64_tlbwi2 (tlbhi64_t hi, tlblo64_t lo0, tlblo64_t lo1, unsigned long long pmsk, int index)`	Writes `hi, lo0, lo1` and `msk` into the TLB entry specified by index.
`void m64_tlbwr2 (tlbhi64_t hi, tlblo64_t lo0, tlblo64_t lo1, unsigned long long pmsk)`	Writes `hi, lo0, lo1` and `msk` into the TLB entry specified by the Random register.
`int m64_tlbprobe2 (tlbhi64_t hi, tlblo64_t *plo0, tlblo64_t *plo1, unsigned long long *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 m64_tlbrwr2 (tlbhi64_t hi, tlblo64_t lo0, tlblo64_t lo1, unsigned long long pmsk)`	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.

`tlbhi64_t, tlblo64_t` and `pmsk` (pagemask) are 64-bits wide and their usage with `EntryHi, EntryLo0` and `EntryLo1` and `PageMask` fields should be consulted in the MIPS Architecture For Programmers Volume III: The MIPS64 and microMIPS64 Privileged Resource Architecture and the MIPS Architecture For Programmers Volume III: The MIPS32 and microMIPS32 Privileged Resource Architecture for using this API with XPA.

Users will also need to implement the xTLB, TLB refill exception handlers and general exception handlers for TLB refill events if they make use of virtual memory management features.

15.13.3. FTLB set stride

The FTLB can be efficiently initialized by using the TLBINVF instruction. Invalidating one way in a set will invalidate the entire set. The structure of the FTLB in terms of which index bits represent ways and sets is not architecturally defined so striding across the sets is implementation specific. The code by default is configured to assume that the sets are indexed from the first bit after the VTLB index but if the first bits represent ways instead then a custom stride must be applied. If, for example, there were 4 sets then the stride would be 0x4. The stride is configured using the `__tlb_stride_length` linker script symbol.

15.13.4. XPA

When using this API with XPA on MIPS32, users should take note that the field layouts for `tlblo64_t` should conform to the implementation-specified layouts given in MIPS32 and microMIPS32 Privileged Resource Architecture and the relevant processor specific documentation.

XPA will not be enabled by default even when it is found to be present. Instead it must be explicitly enabled by defining a linker script symbol `__enable_xpa` to 1. This can be done by passing `ENABLE_XPA=1` to a makefile using `mipshal.mk`.

15.14. Boot Code

Two variations of MIPS Boot code are included; one is intended to work on all possible implementations and one is tuneable to reduce code size and increase performance when preparing production code. Some aspects of boot time initialisation can't be done without core specific information, where this can be detected then an error will be reported unless appropriate code has been provided. Boot code makes use of two special sections:

reset

Only one object in the entire link should contain a .reset section and it must implement the reset vector. It is recommended to place a non-weak symbol called `__reset_vector` at the start of the module so that the linker will detect if multiple such modules are included.

boot

Used for any other boot-time code referenced from the reset vector. This code is placed immediately after the .reset section and this memory should be usable from cold boot.

15.14.1. Pre-built boot code

The pre-built version of the boot code inspects hardware configuration at runtime and initialises any features that are found. It will also try to detect situations where additional core specific support is required for successful initialisation and halt the processor reporting what is missing. This code can be included in a program by including the secondary link script bootcode.ld

See example `romable` in `$MIPS_ELF_ROOT/share/mips/examples/`

15.14.2. Optimized boot code

The HAL provides a parameterized set of boot code in source form. This version does not inspect the hardware configuration at run time, instead the necessary initialization code is selected and configured at compile time. To do this users supply the values of the various configuration registers when building the custom version. These values can be extracted from a core using a debugger; it does not matter what state a core is in when obtaining the values as only the read-only bits are relevant. The registers include: Config1 to Config5, WatchHI, and the additional watch registers in CP0..

The following register values must be defined:

`C0_CONFIG1_VALUE, C0_CONFIG2_VALUE, C0_CONFIG3_VALUE`

If Config4 or Config5 is implemented then their values are also required:

`C0_CONFIG4_VALUE, C0_CONFIG5_VALUE`

If watch registers are implemented then all the WatchHI register values must be defined:

`C0_WATCHHI_VALUE, C0_WATCHHI1_VALUE …`

If a Coherency Manager version 3 is implemented then the CMGCRBase and GCR L2 Config value must be defined:

`C0_CMGCRBASE_VALUE, GCR_L2_CONFIG_VALUE`

See `romable_predef` in `$MIPS_ELF_ROOT/share/mips/examples/` for an example solution for the P5600 and I6400 cores.

15.14.3. Boot overrides

The boot code can detect when externally configured L2 cache is attached and will report an error if there is no custom initialisation code provided. To provide a custom implementation of L2 cache initialisation then create a module with a function called `__init_l23cache` and ensure that it is included in the link. The module should have its text in a section called .boot by defining the `_BOOTCODE` macro prior to including `<mips/asm.h>`. This function has no arguments and should

not reference the SP or GP registers as these will not be configured. Also, the callee-saved registers are reserved for persistent storage throughout the boot process and should not be clobbered.

A general boot hook is provided that will be called after all basic initialisation has happened. To install the hook create a module with a function called `__boot_init_hook` and ensure it is included in the link. This module must also have its code in a section called .boot as above. `__boot_init_hook` is called before the application code is copied to RAM, permitting additional hardware/software initialization. The same rules apply to the implementation of `__boot_init_hook` as the `__init_l23cache` function above.

Function prototype	Description
`void __boot_init_hook()`	Optional extra initialization function executed after cache initialization.

15.14.4. Application handover

The boot code includes support for copying application code and data out of ROM into its runtime address. This is performed immediately after the `__boot_init_hook` function is called. See the section on Custom Exception Handlers for details of how to control placement. After copying code and data the boot code sets up the exception return address (EPC) to point at the `_start` symbol and performs an ERET to begin execution. The code at `_start` is responsible for setting up the stack pointer, global pointer (if required), and any CP0 settings required by the application.

15.15. Optimizing for size

By default the standard linker scripts include and enable all possible features and are as informative as possible about any runtime errors. Many of these features will not be required in production code and may have to be removed in order to meet size constraints.

In general it is necessary to create a custom version of the standard linker scripts released with the HAL to fine tune the features and size of the support code. (Note: the default linker scripts may be updated with new versions of the HAL so tracking the changes in a derived version will be important.)

The HAL provides an example of a bootable application with all optional support code removed. This is achieved through a custom linker script which applies the changes listed in the following sections.

See example `romable_minimal` in `$MIPS_ELF_ROOT/share/mips/example/`

15.15.1. Boot context

The code which enables applications to interact with the boot context (i.e. a boot monitor) is generally only required if running from within a boot monitor such as YAMON or U-boot. This code can be safely removed if such platforms are not important.

In a copy of a HAL linker script, update the value of `__use_excpt_boot` to be 0 instead of 1 and remove the following line:

```
EXTERN (__register_excpt_boot);
```

15.15.2. Dynamic argument handling

The code to support UHI compliant argument passing where argc has value -1 is all contained within the `__getargs` function. This can safely be removed and the CRT code will create a null argument array instead.

In a copy of a HAL linker script, remove the following line:

```
EXTERN (__getargs);
```

15.15.3. Quieter exception reporting

The textual output from the default exception handler occupies 1KB of data space as well as over 1KB of code. This verbose output can be removed and instead any runtime exception will only be reported via the UHI exception operation which, if unhandled by software, will be reported to a debugger.

In order to use the quieter exception reporting take a copy of a HAL linker script and replace the following line:

```
EXTERN (__exception_handle_verbose);
EXTERN (__exception_handle_quiet);
```

15.16. Argument handling

The HAL has support for argument handling that is compliant with the UHI specification. This interface is implemented by the _start application entry point.

$4 is the control register for determining how arguments are passed.

Value	Meaning
0	All other registers are ignored. A null argument vector is created for main with `argc == 0, argv[0] == 0 *envp == 0`.
Any non-zero positive value.	$4 contains `argc`, $5 contains `argv`, $6 contains `envp`. `Envp` contains a null terminated of `char*` pointers with each string being in the format key=value.
-1	The UHI interface is used to obtain the command line. The memory for the `argv` array is allocated from the start of the application stack.
All other negative values	Unknown interface, the same as `$4 == 0`.

15.17. Miscellaneous support functions

A number of additional support functions are provided that may be used by advanced users who override large portions of the standard HAL code. These should not be referenced by casual users.

Function prototype	Description
`int __uhi_exception (struct gp_ctx *ctx)`	Report an unhandled exception with the specified context. This will use the UHI syscall interface.
`int __uhi_indirect (struct gpctx * ctx)`	Forward a UHI SYSCALL to the SDBBP interface.
`__getargs ()`	Obtains arguments using the UHI argument operations, allocating space from the stack. This function uses a non-standard calling convention.
`int __register_excpt_handler (void* RA, int SR)`	Sets up exception handling and optionally calls `__register_excpt_boot`

15.18. Boot monitor interoperability

The HAL supports the optional parts of the UHI specification for interoperability with boot monitors. This includes the ability to return back to a boot monitor upon completion of an application and also making use of UHI handling present in the monitor. The HAL follows an assumption described in the UHI specification that if started with Status.BEV clear then a boot monitor is present with UHI operation support in its exception handlers. The `__use_excpt_boot` setting can be used to override this assumption should it not hold true.

The support code for this can be found in `mips_excpt_boot.S` and the functions provided are briefly described below. None of these functions are expected to be invoked or modified by ordinary users but advanced users who configure additional CP0 registers may wish to provide custom versions of these routines to preserve and restore more context.

Function prototype	Description
`int __get_startup_BEV ()`	Obtains the value of the BEV bit from CP0 Status at the point of starting the application
`void* __register_excpt_boot (void * RA, int SR, void* cookie)`	Saves initial boot state. This function is called from a context which is representative of the original state at the point of starting the application. It stores all important state to enable `__return_to_boot` to restore the caller's context before exiting the application. The cookie is returned to the caller. Note: There is no stack available when this function is called.
`void __chain_uhi_excpt (struct gpctx* ctx)`	Restores the context from `ctx` and indirect jumps to the general exception vector entry point of the boot time (pre-application) exception vector. `$3` is clobbered to achieve the indirect branch which is compliant with the UHI specification.
`void __return_to_boot (int exitcode)`	If the UHI exit operation returns then return back to caller of `_start` with the specified exit code.
`void __convert_argv_pointers (void)`	Invoked with a non-standard ABI to convert argument pointers passed to a 32-bit application from a 64-bit environment. Memory is allocated from the stack to hold a new `argv` array and the `envp` array is replaced with a null pointer.

Appendix A. CP0 register name macros

Macro	Register name	Description
C0_INDEX	Index register	Index into the TLB array
C0_RANDOM	Random Register	Randomly generated index into the TLB array
C0_ENTRYLO0	EntryLo0	Low-order portion of the TLB entry for even-numbered virtual pages
C0_ENTRYLO1	EntryLo1	Low-order portion of the TLB entry for odd-numbered virtual pages
C0_GLOBAL	Global	Global number register
C0_CONTEXT	Context	Pointer to page table entry in memory
C0_CONTEXTCONF	ContextConfig	Context register configuration
C0_USERLOCAL	UserLocal	User information that can be written by privileged software and read via RDHWR register 29. If the processor implements the MIPS® MT Module, this is a per-TC register.
C0_XCONTEXTCONF	XContextConfig	XContext register configuration
C0_PAGEMASK	PageMask	Control for variable page size in TLB entries
C0_PAGEGRAIN	PageGrain	Control for small page support
C0_SEGCTL0	SegCtl0	Programmable Control for Segments 0 & 1
C0_SEGCTL1	SegCtl1	Programmable Control for Segments 2 & 3
C0_SEGCTL2	SegCtl2	Programmable Control for Segments 4 & 5
C0_PWBASE	PWBase	Page Table Base Address for Hardware Page Walker
C0_PWFIELD	PWField	Bit indices of pointers for Hardware Page Walker
C0_PWSIZE	PWSize	Size of pointers for Hardware Page Walker
C0_WIRED	Wired	Controls the number of fixed (“wired”) TLB entries
C0_PWCTL	PWCtl	HW Page Walker Control
C0_HWRENA	HWREna	Enables access via the RDHWR instruction to selected hardware registers
C0_BADVADDR	BadVAddr	Reports the address for the most recent address-related exception
C0_BADINSTR	BadInstr	BadInstr Reports the instruction which caused the most recent exception.
C0_BADINSTRP	BadInstrP	Reports the branch instruction if a delay slot caused the most recent exception.
C0_COUNT	Count	Processor cycle count
C0_ENTRYHI	EntryHi	High-order portion of the TLB entry.
C0_COMPARE	Compare	Timer interrupt control.
C0_STATUS	Status	Processor status and control
C0_INTCTL	IntCtl	Interrupt system status and control
C0_SRSCTL	SRSCtl	Shadow register set status and control
C0_SRSMAP	SRSMap	Shadow set IPL mapping
C0_CAUSE	Cause	Cause of last general exception

Macro	Register name	Description
C0_NESTEDEXC	NestedExc	Nested exception Support - EXL, ERL values at current exception
C0_EPC	EPC	Program counter at last exception
C0_NEPC	NestedEPC	Nested exception Support - Program Counter at current exception
C0_PRID	PRId	Processor identification and revision
C0_EBASE	EBase	EBase Exception vector base register
C0_CDMMBASE	CDMMBase	Common Device Memory Map Base Register.
C0_CMGCRBASE	CMGCRBase	Coherency Manager Global Control Register Base register
C0_CONFIG	Config	This maps to Config0
C0_CONFIG0	Config	Configuration register
C0_CONFIG1	Config1	Configuration register 1
C0_CONFIG2	Config2	Configuration register 2
C0_CONFIG3	Config3	Configuration register 3
C0_CONFIG4	Config4	Configuration register 4
C0_CONFIG5	Config5	Configuration register 5
C0_LLADDR	LLAddr	Load linked address
C0_MAAR	MAAR	Memory accessibility attribute register
C0_MAARI	MAARI	Memory accessibility attribute register index
C0_WATCHLO	WatchLo	Watchpoint address
C0_WATCHHI	WatchHi	Watchpoint control
C0_XCONTEXT	XContext	Extended Addressing Page Table Context
C0_DEBUG	Debug	EJTAG Debug register
C0_DEBUG2	Debug2	EJTAG Debug2 register
C0_DEPC	DEPC	Program counter at last EJTAG debug exception
C0_PERFCNT	PerfCnt	Performance counter interface
C0_ERRCTL	ErrCtl	Parity/ECC error control and status
C0_CACHEERR	CacheErr	Cache parity error control and status
C0_TAGLO	TagLo	Low-order portion of cache tag interface
C0_ITAGLO		Cache-tag interface, implementation specific (instruction low).
C0_DTAGLO		Cache-tag interface, implementation specific (data low).
C0_TAGLO2		Cache-tag interface, implementation specific
C0_DATALO		Cache-tag interface, implementation specific (data low).
C0_IDATALO		Cache-tag interface, implementation specific (data low).
C0_DDATALO		Cache-tag interface, implementation specific (data low).
C0_DATALO2		Cache-tag interface, implementation specific (data low).
C0_TAGHI	TagHi	High-order portion of cache tag interface

Macro	Register name	Description
C0_ITAGHI		Cache-tag interface, implementation specific (instruction high).
C0_DTAGHI		Cache-tag interface, implementation specific (data high).
C0_DATAHI		Cache-tag interface, implementation specific.
C0_IDATAHI		Cache-tag interface, implementation specific.
C0_DDATAHI		Cache-tag interface, implementation specific (data high).
C0_ERRPC	ErrorEPC	Program counter at last error
C0_DESAVE	DESAVE	EJTAG debug exception save register
C0_KSCRATCH1	KScratch1	Scratch Registers for Kernel Mode
C0_KSCRATCH2	KScratch2	Scratch Registers for Kernel Mode
C0_KSCRATCH3	KScratch3	Scratch Registers for Kernel Mode
C0_KSCRATCH4	KScratch4	Scratch Registers for Kernel Mode
C0_KSCRATCH5	KScratch5	Scratch Registers for Kernel Mode
C0_KSCRATCH6	KScratch6	Scratch Registers for Kernel Mode

Appendix B. Exception Types

Exception define	Exception number	Type
#define EXC_INTR	0	/* interrupt */
#define EXC_MOD	1	/* tlb modification */
#define EXC_TLBL	2	/* tlb miss (load/i-fetch) */
#define EXC_TLBS	3	/* tlb miss (store) */
#define EXC_ADEL	4	/* address error (load/i-fetch) */
#define EXC_ADES	5	/* address error (store) */
#define EXC_IBE	6	/* bus error (i-fetch) */
#define EXC_DBE	7	/* data bus error (load/store) */
#define EXC_SYS	8	/* system call */
#define EXC_BP	9	/* breakpoint */
#define EXC_RI	10	/* reserved instruction */
#define EXC_CPU	11	/* coprocessor unusable */
#define EXC_OVF	12	/* integer overflow */
#define EXC_TRAP	13	/* trap exception */
#define EXC_MSAFPE	14	/* MSA floating point exception */
#define EXC_FPE	15	/* floating point exception */
#define EXC_IS1	16	/* implementation-specific 1 */
#define EXC_IS2	17	/* implementation-specific 2 */
#define EXC_C2E	18	/* coprocessor 2 exception */
#define EXC_TLBRI	19	/* TLB read inhibit */
#define EXC_TLBXI	20	/* TLB execute inhibit */
#define EXC_MSAU	21	/* MSA unusable exception */
#define EXC_MDMX	22	/* mdmx unusable */
#define EXC_WATCH	23	/* watchpoint */
#define EXC_MCHECK	24	/* machine check */
#define EXC_THREAD	25	/* thread */
#define EXC_DSPU	26	/* dsp unusable */

Appendix C. Imagination Technologies Free to Use

Appendix D. Small C Library Reference

D.1. Introduction

This appendix discusses design principles of size-optimized variants of the C standard library, referred to as SmallCLib which provided as part of the MIPS bare metal toolchain and also serves as a technical reference.

SmallCLib is derived from Newlib v2.0 with small influences from Musl v0.9.14. This appendix enumerates all functional differences in the library as compared with Newlib 2.0.

It is intended to be used in conjunction with Newlib reference documentation provided with the MIPS Bare Metal toolchain under <toolchain-root>/share/doc or available on-line at https://sourceware.org/newlib

Note: SmallCLib is an optional component of Codescape GCC Tools for MIPS.

D.2. SmallCLib

The goal of SmallCLib is to provide as much functionality as possible in a small amount of space, and it is intended primarily for embedded use. The standalone library containing size optimized versions of functions is usable from the MIPS bare metal toolchain.

The SmallCLib comes in two variants:

D.2.1. ISO conforming

The ISO C-99 conforming version of SmallCLib, referred to as the small variant, does not omit functionality that's required by the standards in order to achieve a smaller library. Most of the space saving in this version comes from aggressive refactoring of the code to eliminate redundancy and some of the space saving comes at the cost of performance. This version provides an IEEE 754 compliant software floating point Math library.

D.2.2. Non-conforming

This version, referred to as the tiny-variant, omits some ISO features in order to achieve higher code density as compared to the compliant version. Following sections describe differences between ISO conforming and non-conforming versions.

File and standard IO

The file and standard IO functions provided by this version operate only on standard streams (stdin, stdout and stderr). All the supported streams are un-buffered.

Locales

Locale support in this version is limited to UTF-8 encoding only, which allows direct code paths and better performance. The only locale supported by this version is the default C.UTF-8 locale.

Floating point support

The floating point support in this version differs from IEEE 754 standard in the following ways,

- NaN, INF and de-normal input values are not handled. Operations involving such inputs generate unpredictable results.
- Sign of zero is ignored
- No IEEE exceptions are flagged

Reentrancy

This version of SmallCLib does not support reentrancy.

D.2.3. Integration with the GCC-based toolchain

Small C library support in GCC is provided via new multi-lib configurations. Pre-built small/tiny libraries variants are provided with the toolchain for select multi-lib combinations. In order to see the multi-lib configurations available with your MIPS toolchain, invoke gcc with the option '–print-multi-lib'.

The Small C library is provided within the toolchain as a set of three static library files, for both small and tiny variants. At the least, the toolchain installation will contain the files mentioned below:

File	Location in toolchain-root	Description
libc.a	mips-mti-elf/lib/small	ISO conforming size optimized string, memory, IO, time and other functions.
libm.a	mips-mti-elf/lib/small	ISO conforming size optimized Math functions.
libg.a	mips-mti-elf/lib/small	ISO conforming size optimized string, memory, IO, time and other functions, compiled with debugging enabled
libc.a	mips-mti-elf/lib/tiny	Non-conforming size optimized string, memory, IO, time and other functions
libm.a	mips-mti-elf/lib/tiny	Non-conforming size optimized Math functions.
libg.a	mips-mti-elf/lib/small	Non-conforming size optimized string, memory, IO, time and other functions, compiled with debugging enabled

Based on the actual combinations of multilib that the toolchain is built with, there may be multiple versions of these libraries – each having been compiled using a unique combination of compiler options and located within a directory structure determined by the corresponding multi-lib configuration. The libraries listed above correspond to big-endian, hard-float build for mips32r2 architecture.

D.2.4. Linking with and using the small C Library

Selection of standard C library is controlled by a new command-line option to the compiler driver

```
gcc -mclib=[newlib|small|tiny] ...
```

If no -mclib option is specified, the default newlib implementation will be used. At compilation stage, mclib option automatically defines macros necessary for correct linkage. At the link stage, this option controls the library search path so that the desired version of the standard C library gets linked against. If specified, this option must be consistent across both compile and link stages.

Note: If mclib=[small|tiny] is specified in conjunction with an unsupported combination of other multi-lib options, the compiler will link with the default C library(newlib), unless an alternate library search path is explicitly supplied.

Example application

A variant of the simple hello world application is available in <toolchain-root>/share/examples/helloclib. This example allows the user to build with any one of the 3 C-library variants and prints the sizes of the resulting binary for comparison.

Command to build this example is:

```
# make [MIPS_ELF_ROOT=path] [LITEND=1] [CLIB=[newlib|small|tiny]]
```

D.3. Specification Differences with Newlib

The following sections provide details about specific function behaviour. See 'Table 1 Feature comparison between libraries' on page 13 for a summary of the differences between newlib and the size-optimized variants.

D.3.1. Character handling and localization

Newlib and the small variant support the default “C”, “POSIX”, “C-JIS", "C-SJIS", "C-EUCJP" and “C-KOI8-R” locales with full wide and multi-byte functionality. The support for locale in tiny variant is limited to default C locale only.

D.3.2. Math functions

Changes in the small-variant of the library are oriented towards better compliance with the ISO C standard in places where Newlib implementation differs from the standard. Key differences relate to the setting of *errno* and correct generation of floating point exceptions. In general, the small-variant tries to do the minimum as required by the ISO C standard – behaviour that is not required or optional under the ISO standard is curtailed to favour a smaller code size.

The tiny-variant departs from IEEE754 and ISO C99 standards by removing support for special values such as de-normalized numbers, infinity and NaN inputs. Special values are correctly supported only by the predicate functions necessary to detect them, so code that uses the math-library can catch these input conditions before invoking library functions. Error-handling based on errno is kept true to the standard, subject to the exclusion of special values, mentioned earlier. Exception generation is entirely disregarded – the library may or may not generate exceptions in any given operation – the only predictable mode of use is with all floating-point exceptions disabled.

For elementary trigonometric functions(sin/cos/tan), the tiny-variant differs from Newlib in terms of arithmetic correctness – it is accurate only for a limited range of inputs. See details below.

expm1 – exponential minus 1

```
#include <math.h>
double expm1 (double x);
float expm1 (float x);
```

Error-handling for large x

ISO standard requires double-precision expm1 to signal a range-error for large values of x. Newlib does not set the errno, whereas both small and tiny variants of the C-library set errno to ERANGE.

fma – fused multiply add

```
#include <math.h>
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
```

Arithmetic accuracy

Double precision variant provided by Newlib performs a non-fused multiply-add. Both small and tiny variants of the C library provide the accuracy of fused operation by performing the intermediate multiplication in parts. Resultant accuracy is close to 80-bit extended precision operation, with very few ULP errors.

The single precision function behaves the same way for all library variants, by approximating fused operation using non-fused operation in double precision.

remquo, remquof -- remainder and quotient

Synopsis

```
#include <math.h>
double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
```

Boundary condition: y=0

The implementation may either set errno to signal a domain error or return 0.0 for this condition, as per Section 7.12.10.3 of ISO-C99. Newlib implementation returns NaN and does not set errno. Both small and tiny variants of the C library set errno to EDOM and return NaN.

sin, cos, sinf, cosf -- sine or cosine

Synopsis

```
#include <math.h>
double sin (double x);
double cos (double x);
float sinf (float x);
float cosf (float x);
```

Arithmetic accuracy (only for tiny variant)

The tiny variant uses an alternate range-reduction strategy which limits accuracy for large arguments. For the double precision function, accuracy varies with absolute input range as follows:

- For |x| < 253 : equal to or better than newlib implementation
- For 253 <= |x| < 264 : occasionally worse than newlib, but relatively correct.
- For |x| > 2^{64} : completely unreliable

For single precision function, accuracy varies with absolute input range as follows:

- For |x| < 2^{23} : equal to or better than newlib implementation
- For 223 <= |x| < 232 : occasionally worse than newlib, but relatively correct.
- For |x| > 232 : completely unreliable

This variant is significantly smaller and faster than both the small variant and the default newlib implementation.

Inexact exception for small x (< 2^{-28})

Newlib implementation generates an inexact exception for small values of x. Both small and tiny variants of the C library do not generate this exception. It is implementation defined as per Appendix F.9, point 8 of ISO-C99.

tan, tanf -- tangent

Synopsis

```
#include <math.h>
double tan (double x);
float tanf (float x);
```

Arithmetic accuracy (only for tiny variant)

Same as sin/cos

Inexact exception for small x (< 2^{-28})

Same as sin/cos

nextafter, nextafterf

Synopsis

```
#include <math.h>
double nextafter(double x, double y);
float nextafterf(float x, float y);
```

Arithmetic accuracy

The nextafter functions return y if x equals y, as per Section 7.12.11.3 of ISO-C99 standard. Newlib implementation returns x. Both small and tiny variants of the C library return y.

D.3.3. Formatted IO functions

Following two macros are available for restricted version of formatted IO. These macros can be used along with small or tiny version of SmallCLib.

Integer and character only

__mips_fio_int__ : This macro can be used to disable the support for floating point and wide character format specifiers. It enables the use of integer and character format specifiers (d, i , u, o, n, p, x, X, c, s).

Floating point, Integer and character only

__mips_fio_float__ : This macro can be used to enable the support for floating point format specifier (only %f). It enables the use of integer, character and floating point format specifiers (d, i , u, o, n, p, x, X, c, s and f).

The default provides support for all format specifiers.