MIPS® Toolchain Specifics

Document Number: MD00624
Revision 01.05
July 01, 2008

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

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

Table of Contents

List of Tables

Chapter 1

Introduction

1.1 About this Document

Whenever “SDE” is mentioned, this refers to the SDE Library that is supplied with the bare-metal target/ELF toolchain. Such text do not pertain to the linux target toolchain.

Throughout this document, the command prefix “mips-sde-elf-” is used (assuming that your target is bare-metal/ELF), for example `mips-sde-elf-gdb`. If your target is actually linux, the command prefix would actually be “mips-linux-gnu-”.

Throughout this document, the environment variable `$SDETOP` represents the parent directory of the kit library and the SDE examples, most likely `<installation_top_directory>/mips-sde-elf`.

Items listed in square brackets, like [Sweet99], are references to other documents - see the References Appendix for the full description of these other documents.

1.2 Contents

This document deals with options that are specific to MIPS architecture targets when using the GNU tools such as gcc, gdb, etc.

Chapter 2

Compiler Options

The “MIPS Options” section in the GCC manual lists those compiler options which are specific to MIPS-based processors. This chapter provides some additional information about these options, and how you might use them.

NOTE: Throughout this document, the command prefix “mips-sde-elf-” is used (assuming that you’re using the bare-metal/ELF), for example `mips-sde-elf-gdb`. If your target is actually linux, the command prefix would actually be “mips-linux-gnu-”.

2.1 Architecture Flags

There are several flags which adjust the class of instructions generated by the compiler or assembler to match your particular CPU type. You can get more information about the architectural features and choices mentioned here in [Sweet99].

2.1.1 Endianness Flags

The most fundamental architectural switch controls whether to generate big-endian or little-endian code. MIPS architecture processors may be configured either way, but the rest of the hardware usually determines which way your system must work. Software has to be compiled to match the way the CPU is configured, or it will fail every time you perform a sub-word load or store.

It is possible to write bi-endian code by very careful assembler coding (e.g. by performing all data accesses as aligned word transfers), but this is likely to be required for only the first few instructions after a hardware reset, until you have configured the CPU and/or device endianness correctly.

`-EB` Generate code and data for a big-endian CPU.

`-EL` Generate code and data for a little-endian CPU.

2.1.2 Instruction Set Flags

The compiler supports all official and currently implemented 32- and 64-bit MIPS instruction set architectures (ISAs). But the compiler will only generate code compatible with the base MIPS34 ISA unless one of the following switches is used:

`-mips1`

Issue instructions from the original MIPS I ISA. Compiler/assembler only - no libraries are provided.

`-mips2`

Issue instructions from the mips II ISA (branch likely; square root; 64-bit floating point load/store; faster floating point truncate). Compiler/assembler only - no libraries are provided.

-mips3

Issue instructions from the MIPS III ISA (64-bit instructions; 32 f.p. registers). See Section 2.6 “64-bit Support” 64-bits for more information. Compiler/assembler only - no libraries are provided.

-mips4

Issue instructions from the MIPS 64 ISA (floating point multiply-add/sub, indexed addressing, reciprocal, etc.). Compiler/assembler only - no libraries are provided.

-mips32

The new, rationalised, 32-bit MIPS34 instruction set defined by MIPS Technologies in 1998/99. It's not really very different from **-mips2**, but it picks up the conditional move instructions and rationalises the integer multiply/accumulate instructions (which were formerly CPU-specific). The "branch likely" instructions are officially deprecated in MIPS34, but the compiler will still generate them when tuning for CPUs for which it knows they don't have an adverse performance impact.

-mips32r2:

The Release 2 update to MIPS32 adds a few new instructions.

-mips64

MIPS Technologies' rationalised 64-bit MIPS64 instruction set, which is a superset of both **-mips4** (at the user level) and **mips32**.

An update of the specifications added some useful new features to the MIPS34 ISAs in September 2002. Many of these features are for the OS only; but there are also a few new user-level instructions:

- Bit-rotate: previous MIPS ISAs had only shifts. The compiler will make use of the hardware rotate instruction if your source code is written so as to perform the rotate in a single expression. For example:

```
unsigned int a, b, r;
/* fixed rotate right by 8, or left by 24 */
b = (a >> 8) | (a << 24);
/* variable rotate right */
b = (a >> r) | (a << (32 - r));
/* variable rotate left */
b = (a << r) | (a >> (32 - r));
```

- Bit-field operations: single-instruction unsigned bitfield extract and insert instructions make for more efficiency when doing just that. Note that gcc treats bitfields as signed if you don't use an explicit unsigned type modifier - use the**-funsigned-bitfields** option to change that behaviour. The compiler will sometimes use them when given simple and obvious mask and shift expressions. In cases where it doesn't you can use the explicit insert/extract intrinsics.

- Byte-swap instructions: the new instructions `wsbh`, `dsbh` and `dshd` swap bytes within halfwords, or halfwords within doublewords, in a register. So you can do a full 32-bit or 64-bit byte-swap in just two instructions. The compiler will not generate these instructions automatically, but you can access them via intrinsics.

- Sign-extend instructions: bytes and 16-bit values can already be sign-extended automatically when loaded from memory; these new instructions improve code for data which is already in registers.

- 64-bit FPU: a MIPS 32 Release 2 CPU may be paired with a 64-bit FPU, and the extra 16 registers will be used by the compiler if you give it the **-mfp64** option.

Once you've defined your base instruction set, there are a collection of "instruction set extensions" which you can enable:

-mips16

Compile using the MIPS16 "ASE". Each MIPS16 instruction is only 16 bits in size, and although a compiler must use more MIPS16 instructions to compile a function than would be required with the MIPS34 ISA, it allows simple integer code to be compiled with a 30-40% saving in space.

Use of this option is a decision with lots of consequences: see longer discussion inSection 2.7 “MIPS16® ASE Support” mips16-ase below.

Warning: although the name "MIPS16" seems to fit in with "MIPS34" and "MIPS64", it really is something quite different. In fact, MIPS16 encodings are available for 64-bit instructions too.

The MIPS16 ASE is not available on all CPUs. It also isn't possible to write a complete system using MIPS16 instructions, since some vital instructions (CPU control, floating point, etc) have no MIPS16 encoding.

MIPS16 instructions will probably only ever be generated by compiled code, so you will only ever see assembler code when looking at disassemblies or compiler intermediate files. In assembler source files you'll see that assembler code must request generation of MIPS16 code using an explicit `.set mips16` or `.set mips16e` directive; the command line option is not passed to the assembler by the `mips-sde-elf-gcc` front end.

The MIPS16 ASE is only available on bare-metal/ELF targets. It is not available for the GNU/Linux target.

-mips16e

The MIPS16 ASE is an extension to the MIPS16 encodings, built on the basis of experience with some large programs and achieving a useful improvement in density with a few extra instructions. This variant is standard on MIPS34 CPUs; in fact, the combination of flags "`-mips32-mips16`" implies **-mips16e**.

The MIPS16e ASE is only available on bare-metal/ELF targets. It is not available for the GNU/Linux target.

-msmartmips

This option is only valid if you've selected a MIPS34/MIPS64 instruction set, and SmartMIPS cores always implement MIPS16 too. It allows the toolchain to exploit the SmartMIPS extensions to the base MIPS34 ISA: in particular the indexed load (used with grateful thanks by the compiler) and enhanced multiplier instructions - the latter available only through assembler code or special C intrinsics.

SmartMIPS CPUs also anticipate the bit-rotate instruction from MIPS 32 Release 2, as in **-mips32r2** above.

-mpaired-single

For the MIPS 32 Release 2 and MIPS64 ISAs only, where 32 x 64-bit floating point registers are enabled (i.e. **-mfp64**), this flag enables use of the "paired single" SIMD floating point extension which provides instructions to do two single-precision (32-bit) floating point operations at once, keeping the operands in pairs within a 64-bit register. More details on this option can be found in the [Gcc] manual.

-mips3d

Enables the MIPS 3D ASE which includes additional paired-single instructions that are designed to improve the performance of 3D graphics operations. Implies **-mpaired-single**.

-mdsp

This option is only valid if you've also selected the MIPS 32 Release 2 instruction set. It tells the compiler to allow the use of the MIPS DSP ASE either automatically where possible, by using vector types, and by use of builtin intrinsics. The ASE is also enabled if the **-march=option** specifies one of these CPUs from the 24ke/ 34k/74k family.

-mno-dsp

Prevents the compiler from generating MIPS DSP instructions, even if the selected CPU architecture would support it.

-mmt

This option is only valid when you've selected the MIPS32 Release 2 instruction set. It has no direct effect on the compiler, but instructs the assembler to allow the MIPS MT ASE instructions. These instructions can be generated from C code by using the intrinsics.

A CPU which supports a given ISA will happily run code compiled for the previous variants with which it's backwardly compatible:

In practice the first criteria for choosing which level to go for is whether you want to use 64-bit integer data types, which are available only with **-mips64**.

Once you've chosen the integer data width, you'll get small performance increments by choosing the most specialised (usually highest-numbered) instruction set which matches your CPU; you'll make your binary program more portable by using the lowest number.

The MIPS32 instruction set (or its Release 2 variant) is usually best for applications which don't use 64-bit integer variables, and which don't use floating point heavily - even when you've got a 64-bit processor, since you don't waste data cache space storing unnecessary sign extensions.

2.1.3 CPU Flags

The target CPU type may be specified using the compiler's **-mtune=** option. This allows the compiler to optimize the scheduling of instructions to match your CPU's pipeline. If it is not specified, then the compiler picks the most generic CPU type which matches your requested instruction set (e.g. 4Kc® for **-mips32**), but this may generate sub-optimal code for faster CPUs.

Specifying the CPU type also allows the compiler to make more intelligent choices about CPU-specific features, such as the optional presence of fast or slow multipliers.

In addition to this, the **-march=** option may be used to specify the precise set of instructions and features provided by the target CPU. It also selects the pipeline scheduling parameters if **-mtune=** is not used explicitly. For compati-

bility reasons the current SDE makefiles (provided with the bare-metal/ELF version of the tools) do not make use **-march=**, but use **-mips32**, **-mips32r2**, etc, to select the base ISA - this is likely to change in a future release.

Table 2.1 List of -mtune= Names

-mtune=	-mips			Comments
	32	32r2	64	
4km, 4kc	X			32-bit synthesisable 4Kc and 4Km cores, with fast multiplier
4kp	X			32-bit synthesisable 4Kp core, with slow multiplier
4kem, 4kec	X	X		32-bit synthesisable 4KEc and 4KEm cores, with fast multiplier
4kep, m4k	X	X		32-bit synthesisable 4KEp and M4K cores, with slow multiplier
5kc, 5kf	X		X	64-bit synthesisable 5K core family; the 5Kf core has an FPU
20kc	X		X	64-bit 20Kc hard core
24kc, 24kf2_1, 24kf1_1	X	X		32-bit synthesisable 24K core family; the 24kf2_1 option tunes for a 64-bit FPU running at half the integer pipeline frequency while the 24kf1_1 tunes for a FPU running at the same frequencey as the integer pipeline.
24kec, 24kef2_1, 24kef1_1,	X	X		Enhanced version of the 24Kc and 24Kf cores, with additional features such as the MIPS DSP ASE. The 24kef2_1 option tunes for a 64-bit FPU running at half the integer pipeline frequency while the 24kef1_1 tunes for a FPU running at the same frequencey as the integer pipeline.
34kc, 34kf2_1,34kf1_1	X	X		32-bit synthesisable 34K core family, which supports the MIPS MT and MIPS DSP ASEs. The 24kf2_1 option tunes for a 64-bit FPU running at half the integer pipeline frequency while the 34kf1_1 tunes for a FPU running at the same frequencey as the integer pipeline.
74kc, 74kf2_1, 74kf1_1	X	X		32-bit synthesisable, superscalar 74K core family, which supports revision 2 of the MIPS DSP ASE. The 74kf2_1 option tunes for a 64-bit FPU running at half the integer pipeline frequency while the 74kf1_1 tunes for a FPU running at the same frequencey as the integer pipeline.

2.1.3.1 Other CPU-specific Options

You can control some features at a still finer level where necessary:

-mbranch-likely

Enable “branch likely” instructions with **-mips32** and **-mips64**, even though these instructions are officially deprecated.

-mno-branch-likely

Don't use "branch likely" instructions.

-mcheck-zero-division

Generate code to check for integer divide overflow - range checking is disabled by default.

-mno-check-zero-division

Don't generate code to check for integer divide by zero - checking is the default, except with **-mips16**.

-mhard-float

Emit hardware floating point instructions - this is the default.

-msoft-float

Emit calls to a software floating point emulation library.

-mno-float

This option is treated by the compiler's code generator as equivalent to **-msoft-float**, i.e. any use of floating point will generate calls to emulation functions, however it also instructs *mips-sde-elf-gcc* to link your program with libraries which do not include those emulation functions (thus causing a linker warning if they are called) and which also omit all hidden floating point support code, such handling of floating point format codes in `printf()` and `scanf()`. This option is only available with the bare-metal/ELF toolchain.

-mfp64

Emit hardware floating point instructions for a 64-bit FPU - this is the default for 64-bit ISAs, but can also be used in conjunction with **-mips32r2**, which allows a 64-bit FPU to be paired with a 32-bit integer ALU.

2.2 Optimization Options

The "Optimize Options" section in the GCC manual lists the various optimization techniques that are available; serious users should read that. But it's traditional to provide numeric options - the higher the number, the more optimization. You should never compile without at least **-O** (equivalent to **-O1**) unless you're debugging; GNU C's unoptimized code is really unoptimized. Serious application code will be compiled with at least **-O2**; higher numbers may make your code bigger, and the trade-offs are discussed below.

With GCC each number adds more optimization techniques, while keeping all the options from the lower numbers. For a detailed list of which optimizations are enabled at each optimization level see the [Gcc] manual, but in summary:

-O0

Do not optimize.

-O, -O1

Optimize by trying to reduce code size and execution time, but without performing any optimizations that take a great deal of compilation time.

-O2

Performs nearly all available optimizations that do not significantly increase code size. In particular the compiler does not perform loop unrolling or function inlining when you specify **-O2**. Compared to the optimizer settings documented in the [Gcc] reference manual we also enable**-fweb** at **-O2**.

-O3

As **-O2** but also enables **-finline-functions -frename-registers -funswitch-loops**.

-Os

Optimize for size: a lot like **-O2**, but with additional optimizations to reduce code size, and disabling -**falign-functions -falign-jumps -falign-loops -falign-labels -freorder-blocks**.

2.2.1 Optimizing for Speed

In our experience maximum performance is usually obtained by using the -**O2** or -**O3** optimization level. It depends on your application, because sometimes the increased code size due to -**O3**'s loop unrolling can slow a program down, by increasing instruction cache thrashing. We suggest experimentation and using profile feedback to tune the loop unroller.

There are many other compiler flags which allow you to control individual optimizations. Not all of them will do anything useful, but here are a few which do have some useful effect:

-**funit-at-a-time**

Enabled at**-O2**. Instructs the compiler to perform whole module (intra-module) optimization by completely parsing a source file before beginning optimization and code generation. In this way the compiler can use information about all of the functions in the module to make better inlining and optimization decisions.

Furthermore this can perform “inter-module” optimization of your whole program, or a subset of it. This exposes many more optimization opportunities to the compiler, at the cost of greatly increased memory usage in the compiler and compilation time. This feature only works for C, and not yet C++. It requires changing your Makefiles so that instead of using individual commands to compile each module to object code, and then linking the object files together, like this:

```
mips-sde-elf-gcc -O2 -c moda.c -o moda.o
mips-sde-elf-gcc -O2 -c modb.c -o modb.o
mips-sde-elf-gcc -o prog moda.o modb.o ... -lc ...
```

... you now compile a group of modules together with a single invocation of the compiler, like this:

```
mips-sde-elf-gcc -O2 -c moda.c modb.c -o all.o
mips-sde-elf-gcc -o prog all.o ... -lc ...
```

--param inline-unit-growth=

The automatic function inliner (enabled at **-O3**) can be fine-tuned to limit the total growth of a module due to inlining, as a percentage. For example **-param inline-unit-growth=5** limits the total code growth to approximately 5% - the default being 50%, which may be too high for some embedded applications.

-ffast-math

Switches on **-fno-math-errno**, **-funsafe-math-optimizations**, -**fno-trapping-math -ffinite-math-only** and **-fno-signaling-nans** allow the compiler to be much more ambitious when optimizing floating point arithmetic, but it can generate incorrect code if a program depends on an exact implementation of IEEE-754 specifications of precision, non-finites and exception handling. Many embedded applications won't care about this, and can safely enable these extra optimizations.

-fprefetch-loop-arrays

Enables automatic generation of additional instructions to prefetch data accessed sequentially within loops into the cache. On CPUs which implement the `pref` instruction, such as the 24K and 34K, this can increase performance when accessing large arrays. But since this adds extra instructions it may also reduce performance. You can instead use explicit directives where you know it matters.

`-funroll-loops`

Unroll loops whose number of iterations can be calculated at compile time, or at run time upon entry to the loop. It also turns on complete loop peeling (see below). This option makes code larger, and may or may not make it run faster. It is enabled automatically by **`-fprofile-use`**, i.e. when you tell the compiler to perform profile directed optimizations. Most of the loop optimizations can be further fine-tuned using **`-param`**, see the [Gcc] manual for more details.

`-fpeel-loops`

Peels loops when there is enough information that they do not roll much (e.g. from profile feedback). It also turns on complete loop peeling which completely removes loops which iterate a small constant number of times. This option is enabled automatically by **`-fprofile-use`**.

`-funswitch-loops`

Enabled at **`-O3`**. Moves loop invariant conditional tests out of the loop, and then duplicates the loop inside each branch of the conditional. For example:

```
for (i = 0; i < n; i++) {
  if (a < 0)
    arr[i]--;
  else
    arr[i]++;
}
```

would become:

```
if (a < 0) {
  for (i = 0; i < n; i++)
    arr[i]--;
} else {
  for (i = 0; i < n; i++)
    arr[i]++;
}
```

`-fprofile-generate`

Adds code to your program so that when run it will collect profile data which can then be used by **`-fprofile-use`**. This requires that your program has access to a file system where it can store the profile data. Your program will run slower with this extra profiling code, so don't use this option when generating your final executable.

`-fprofile-use`

Use the profile data generated by running a program compiled with **`-fprofile-generate`** to decide when optimizations which increase the size of a program are worthwhile. This also enables **`-funroll-loops`** **`-fpeel-loops`** **`-ftracer`** **`-fprofile-values`** and **`-fvpt`**.

When using optimizations which increase code size we strongly recommend that you measure the effect of each option on your performance.

2.2.2 Optimizing for Size

Use the **-Os** flag to tell the compiler that your priority is to reduce code size. This is similar to **-O2**, but subtly alters optimization heuristics in the interests of making your code smaller. (Higher optimization levels can otherwise increase code size in order to achieve better performance).

Further space savings can be made by the addition of some or all of the following flags. But here as elsewhere: if you're not quite sure what they do, don't use them. Most applications will do just fine with **-Os**.

-finline-functions

Inlining of very small functions can actually reduce code size, by removing the function call overhead. This is now enabled by default by -**Os**, with the following additional parameters implied:

--param inline-unit-growth=0

Limits the total growth of a module due to inlining to approximately 0% - the default for speed is 50%.

--param max-inline-insns-auto=5

Sets the maximum size of function (in internal gcc instructions) which will be considered for automatic function inlining to 5 instructions, - the default for speed is 120 instructions.

--param max-inline-insns-single=5

Similar, but applies to functions declared with an explicit `inline` and to C++ class methods - the default for speed is 500 instructions.

-fmerge-all-constants

Used in addition to the default **-fmerge-constants** this enables merging of *const* variables, as well as constant strings and literals. Languages like C and C++ require that each non-automatic variable has a unique address, so using this will result in non-conformant behavior - you will need to check that your program can survive this.

-mno-check-zero-division

Prevents the normal insertion of inline code which checks for integer divide-by-zero etc.; this won't affect performance, but it is not recommended when debugging your program. This is the default when compiling for MIPS16.

-fno-rtti

For C++ programs which do not use *dynamic cast* and *typeid*, use this option to disable generation of C++ runtime type identification information for every class with virtual functions. This can reduce the size of the code and data.

-fno-exceptions

For C++ programs which do not use exceptions, use this option to disable the generation of the frame unwind information - which will significantly reduce the read-only data size.

-ffunction-sections

Causes each function to be emitted into its own unique object code section. See below how this can be used to reduce code size.

-fdata-sections

Like **-ffunction-sections**, but for variables.

Finally, if you are using the standard SDE run-time board support "kit" code (provided with the bare-metal/ELF version of the tools), then you can in some cases use a stripped-down version of this library.

2.2.2.1 Code and Data Garbage Collection

You can use **-ffunction-sections** and **-fdata-sections** to reduce the size of some applications, to allow automatic removal of unused functions and variables. But note that if your application does not contain much unused code or data, then these flags might slightly increase the total size, due to extra padding between functions and variables.

The trick is achieved by compiling your source files with one or both of these options, which causes each function and variable to be placed into a unique object code section, and then instructing the linker to "garbage collect" unused sections, as identified by performing a tree-walk of all code and data cross-references, starting from the program's entry point. The linker will do this when given the **-gc-sections** option.

Here's an example showing just two files being compiled and linked:

```
$ mips-sde-elf-gcc -Os -ffunction-sections -fdata-sections -c a.c -o a.o
$ mips-sde-elf-gcc -Os -ffunction-sections -fdata-sections -c b.c -o b.o
$ mips-sde-elf-gcc -Wl,-gc-sections -o prog a.o b.o
```

If you are using the SDE example makefiles (provided with the bare-metal/ELF version of the tools) you can do this by setting the CFLAGS and LDFLAGS variables, e.g.

```
$ make SBD=MSIM32 CFLAGS="-Os -ffunction-sections" \
     LDFLAGS="-Wl,-gc-sections"
```

Note that these options shouldn't be used when debugging your code - the multiple sections will confuse the debugger - only do this for production builds.

Tip: It may be counter-productive to use **-fdata-sections** when compiling MIPS16 code, since it disables the MIPS16 "section-relative addressing" optimization.

2.3 GP-relative Addressing

The GCC manual describes the **-Gnum** option, which controls the maximum size of global and static data items that can be addressed in one instruction instead of two. The default value is 8 bytes, which is large enough to hold all simple scalar variables. This optimization technique is known in MIPS toolchains as gp-relative addressing, and relies on the compiler, assembler, linker and run-time initialization code cooperating to pool all of the "small" data items together into a single region, and then setting the gp register to point to the middle of this region. These items can

then be referenced with a single instruction, using a signed 16-bit offset (i.e. -32768 to 32767) from the gp register ($28), instead of the usual two instruction sequence. However there are some potential pitfalls with this technique:

- You must take special care when writing assembler code to declare global (i.e. public or external) data items correctly:

  1. Writable, initialised data of gnum bytes or less must be put explicitly into the `.sdata` section, e.g.:

     ```
        .sdata
     small:.word0x12345678
     ```

  2. Global common data must be declared with the correct size, e.g:

     ```
     .commsmall, 4
     .commbig, 100
     ```

  3. Small external variables must also be declared correctly, e.g:

     ```
     .externsmallext, 4
     ```

- In C you must declare global variables consistently in all modules which define or reference them. For external arrays either omit the size (e.g. `extern int extarray[]`), or give the correct size (e.g. `int cmnarray[NARRAY]`). Don't just give a dummy size of 1. Watch out particularly for use of the magic compiler/linker variables like `_end`, `_edata`, etc.: they should be declared as character arrays of unknown size, e.g.

  ```
  extern char _end[];
  ```

- If your program has a very large number of small data items or constants, the **`-G8`** option may still try to push more than 64KB of data into the "small" region; the symptom will be obscure relocation errors ("relocation truncated") when linking. Fix it by disabling gp-relative addressing with the **`-G0`** option; most of the time you won't lose too much.

- Some real-time operating systems and PROM monitors can be entered by direct subroutine calls, rather than via a "system call" trap. The use of simple subroutine calls between sections of the program which were not linked together means that it is not possible for the application and the monitor to share a gp area. In this case either the application or the monitor/RTOS (but not necessarily both) must be built with **`-G0`**.

When a particular **`-G`** option has been used for compilation of any set of modules, then it is usually necessary that all other modules and libraries should be compiled with the same value, to avoid linker relocation errors (e.g. one module references a variable which it thinks is in a "small data" section, while the other defines it in a non-small section). To avoid relocation overflow errors when linking, the safest solution is to compile all modules within a mixed 32-bit and MIPS16 system using the same value of **`-G`**.

Of course larger values of **`-Gnum`** can be used to increase the scope of this optimization. However, at the moment the only way to find the limit is an iterative process of recompiling with increasing values, until you overflow the 64K limit. One day it may be possible to determine an optimal value automatically.

The `bestgp` program will estimate the optimum value to use when compiling your program. Simply run `bestgp`, giving it the name of your already-linked executable, and then recompile your program using the new value that it reports. You should probably only use this facility during final tuning, since any large changes to your code may cause the gp-area to overflow. In that case you'll need to rebuild with **`-G`**, and repeat the whole process.

2.4 Unaligned Data

The standard MIPS load and store instructions require that all data is aligned on its “natural” boundary, i.e *shorts* on a multiple of 2 bytes, *ints* on a multiple of 4, and *doubles* on 8. If the alignment is not correct, then the CPU will generate an address exception.

Because of this restriction, *gcc* will normally align all data structures and their fields on their natural boundaries. However some software ported from 8 or 16-bit CPUs may rely on data structures whose fields align to a smaller boundary (e.g. for network protocol headers, or printer font cartridges, etc.).

There are two ways to convince gcc to change its default alignment rules:

1. Use the GCC `attribute (packed)` extension on whole structures or individual structure fields - see the Extensions section of the GCC manual for full details.

2. Precede the definitions of packed structures with the single line `#pragma pack(x)`, where x is the alignment boundary, in bytes. Follow the declaration with the line `#pragma pack(x)`, which restores the normal alignment rules - don't forget this, your code may continue to work, but quietly become bigger and slower! For example:

```
#pragma pack(1)
struct packedstruct {
  short  s;                          /* offset: 0 */
  int     i;                          /* offset: 2 */
  int     j;                          /* offset: 6 */
};
#pragma pack()
```

3. In desperation you can compile your program with the **`-fpack-struct`** option, which removes padding from all structures. But that will make your whole program bigger and slower, and may cause problems such as ABI incompatibility with libraries that weren't compiled with this option.

The compiler will where possible use the MIPS left/right load and store instruction pairs to access unaligned structure fields, but this will be less efficient than if the data were correctly aligned. So use the pack options only on data structures where it is essential.

However these mechanisms do not solve the problem of how to handle unaligned pointers to simple scalar types (e.g. `int`). Currently there are two ways to handle this:

If you know that there a few specific pointers which will frequently hold unaligned addresses, then you can modify your code to use the generic macros defined in `<unaligned.h>`. For example replace this:

```
int foo (int *ip)
{
  return *ip;   /* ip is known to be frequently unaligned */
}

void bar (short *sp, short val)
{
  *sp = val;    /* sp is known to be frequently unaligned */
}
```

by this:

```
#include <unaligned.h>

int foo (int *ip)
{
  return unaligned_get (ip);
}

void bar (short *sp, short val)
{
  unaligned_put (sp, val);
}
```

Where there are very occasional and unpredictable unaligned references, then you can install an exception handler which fixes up instructions which generate an unaligned Address Error (**`XCPTADES`** or **`XCPTADEL`**) exception. So long as you use standard exception handlers then you can do this by putting a call to `_mips_unaligned_init()` at the beginning of your code, or simply by defining `FEATURES=unaligned` if you are using the example SDE makefiles (provided with the bare-metal/ELF version of the tools). Certainly don't do this for performance critical code - just use it as a way to get started when first porting an application.

2.5 Software Floating Point

When an application performs floating point computations and the target CPU is not equipped with a floating point unit (called coprocessor 1, or"CP1" in MIPS-speak), then the floating point operations must be performed by software subroutines. The SDE library (provided with the bare-metal/ELF version of the tools) includes an IEEE-754 compliant software floating point library (in library `libe.a`, or **`-le`** to the linker) which performs floating point arithmetic using only integer operations. There are different ways in which this library is used:

1. When you use the compiler's **`-msoft-float`** option it will keep all floating point values in integer registers (a pair of them for double-precision when using 32-bit registers), and will generate direct calls to the software floating point library to perform all floating point arithmetic. This is the best option if you know that you will never have a hardware floating point unit in your target system.

2. If **`-msoft-float`** is not used (or **`-mhard-float`** is) then the compiler will emit code which uses hardware floating point registers and instructions. You then have to include a "CP1 emulator" in your program which catches "Coprocessor Unusable" traps, interprets the instructions, and invokes the software library to emulate them. This results in even slower code than when using **`-msoft-float`**, but may be the option to use when creating a single program binary which must be capable of working either with or without a hardware floating point unit, detected at run-time.

3. The final option is only for the IDT R4650 and R4640 CPUs, which are equipped with only a single-precision floating point unit. For these CPUs you should use the **`-msingle-float flag`**, which tells the compiler to generate hardware floating point instructions for single-precision operations, but call the emulation library for double-precision.

Remember that in all cases emulated floating point is much slower than hardware - up to 100 times slower for the trap-based emulation.

The example makefiles determine which options to use based on the value of the **`FLOAT`** parameter defined for that program, and the **`FPU`** parameter defined for the selected target board.

2.6 64-bit Support

The toolchain supports the MIPS64 instruction set, which extends the MIPS34 architecture to support 64-bit integer registers. All MIPS64 CPUs, when equipped with an FPU, also support the full set of thirty two 64-bit floating point registers.

There's no "mode switch" for 64-bit operation in MIPS architecture processors. 64-bit CPUs execute all the 32-bit instructions, always producing 64-bit results, and where possible doing the same job - so the `or` instruction on 64-bit CPUs is always a 64-bit "inclusive or", and so long as you only give it valid 32-bit operands, you'll always get a valid 32-bit result. Separate 64-bit versions of instructions are required only if they generate results which might overflow from 32 bits (i.e. the 64-bit result might not be equal to the result of sign-extending the 32-bit result to 64-bits); so as well as the 32-bit `addu` there is now a 64-bit `daddu`, and similarly for `dsubu`, `dsll`, etc. See [Sweet99] for an account of how this all works.

When you use the **`-mips64`** option, you allow the compiler to generate 64-bit instructions, and implicitly select a different calling convention - also called an ABI (Application Binary Interface).

2.6.1 64-bit Calling Conventions

Once you select a 64-bit ISA or ABI then your computations will use 64-bit registers, and computations on `long long` variables will use 64-bit machine instructions. To support the wider registers, a new and more efficient calling convention is used: by default this is the "N32" ABI.

Although N32 sounds like a 32-bit calling convention, it requires a 64-bit CPU and is - except for the most trivial cases - wholly incompatible with the old 32-bit MIPS calling convention commonly known as "O32" (which is what Silicon Graphics called it). See [Sweet99] for a discussion of the calling conventions and why they're like they are. You can also find a detailed description of the N32 ABI, and a discussion of 32- to 64-bit porting issues at SGIs

It is important to note that while the N32 ABI does support 64-bit registers and uses 64-bit instructions, it does not use 64-bit pointers. N32 implements the "ILP32" model, where int, long and pointer types are all 32-bits - only `long long` and `double` are 64-bits. In an embedded system, this seems to be the best compromise. It is not clear that 64-bit addressing is useful or sensible in most embedded environments - it certainly increases the memory footprint, and it introduces significant portability problems (e.g. `sizeof(void*) !=sizeof(int)`) which many embedded applications have not yet had to deal with. If you need to access physical addresses above 512MB then you could use a TLB entry to map the physical address into the 32-bit virtual address space (i.e. KSEG2 or KUSEG); or you could store 64-bit addresses in `long long` variables, and then use assembler subroutines or C `asm`'s to perform the loads and stores.

Unlike some 64-bit versions of GCC, the compiler does not currently provide a 128-bit extended `long double` type when using the N32 ABI: instead a `long double` is treated as identical to a 64-bit double. The cost of implementing such an extended precision type would be significantly increased code size, due to the 128-bit software floating point emulation library which would then be required.

2.6.1.1 N64 ABI (partially supported)

The compiler and other tools can generate code which uses the "N64" ABI, which uses 64-bit long and pointer types, with a 32-bit int - known as the "LP64" model. Select this by using the **`-mabi=64`** option, but note that the N64 ABI is not compatible with N32, and is not currently supported by SDE header files, libraries and run-time system (all supplied with the bare-metal/ELF version of the tools). If you absolutely have to use 64-bit pointers and N64, then you will have to bring your own 64-bit-safe header files and libraries.

2.6.1.2 O64 ABI (deprecated)

A different 64-bit calling convention was defined and used by Cygnus/RedHat and some of their customers. You could describe it, approximately, as what you get by taking the 32-bit ``O32'' standard and replacing all the 32-bit fields by 64-bits, and this is called the ``O64'' ABI (specify **`-mabi=o64`**). It's incompatible with the SDE libraries (supplied with the bare-metal/ELF version of the tools), and we have not tested it in any way - we don't recommend its use for new development projects.

2.6.1.3 ABI-specific Code

If you need to write assembler routines which stand some chance of working in either call-convention universe, use compiler predefined macros as follows:

```
#if _MIPS_SIM == _ABIO32
/* 32-bit O32 calling convention */
#endif
#if _MIPS_SIM == _ABIN32
/* 64-bit N32 calling convention */
#endif
```

Available for backward compatibility are the SGI-inspired `_MIPS_SIM` definitions, as follows:

```
#include <sgidefs.h>
#if _MIPS_SIM == _MIPS_SIM_ABI32
/* 32-bit O32 calling convention */
#endif
#if _MIPS_SIM == _MIPS_SIM_NABI32
/* 64-bit N32 calling convention */
#endif
```

By including the file `<mips/asm.h>` you have access to the utility macros described in the SGI N32 ABI Handbook, as in the link above, such as:

2.6.2 64-bit Optimization

Unfortunately GCC is not always as successful at optimizing 64-bit code as it is with ``normal'' 32-bit code. In particular it sometimes fails to spot when it can avoid conversions between 64- and 32-bit values in registers, and can fail to optimise certain sub-expressions involving 64-bit constants. We suggest that you use `long long` only where the extra bandwidth or precision is important, and don't try to use it as a global replacement for int or long.

2.6.3 64-bit Assembler Changes

Like the compiler, the assembler recognises the directives which identify a 64-bit CPU. See [Sweet99] for a complete description of 64-bit features, or [Kane92] for a reference-manual approach to the machine instructions.

2.6.3.1 64-bit Assembler Constants

To prevent uncertainty regarding their size, and whether or not they are sign-extended, immediate operands are truncated to 32-bits. You can specify full 64-bit immediates only for the `dli` instruction and `dword` pseudo-op.

2.7 MIPS16® ASE Support

The MIPS16 ASE is only available on bare-metal/ELF targets. It is not available for the GNU/Linux target.

The ``MIPS16'' instruction set is an extension to the MIPS architecture (an ``ASE'') that allows you to build much smaller binaries. It requires that the CPU implement a set of operations encoded with fixed-length 16-bit instructions; this new instruction set is selected with a ``mode switch'' controlled by a ``least significant bit'' included in the instruction address. You can successfully build and run a program with a mix of functions built both with MIPS16 and conventional instructions, but you can't mix the two instruction sets inside one C function.

The MIPS16 ASE is most useful to the smallest and most deeply embedded systems, and is often not implemented on higher-end CPUs. ``MIPS16'' is the name of an enhanced version of the MIPS16 instruction set; the enhancements were worked out from experience and help the compiler generate even smaller code. Note that all those MIPS34-compliant CPUs which support the MIPS16 ASE implement the MIPS16 extensions.

Most often a MIPS16 operation corresponds to a single conventional MIPS instruction, but the small size imposes restrictions on choice of registers and the size of ``immediate'' fields.

For straightforward integer code **`-mips16`** can cut code size by around one third, but it certainly won't do this if:

1. you use floating point: the MIPS16 ASE doesn't encode f.p instructions or registers, which have to be replaced by calls to 32-bit code - even if the CPU has an FPU, or

2. you use unaligned data structures heavily: there are no `lwl` or `lwr` MIPS16 instructions, so these have to be synthesised as a sequence of byte loads, shifts, ors, etc.

Most users will never, and should never, write MIPS16 assembler code. You'll find no assembler language documentation here. MIPS16 instructions are meant to be an intermediate code generated by the compiler to save space - possibly at the cost of some speed. MIPS16 CPUs always run the normal 32-bit MIPS instruction set as well, which is usually a better choice for assembler modules.

MIPS16 functions can safely call functions consisting of ordinary 32-bit MIPS instructions, and vice versa. The hardware keeps track of MIPS16 mode by adding a bit zero to the instruction address pointer; so a jump-register instruction to an odd address implicitly switches into MIPS16 mode. Because normal absolute `jal` instructions don't contain the bottom address bits (since regular MIPS instructions are 4 byte aligned), a new instruction `jalx` is added which calls MIPS16 code from regular 32-bit code, or vice versa. The linker automatically converts a `jal` to a `jalx` when it sees a call across the MIPS16/regular-MIPS divide.

MIPS16 functions using floating point must be declared carefully. The compiler automatically generates small ``trampoline'' stubs to copy floating point arguments and results back-and-forth between ``hard'' f.p. registers and the MIPS16 integer registers used for f.p. arguments. It's essential to provide full prototypes for such functions.

2.7.1 Global Variables and MIPS16 Code

The global-pointer (GP) optimization used in 32-bit MIPS code to speed up access to small global variables is not usually appropriate to MIPS16 code, with its restricted load offsets (all GP-relative addresses would require an extended instruction). A mechanism has been developed for MIPS16 code which accesses variables defined within the same compile unit as the code using short ``section relative'' offsets. This optimization is of no benefit to ``extern'' or ``common'' variables, but is a big win when accessing locally defined variables.

In order for this optimization to be more effective, code compiled using **`-mips16`** or **`-mips16e`** will by default also imply the specification of ``**`-G0 -fno-common`**''. This has the following implications:

- If you are compiling any modules using a 32-bit ISA, but you expect that they may be linked with MIPS16 code, then you must specify explicitly for the 32-bit modules. You can still link with existing, pre-compiled, 32-bit libraries that were compiled gp-relative addressing enabled, so long as the precompiled code does not try to reference global symbols defined in the **`-G0`** compiled code. Using **`-mno-gpopt`** is a better choice because then

the small variables are placed in the small data sections, but the compiler just won't generate short references to them. The safest solution is to compile all modules in a mixed 32-bit and MIPS16 system using **-G0**.

- The "`traditional" (but not ISO / ANSI compatible) C "common variable" behaviour - named after the Fortran construct, which allows several modules to declare the same global variable, as long as no more than one of the declarations actually initialises the variable - will no longer work. If possible you should avoid relying on this feature in portable code, but if it cannot easily be changed in your code, then you will have to specify -**fcommon** on the command line, and you will lose the section-relative addressing optimization on uninitialised global variables (uninitialised static variables will be optimized). Existing, pre-compiled libraries which use common variables will continue to work correctly when linked with code compiled with **-fno-common**, as long as they don't initialise the same variables.

- You can flag individual variables where ``common" behaviour is absolutely required, by using gcc's `__attribute__ mechanism`. For example:

```
int errno __attribute__((common));
```

2.7.2 Global Register Variables

In MIPS16 code only 8 registers are directly usable for arithmetic and pointers, but the remaining 24 registers are accessible indirectly. The compiler allows MIPS16 code to use gcc's global register variable extension to access these extra registers, which can provide a performance boost for global variables which are very frequently accessed in many separate, small functions. It is recommended that callee-saved registers $s3-$s7 only are used for this purpose ($s0 and $s1 are used by normal MIPS16 code, $s2 is used by MIPS16 code if there is a hardware FPU, and $s8 is sometimes used as a stack frame pointer in 32-bit code).

Global register variables must be declared in a header file which is common to all modules, so that the register does not get reused for normal variables or temporaries by 32-bit code. Here is an example of how to declare and use a global register variable:

```
register struct insn *curinsn __asm__("$s3");
unsigned int getinsn_opcode (void)
{
    return curinsn->opcode;
}
```

2.7.3 Divide by Zero Checks (-mcheck-zero-division)

When generating MIPS16 code the compiler will not generate the extra code to check for division by zero, so divide by zero will generate an undefined result. If for debugging purposes you wish division by zero to generate a trap, then use the **-mcheck-zero-division** compiler option.

2.7.4 Execute-only Code / Split I-D RAM

In MIPS16 code the compiler normally places implicit constants inline within the executable code section, interleaved with or following the function which uses them. This allows the constants to be accessed efficiently using the MIPS16 PC-relative load and addiu instructions.

However some MIPS Technologies cores support independent, Harvard-style on-chip instruction and data memories known as SRAM or SPRAM. In such a configuration a program cannot read constant data from the I-side memory without special hardware support, which causes the CPU to treat the MIPS16 PC-relative load instructions like an instruction fetch, and ``redirect" the load from the D-side memory port to the I-side.

Use the **-mno-data-in-code** flag when compiling MIPS16 code to run in ISRAM on a system without the hardware redirect. It will generate larger and slower code (5% larger on average) - so don't use it unless you have to. Also make sure that you use the **-mno-data-in-code** flag when linking your program, to select a compatible multilib variant.

When the **-mno-data-in-code** flag is used, it also switches off the **-G0** option - otherwise the default for MIPS16 code - so that it can place the constants into the small data section, and access them via the $gp register. You can use the **-G0** option explicitly to prevent this, but it may increase code size significantly.

The **-mcode-xonly** flag is a weaker alternative for MIPS16 code running in on-chip ISRAM where the system does implement the hardware redirect (e.g. the M4K). The hardware redirect operates only for PC-relative loads, but MIPS16 code can still create pointers to the implicit constants - most obviously to literal character strings - to be used later by conventional load instructions, which would then read the data from the wrong memory.

The **-mcode-xonly** option instructs the compiler not to place constant strings and computed jump tables into the code segment, while keeping simple integer and floating point constants inline with the code. This will usually result in only slightly larger code than a standard MIPS16 compilation. All of the MIPS16 libraries are now built with this option.

The **-mcode-xonly** flag may also be useful for cores which implement the SmartMIPS ASE, which provides an extended virtual memory protection model that can mark pages as ``execute-only''. Similar to the I/D redirect above, the MIPS16 PC-relative load represents itself to the TLB as an instruction-fetch so, for MIPS16 code running in mapped space, use **-mcode-xonly** to prevent strings and jump tables from being placed in the executable code section.

2.7.5 Generating MIPS16® Code

Add the compiler flag **-mips16** or **-mips16e**, and the module will be compiled using MIPS16 or MIPS16 instructions to generate compact code. The flags are (mostly) orthogonal in effect to other flags which set code generation options.

It goes further than that: the **-mips16** flag used on the mips-sde-elf-gcc command line when linking your files will select MIPS16 or MIPS16 libraries.

Back to compilations: sometimes a module might contain functions you want to compress, and some you would rather compile to regular 32-bit instructions - perhaps because the 32-bit instructions will give you better performance, or because you need to use instructions that are not available in MIPS16.

MIPS16 code always takes longer to execute within the CPU, but if instruction fetch bandwidth is the critical determinant of the performance of some piece of code, then the smaller size of MIPS16 code can make it faster overall.

The compiler uses the GCC `_attribute_ extension` to permit the instruction set to be selected on a per-function basis. For example:

```
__attribute__((mips16)) void smallfunc ()
{ /* generates MIPS16 code */ }

void __attribute__((nomips16)) bigfunc ()
{ /* generates 32-bit MIPS code */ }

void normalfunc ()
{ /* compiled as per command-line flags */ }
```

It is likely that the attribute construct will be hidden by a macro, which can be controlled by an ifdef, e.g.

```
#if __mips
#define large__attribute__((nomips16))
#define compact__attribute__((mips16))
#else
#define large
#define compact
#endif



compact void smallfunc ()

{ }
```

If the command-line selects **-mips32**, then `__attribute__((mips16))` will generate extended MIPS16 instructions, otherwise it will generate only "standard" MIPS16 instructions. Similarly, if the command-line selects **-mips16e**, then `__attribute__((nomips16)` will generate MIPS34 code.

If you have used the `mips16` attribute, but wish to prevent it from taking effect, then compile with **-mno-mips16**.

2.7.6 Sibling Call Optimization

If you are mixing MIPS16 functions and 32-bit functions in your program, then it is not safe to allow the compiler to perform its ``sibling call'' optimization, which can replace a call at the end of a function by a jump to the other function. Since there is no `jx` instruction to switch from 32-bit to MIPS16 mode, only `jalx`, this optimization must be prevented when a 32-bit function calls a MIPS16 function. If it were to occur, then it would result in a error message from the linker when it tried to relocate the jump instruction. You can prevent this optimization from taking place in two ways:

1. Most easily by using the compiler's **-fno-optimize-sibling-calls** option.

2. At a more fine-grain level, by ensuring that all global MIPS16 functions are correctly declared with function prototypes which include `_attribute__((mips16))` in the function type. It is not necessary to do this for 32-bit functions, since the compiler will never generate sibling calls from MIPS16 functions.

2.7.7 Main differences between MIPS16® and MIPS16e™ Code

The new MIPS16 instructions clean up a few wrinkles where the original MIPS16 definition caused the compiler to generate wasteful code. These are:

- An instruction to save registers and do other function entry housekeeping, with a matching instruction to restore registers on function exit. (They only support a 32-bit register model.)

- Instructions which sign- or zero-extend partial-word values in registers.

- Variants of the indirect jump and jal instructions which don't have a visible branch delay slot.

You'll be surprised how much they help.

If you have only the original MIPS16 instruction set available, then there is a compiler option **-mentry** which will reduce code size further, by generating ``reserved'' MIPS16 instruction codes to perform commonplace function

entry/exit housekeeping. These unimplemented instructions cause an exception and require an appropriate exception handler - that's dreadfully slow with the standard kit, and will be pretty bad even with a tuned handler.

2.8 Predefined Preprocessor Macros

Your program can detect what sort of CPU and instruction set it is being compiled for by testing a number of predefined C preprocessor macros. For example:

```
#if __mips == 32
#if __MIPSEB
/* big-endian MIPS32 code */
#endif
#if __MIPSEL
/* little-endian MIPS32 code */
#endif
#enfig
```

The full table of predefined macros defined by GCC for MIPS is as follows:

Table 2.2 MIPS Predefined Macros

Macro	Purpose
`_mips`	Defined whenever compiling code for a MIPS ISA. Has as its value the selected ISA level, e.g. 1 for **`-mips1`**, 32 for **`-mips32`**, and 64 for **`-mips64`**.
`__mips_isa_rev`	The ISA revision level - only relevant for MIPS34 and MIPS64 - has the value 1 for the original revision, or 2 for the second revision of the ISA (i.e. **`-mips32r2`**).
`__mips64:`	Defined when compiling for an ISA which supports 64-bit general purpose registers. Not the same as ``__mips == 64``", since it will also be defined for the 64-bit MIPS 3 and MIPS IV ISAs.
`__mips_fpr`	Specifies the size in bits (64 or 32) of each floating point register, as selected by the base ISA and ABI, or by the **`-mfp64`** compiler flag.
`__mips16`	Defined when **`-mips16`** is used to select generation of compact MIPS16 code.
`__mips_hard_float`	Defined when generating hardware floating point instructions.
`__mips_soft_float`	Defined when **`-msoft-float`** is used, and the compiler will generate calls to a software floating point emulation library.
`__mips_no_float`	Defined when the **`-mno-float`** flag is used, to request libraries without floating point support, to reduce program size. Otherwise equivalent to `_mips_soft_float`.
`__mips_dsp`	Defined when the DSP ASE is enabled, using either the **`-mdsp`** compiler flag is specified, or when the **`-march`**= option specifies one of these CPUs from the 24ke/34k/74k family.
`__mips_paired_single_float`	Defined when **`-mpaired`**-single is used to enable the ``paired single" SIMD floating point extension.

Table 2.2 MIPS Predefined Macros (Continued)

Macro	Purpose
__mips3d	Defined when the **-mips3d** flag is used to enable the MIPS 3D ASE.
__mips_smartmips	Defined when the **-msmartmips** flag is used to enable the SmartMIPS ASE, or when enabled implicitly because**-march**= is set to 4ksc or 4ksd.
_MIPSEB	Defined when compiling code for a big-endian CPU, i.e. when the **-EB** flag is used.
_MIPSEL	Defined when compiling code for a little-endian CPU, i.e. when the **-EL** flag is used.
_MIPS_ARCH	Where CPU is the name specified with the compiler's **-march**= option, converted to upper case.
_MIPS_TUNE	Where CPU is the name type specified with the compiler's **-mtune**= option, converted to upper case
_SOFT_FLOAT	Same as _mips_soft_float, for compatibility with previous versions of MTI SDE.
_NO_FLOAT	Same as _mips_no_float, for compatibility with previous versions of MTI SDE.
__pic__	Defined when generating MIPS/abi position-independent code, as selected by the **-fpic** or **-mabicalls** compiler flags.
__PIC__	Equivalent to _pic_.
__SDE_MIPS__	Defined to indicate that the code is being compiled by an configuration of GCC, and the SDE headers and libraries will be available.
_MIPS_SIM	Indicates the ABI or calling convention in use - takes one of the values _ABIO32 (1), _ABIN32 (2), _ABI64 (3), _ABIO64 (4), or _ABIEABI (5).
_MIPS_FPSET	Indicates the number of 64-bit floating point registers available: 16 or 32. This encodes the same information as _mips_fpr above, but in a different way, and is included for compatibility with Irix.
_MIPS_SZINT	Indicates the size in bits of the int type: 32 or 64.
_MIPS_SZLONG	Indicates the size in bits of the long type: 32 or 64.
_MIPS_SZPTR	Indicates the size in bits of pointer types: 32 or 64.

The compiler also makes a number of predefined ``assertions" which can be tested at compile-time, however these are deprecated in favor of the more widely supported conventional pre-processor macros and constants.

Chapter 3

GDB Debugging with the MDI interface

NOTE: Whenever “SDE” is mentioned, this refers to the SDE Library that is supplied with the bar-metal target/ELF toolchain. Such comments do not pertain to the linux target toolchain

NOTE: Throughout this document, the command prefix “mips-sde-elf-” is used (assuming that you’re using the bare-metal/ELF), for example `mips-sde-elf-gdb`. If your target is actually linux, the command prefix would actually be “mips-linux-gnu-”.

Source-level debugging of an embedded application requires two components. The host debugger *mips-sde-elf-gdb* has access to your source and object files, and understands the structure of your program and data. But to interact with the running software *gdb* needs to be able to read/write memory and registers, and access on-CPU debug functions on your target system.

The connection between GDB and the target will be one of the following:

1. A connection which exploits an on-CPU debug connection such as MIPS Technologies' EJTAG. This will need a special piece of hardware (a probe) connected to the CPU on the board under test, some physical connection to the probe (typically Ethernet, USB or parallel port), and some host software to connect GDB to the probe.

   MIPS Technologies promotes a software interface called “MDI”; it's a standard interface for the on-host software which connects to an EJTAG probe..

   Some EJTAG probe manufacturers don't provide an MDI interface, but are compatible with *gdb*'s standard remote debug protocol (Abatron, for example). Some others have totally proprietary interfaces, in which case they may come with their own proprietary debugger, which may be compatible with the compiler - check with your probe supplier.

2. An ethernet or serial port connection to the target, together with a “target monitor” program running on your target CPU. The target monitor is a little “server”, attached to the host via serial port or network link, which can be requested to inspect or patch memory, to catch exceptions (particularly breakpoint exceptions) and report the application's CPU state.

   MIPS Technologies' YAMON monitor includes a built-in target monitor, which can communicate directly with *mips-sde-elf-gdb* over a serial port. But if your target doesn't have the YAMON monitor (or if your application takes over exception handling from the ROM monitor, or if you need multi-thread support) then you can instead rely on linking your application with the “remote debug stub” code provided with SDE run-time software (provided with the bare-metal/ELF version of the tools).

3. Your target may not be a real piece of hardware, but a software simulator. The basic GNU MIPS simulator included with the toolchain is built-in to *mips-sde-elf-gdb*; while MIPS Technologies' MIPSSim simulator (much more grown-up and accurate) is available as a separate DLL which connects to *gdb* via the MDI interface. MIPSSIM is available from MIPS Technologies.

Usually you will use *gdb*'s `load` command to download your application to the target - but that can be very slow and tedious over a serial port. If you don't have a dedicated debug probe, then a ROM monitor which supports Ethernet downloading (such as the YAMON monitor) can be very helpful - see Chapter 4, “Manual Downloading” on page 61.

All of the debugging features described in the [Gdb] reference manual are available for remote programs, but note:

1. While you may be able download a program via Ethernet, or some other high-speed mechanism, you will usually still need some other connection (e.g. EJTAG or serial cable) by which *gdb* can control the monitor. No known MIPS boards support a complete download and debug cycle over Ethernet alone.

2. Once a program has started running it cannot be restarted simply by using the *gdb*'s `run` command - the initialised data has most likely been modified by the program, and must be re-initialised by reloading the program first.

Please refer to the printed or online GDB manual for more information about the GDB command line interface.

3.1 MDI Debugging

MIPS Technologies promotes a software API called “MDI”; it's a standard procedural interface by which host software can connect to an EJTAG probe or software simulator, via a dynamically loaded library conforming to the Microprocessor Debug Interface (MDI) specification.

Once you have configured MDI for the first time, following the instructions below, it is as easy to operate as any other *gdb* remote target. A typical command-line debug session might start like this on the Host system:

```
$ setenv LD_LIBRARY_PATH /path/to/mdi/link/library:${LD_LIBRARY_PATH}
$ setenv GDBMDILIB library_filename

$ mips-sde-elf-gdb xxxram
(gdb) b main
(gdb) target mdi 15:1
(gdb) load
(gdb) run
Breakpoint 1 at main...
```

The following sections look in more detail at setting up and using the two most common MDI targets: the MIPSSim simulator and an MDI-enabled EJTAG probe.

3.1.1 MDI Debugging with the MIPSSim™ Simulator

MIPS Technologies Inc. has developed the comprehensive and accurate MIPSSim simulator for its core CPUs. If your toolchain did not include MIPSSIM, MIPSSIM is available from MIPS Technologies. The MIPSSim software runs on Windows (NT, 2000 and XP), x86 Linux, and Solaris 2.6 or above.

3.1.1.1 Configuring the MIPSSim™ Simulator for GDB

mips-sde-elf-gdb connects to the MIPSSim simulator via its MDI library interface, and there are a few configuration steps which you must perform first, so that *gdb* can “find” the MIPSSim library.

1. First install, configure and test your MIPSSim package, following the instructions in the MIPSSim Getting Started Guide supplied with it.

2. *mips-sde-elf-gdb* finds the MDI library using environment variables.

For Linux/unix:

For bash, ksh, etc:

```
$ export LD_LIBRARY_PATH=/path/to/mdi/link/library:$LD_LIBRARY_PATH
$ export GDBMDILIB=library_filename
```

For csh and tcsh:

```
$ setenv LD_LIBRARY_PATH /path/to/mdi/link/library:${LD_LIBRARY_PATH}
$ setenv GDBMDILIB library_filename
```

The library filename is most likely something like MIPSsim_MDI.so or MIPSsim_MDI.soc.

For MS windows:

You'll need to make sure that the directory containing the MIPSSIM MDI DLL has been added to your PATH variable, or copy the DLL to \windows\system on Win9x, or \windows\system32 on Win32 systems (NT and above).

There are multiple ways to set the environment variables:

Right Click `MyComputer->Properties->Advanced->Environment Variables`

OR if you're using cgywin:

```
$ export PATH=/path/to/mdi/link/library:$PATH
```

OR in a windows shell:

```
$ set PATH=yourdrive:\path\to\mdi\link\library;%PATH%
```

You'll also need to set the GDBMDILIB environment variable:

```
$ export GDBMDILIB=library_filename
```

The DLL filename is most likely something like MIPSsim_MDI.dll.

3. Now you can start the debugger and use the MDI target:

```
$ mips-sde-elf-gdb helloram
(gdb) target mdi 15:1
```

3.1.1.2 Selecting the MIPSSim™ CPU

When you connect to the MIPSSim simulator you have tell it which CPU core to simulate. You do this by specifying an MDI *target group* and *device* pair. The way that you do this depends on whether you are using the command-line or GUI interface to *gdb.*

1. For the command-line interface to *gdb* enter these commands:

```
$ mips-sde-elf-gdb
(gdb) show mdi devices
Targ 01: Default
    Dev 01: MIPS32_4Kc BE
    Dev 02: MIPS32_4Kc LE
    Dev 03: MIPS32_4Km BE
    Dev 04: MIPS32_4Km LE
    Dev 05: MIPS32_4Kp BE
    Dev 06: MIPS32_4Kp LE
    Dev 07: MIPS32_4KEc BE
    Dev 08: MIPS32_4KEc LE
    Dev 09: MIPS32_4KEm BE
    Dev 10: MIPS32_4KEm LE
    ...
```

That should print out a list of all the CPU devices supported by the MIPSSim software, and their associated target group and device numbers. If it instead says 'MDI not available", then you have probably not installed the MIPSSim package correctly, or not run the `mdi` command to select the MIPSSim library.

Now you can tell *gdb* which device to use. Assuming that you wanted a little-endian 4KEc core, then looking at the above list we can see that it's target group 1, device 8. So:

1. Set the `GDBMDITARGET` and `GDBMDIDEVICE` environment variables to the appropriate target group and device numbers.

   For bash, ksh, etc:
   ```
   export GDBMDITARGET=1
   export GDBMDIDEVICE=8
   ```

   For csh and tcsh:
   ```
   setenv GDBMDITARGET 1
   setenv GDBMDIDEVICE 8
   ```

2. Or add the following *gdb* commands to your `.gdbinit` file:

   ```
   set mdi target 1
   set mdi device 8
   ```

3. Or specify them on the `target` command line when you connect to the MDI library, for example:

   ```
   $ mips-sde-elf-gdb
   (gdb) target mdi 1:8
   ```

Note that in both cases MIPSSim's MDI interface currently lists all of the CPU cores which it knows about, even if that core simulator is not installed. If you select a CPU type for which you do not have the core simulator library installed, then you will see an error reported when you try to connect to it.

3.1.1.3 Building for a MIPSSim™ Target

Select an appropriate value of SBD which most closely matches your chosen CPU family, with the "`MSIM`" prefix. Now you can build one or more of the SDE example programs and run them on the MIPSSim simulator, for example:

1. Change directory to the "hello world" example program:

   ```
   $ cd $SDETOP/examples/hello
   ```

2. Build the example:

   ```
   $ make SBD=MSIM32L
   ```

Note that this will build the rom version of the hello world program. Why? Because the MIPSSim software simulates a bare CPU, without a ROM monitor. When you start a MIPSSim simulation it is going to start executing code at the MIPS reset exception vector in ROM - so you are building a rommable version of the program, which has to initialise the simulated CPU, caches, and so on - just as if it was running on a real, physical CPU.

3. You can run the program in command-line mode:

   ```
   $ mips-sde-elf-gdb helloram
   (gdb) target mdi
   (gdb) load
   (gdb) run
   ...
   (gdb) quit
   ```

3.1.1.4 Downloading to a MIPSSim™ ROM Target

If you use the supplied SDE example Makefiles (provided with the bare-metal/ELF version of the tools) then you can probably skip this section. We include it in case you need to write your own Makefiles, or in case something goes wrong.

When you build a program to blow into a physical ROM memory (e.g. EPROM or Flash) the SDE Makefiles will normally use the mips-sde-elf-conv program to convert it into an ASCII S-record file (or similar), suitable for a PROM programmer. At the same time its initialised, writable data segment is relocated and concatenated to the end of the code segment, from where it is later copied down into RAM. But *gdb* can't load an S-record file, so how do you load a ROM image into a bare MIPSSim simulator via *gdb*?

The answer is that mips-sde-elf-conv takes your executable ELF file, and outputs a new, relocated ELF file with the `.relf` extension. The relocation is done exactly the same way as when creating a real, physical PROM image.

The final step in the chain is that *gdb*'s "load" command automatically checks for a file with the same name as your executable, but with the `.relf` extension. If this is found then it is this file that will actually be downloaded via MDI into the simulated MIPSSim ROM. When execution is started the ROM startup code will (after initialising caches, etc) copy the initialised data and possibly code into RAM. Your program image will now correspond to the original ELF executable file, and debugging can begin.

Finally, if you are not using *gdb* to load and run the program, but wish to load a program directly into the MIPSSim simulated ROM using the `APP_FILE` setting in the MIPSSim configuration file, then remember to use the `.relf` file, not the original ELF file.

3.1.1.5 Non-standard MIPSSim™ Configurations

By default GDB will dynamically create a MIPSSim CPU configuration file to match your selected CPU type. It does this from a template stored in file `<install_top_dir>/share/mipssim.cfg`.

While this will a sufficient MIPSSim configuration to get you going, if you later need to change any of the CPU or memory parameters, or add new device or CorExtend libraries, then you'll need to create your own MIPSSim configuration file.

You can do this using either the MIPSSim GUI, supplied as part of the MIPSSim package, or by using a simple text editor. Full details of the configuration file format are contained in the MIPSSim documentation. The crucial configuration settings which you must change from the defaults supplied with the MIPSSim package are as follows:

Table 3.1 MIPSSim Configuration Settings

File	Description
`APP_FILE`	Must be blank, or commented out.
`DUMP_FILE`	Must be blank, or commented out.
`BIG_ENDIAN`	To avoid a warning message set this to match your program's endianness.
`TRACE_FILE`	In v4.x of the MIPSSim simulator, just setting this will cause a trace log to be written to that file. You may not want to do that for normal debugging, since it will slow down the simulator.

You then have to tell *gdb* and the MIPSSim library how to find the configuration file which you just created, either:

1. Set the `GDBMIPSSIMCONFIG` environment variable to the name of the file, e.g.

   For bash, ksh, etc:

   ```
   export GDBMIPSSIMCONFIG=/path/to/myconfig.cfg
   ```

   For csh and tcsh:

   ```
   setenv GDBMIPSSIMCONFIG /path/to/myconfig.cfg
   ```

2. Or set it in the local `gdbinit` file as follows:

   ```
   set mdi configfile /path/to/myconfig.cfg
   ```

It may be that you are happy to use GDB's default CPU core configuration file, but want to define a new device configuration file with more realistic memory timings, or new device models. GDB will add a reference to your device configuration file to its auto-generated core configuration file if you do one of the following:

1. Set the `GDBMIPSSIMDEVCFG` environment variable to the name of the device configuration file.

2. Or set it in the local `gdbinit` file:

   ```
   set mdi devcfgfile /path/to/mydev.cfg
   ```

3.1.2 MDI Debugging with an EJTAG Probe

MIPS Technologies is encouraging EJTAG probe manufacturers to offer an MDI interface to their devices. This provides a powerful way to debug system software using *gdb* at the lowest level, directly controlling the CPU core.

3.1.2.1 Configuring your Probe for GDB

1. First follow the installation instructions supplied with your probe hardware, and check that you can access and control your CPU core via the probe vendor's own command-line debug tool.

2. Set the environment variables so that gdb can reach the link library for the probe:

For Linux/unix:

For bash, ksh, etc:

```
$ export LD_LIBRARY_PATH=/path/to/mdi/link/library:$LD_LIBRARY_PATH
$ export GDBMDILIB=library_filename
```

For csh and tcsh:

```
$ setenv LD_LIBRARY_PATH /path/to/mdi/link/library:${LD_LIBRARY_PATH}
$ setenv GDBMDILIB library_filename
```

For MS Windows:

You'll need to make sure that the directory containing the probe's MDI DLL has been added to your PATH variable, or copy the DLL to \windows\system on Win9x, or \windows\system32 on Win32 systems (WinNT and above).

There are multiple ways to set the environment variables:

Right Click `MyComputer->Properties->Advanced->Environment Variables`

OR if you're using cgywin:

```
$ export PATH=/path/to/mdi/link/library:$PATH
```

OR in a windows shell:

```
$ set PATH=yourdrive:\path\to\mdi\link\library;%PATH%
```

You'll also need to set the GDBMDILIB environment variable:

```
$ export GDBMDILIB=library_filename
```

The DLL filename is most likely jnetfs2mdilib.dll.

3. Now you can select your probe configuration and run *mips-sde-elf-gdb*, for example:

```
$ mips-sde-elf-gdb helloram
(gdb) target mdi
```

3.1.2.2 Selecting the EJTAG CPU

EJTAG probes connected by USB or parallel port probably support only one CPU at a time - the one to which it is currently connected. In that case you can probably connect to the probe without having to specify an MDI device number.

But with some probes you may have to tell their MDI interface the name of the CPU, or the probe's Ethernet address, or some such. This selection can be made following exactly the same procedure described for selecting a MIPSSim CPU type. described in Section 3.1.1.2 “Selecting the MIPSSim™ CPU”.

3.1.2.3 Building for an EJTAG-connected Target

Select an appropriate value of `SBD` which most closely matches your chosen CPU family and evaluation board. This will have either the "`MALTA`" or "`SEAD`" prefix, but crucially it will have the "`J`" suffix, which indicates that the run-time system is configured to perform console and file i/o via MDI, rather than using the YAMON i/o system.

Now you can build one or more of the SDE example programs and run them on your target board, for example:

1. Change directory to the "hello world" example program:

   ```
   $ cd $SDETOP/examples/hello
   ```

2. Build the example:

   ```
   $ make SBD=MALTA32LJ
   ```

3. You can run the program in command-line mode:

   ```
   $ mips-sde-elf-gdb helloram
   (gdb) set mdi connectreset 7
   (gdb) target mdi
   (gdb) load
   (gdb) run
   ...
   (gdb) quit
   ```

3.1.2.4 Resetting the CPU

When you connect to a remote CPU via an EJTAG probe to download and run your program, you may want to simultaneously reset the CPU to ensure that it always starts in a known good state. However on many evaluation boards the reset signal will also reset the memory controller, which will prevent you (and *gdb*) from accessing DRAM until it has been programmed.

Rather than teaching *gdb* how to initialise your memory controller, the simplest thing to do is allow the onboard PROM monitor (e.g. the YAMON monitor) to run just long enough to program the memory controller, and then halt the CPU so that *gdb* can take control. This behaviour is controlled by *gdb*'s "`mdi connectreset`" setting, which can have the following values:

- Off is the default value, and in this case *gdb* does not try to reset the remote CPU, it simply halts it. Note that for this to work you may need to modify your probe software's configuration files to prevent it from automatically resetting the CPU.

- On In this case gdb will reset the CPU and then halt it immediately. You shouldn't use this unless your memory controller automatically resets into a usable state, or you are willing to use *gdb* commands to program it manually.

- N In this case *gdb* will reset the CPU, allow it to run for N usually sufficient to allow the YAMON monitor to initialise the board.

You can effect this setting in a number of different ways:

1. Set it in the local.`gdbinit` file as follows:

```
set mdi connectreset 7

set mdi connectreset on
```

2. Or set the `GDBMDICONNRESET` environment variable:

For bash, ksh, etc:

```
export GDBMDICONNRESET=7
export GDBMDICONNRESET=0      # On
```

For csh and tcsh:

```
setenv GDBMDICONNRESET 7
setenv GDBMDICONNRESET 0
```

3. Or you can set it each time you enter the "target" connect command, by appending "`,rst=N`" to the device number. For example:

```
(gdb) target mdi 1,rst=7
```

3.1.3 MDI Debugging Tips

3.1.3.1 Command line arguments

If your application has been linked with the standard SDE run-time system, then you can pass command-line arguments to your application (via argc and argv) when debugging via MDI:

1. When using the *gdb* command-line interface, append the arguments to *gdb*'s "run" command, or set the *gdb* "args" variable. See the [Gdb] reference manual for more details.

3.1.3.2 MDI Host File I/O

If your application has been linked with the standard SDE run-time i/o system, then console and file i/o requests will be passed via the MDI interface to *gdb*. You can see your program's output in *gdb*'s console window. If your program attempts to read from its console, then you can input text through *gdb*'s console window when you see the "`app>`" prompt. Your program can also read and write files on your host computer.

Beware that this i/o mechanism will not work if you don't use *gdb* to load and run your program; for example if you load the program directly into MIPSSim using the `APP_FILE` setting in the MIPSSim configuration file. In such cases you must find some other way to perform console and file i/o, such as via an additional MIPSSim device which you provide.

3.1.3.3 MDI Variables and Commands

The MDI interface adds a number of new *gdb* variables and commands which provide finer grain control over the MDI library and its attached CPU than would normally be available with remote *gdb* targets.

```
set mdi stepinto
```

When set to on an MDI single-step will always execute exactly one instruction - if an interrupt or exception occurs then execution will stop with the PC pointing to the start of the exception handler. In environments where interrupts are occurring faster than the time it takes to step through the interrupt handler, it may not be possible to make any progress in the foreground application in this mode.

When off, a single-step will always execute one instruction in the foreground application, ignoring asynchronous interrupts. This may be implemented simply by disabling interrupts globally while single-stepping.

The variable defaults to off.

`set mdi threadstepall`

Selects simultaneous TC stepping mode when scheduler locking is enabled. When on, all TCs are stepped together, otherwise single-stepping only enables execution in the selected TC. Defaults to off.

`set mdi continueonclose`

When set, the target will be told to restart CPU execution when *gdb* closes its MDI connection. If off, then the target will be reset when the connection is closed. Defaults to on.

`set mdi rununcached`

If on then the program's start address is forced to an uncached address, since it may need to initialise the caches before trying to execute code. When false the start address is not changed. Defaults to on.

`set mdi waittime`

Sets the number of milliseconds which MDI should wait before returning a result to *gdb*, when waiting for the run/halt state of the CPU to change. Some MDI libraries ignore this. It defaults to 10ms.

`set mdi library` *`NAME`*

The name of the MDI DLL to connect to. Initialised to the value of the `GDBMDILIB` environment variable, if available.

`set mdi configfile` *`NAME`*

The name of the MIPSSim CPU configuration file. Initialised to the value of the `GDBMIPSSIMCONFIG` environment variable, if available. See Section 3.1.1.5 “Non-standard MIPSSim™ Configurations”.

`set mdi devcfgfile` *`NAME`*

The name of the MIPSSim device configuration file. Initialised to the value of the `GDBMIPSSIMDEVCFG` environment variable, if available. Section 3.1.1.5 “Non-standard MIPSSim™ Configurations”

`set mdi target` *`TARGNUM`*

The MDI target group number to connect to. Defaults to the value of the `GDBMDITARGET` environment variable, if available.

`set mdi device` *`DEVNUM`*

The MDI device number to connect to. Defaults to the value of the GDBMDIDEVICE environment variable, if available.

`show mdi devices`

Displays a list of the available MDI target groups and devices. The MDI DLL library name must be known before this will work.

`set mdi prompt`

Sets the prompt to use when the application program requests console input. Defaults to "app>".

`set mdi asid auto|off|on|`*`ASID`*

Controls which address space to use when accessing mapped virtual addresses through the TLB, and for qualifying breakpoints. When set to "off" it uses the global address space; when "on" it uses the current ASID value in the CPU's *EntryHi* register; when "auto" it uses the global address space for unmapped address, and the current ASID for mapped addresses; otherwise it must be an explicit numeric ASID (0 to 255). Defaults to "auto". Breakpoints use the same setting to qualify the breakpoint request, which on certain targets may allow breakpoints to be triggered only when executed by a specific ASID.

`show mdi tlb `*`INDEX`*`]`

Displays the contents of the TLB. *`INDEX`* is an optional TLB index, else the whole TLB is displayed.

`set mdi tlb `*`INDEX HI LO0 LO1 MASK`*

Programs the *`INDEX`*'th entry in the TLB using the values *`HI`*, *`LO0`*, *`LO1`* and *`MASK`*.

`show mdi cp0 `*`REG`*`[`*`BANK`*`]`

Displays arbitrary Coprocessor 0 registers which are not normally accessible via *gdb*. The argument *`REG`* is the register number; /*`BANK`* is the optional bank number, default 0.

`set mdi cp0 `*`REG`*`[/`*`BANK`*`] `*`VALUE`*

Sets arbitrary Coprocessor 0 registers which are not normally accessible via *gdb*.

`show mdi icache|dcache|scache `*`ADDRESS`*` [, `*`SET`*`]`

Displays the contents of one line in the CPU's primary instruction, primary data or secondary cache. The *`ADDRESS`* argument is a byte offset into the cache, and *`SET`* is the cache set. Note that *`SET`* is optional, and if present a comma is required as separator between the two arguments; if absent then all sets at that cache offset are displayed.

This command has the side-effect of setting *gdb* internal variables $ctag, $cparity, $cdata0, $cdata1, etc to the values displayed. If multiple sets are displayed, then only the highest numbered set is recorded in these variables.

`set mdi icache|dcache|scache `*`ADDRESS, SET, TAG, PARITY, DATA, ...`*

Sets the contents of one line in the CPU's primary instruction, primary data or secondary cache, using the values provided. Note that a comma is required as separator between the values.

`set mdi connectreset on|off|N`

See Section 3.1.2.4 “Resetting the CPU”.

`set mdi gmonfile` *`NAME`*

Sets the file name to which *gdb* will write gprof profiling data, when enabled. The default file name is "`gmon.out`".

`set mdi connecttimout` *`N`*

The number of seconds for which *gdb* will wait for a target to halt execution when first connecting to it. The default is 1 second, set to 0 for unlimited timeout. GDB may be safely interrupted while it is waiting for the halt to complete.

`set mdi gmonfile` *`NAME`*

Sets the file name to which *gdb* will write *gprof* profiling data, when enabled. The default file name is "`gmon.out`".

`set mdi profile`

If set to "on", and you are using the MIPSSim simulator, then *gdb* will tell the simulator to collect profiling information which *gdb* will write to `gmonfile` when the program exits. If set to "auto", then *gdb* will automatically collect and output the profiling data, but only if your program contains the `_mcount` symbol, which will be the case if your program was compiled with profiling enabled. The default is "auto".

`set mdi profile-cycles`

If set then, if MIPSSim profiling is enabled, *gdb* will tell the simulator to count cycles rather than instructions. This will only work of your MIPSSim software is licensed for cycle counting. Defaults to off. This can also enabled using the "`mdi cycles enable`" command, described below. In MIPSSim 4.0 and above you select whether you want cycle counting or not by the MDI device which you connect to - this setting will have no effect.

`set mdi profile-mcount`

If set then *gdb* includes the `_mcount` function in the profile data. Defaults to off, which doesn't profile `_mcount`.

`set mdi mcount-symbols` *`SYM`* `...`

A list of symbol names in the executable which may label the function which is used to collect call-graph profile data, and should b excluded from the profile data unless `mdi profile-mcount` is set. Defaults to "`_mcount`".

`set mdi ftext-symbols` *`SYM`* `...`

A list of symbol names in the executable which may define the start of the executable code segment, for profiling. Defaults to "`_ftext`".

`set mdi etext-symbols` *`SYM`* `...`

A list of symbol names in the executable which may define the end of the executable code segment, for profiling. Defaults to "`_ecode _etext`".

`set mdi logfile` *`NAME`*

Name of a file in which to store a trace of calls made to the MDI library, for troubleshooting. Requires that GDB's `debug remote` is set to 1 or 2. It must be set before issuing the `target mdi` command.

`mdi cacheflush`

Causes dirty lines in the CPU data cache to be written to memory, and then invalidates all CPU caches.

`mdi cycles enable`

Enable MIPSSim cycle counting, if licensed. From this point on *gdb*'s `$cycles convenience` variable will be set to the current cycle count.

By using the command `display $cycles` you can then see how many cycles have been used as you step through your code. In MIPSSim 4.0 and above you select whether you want cycle counting or not by the MDI device which you connect to - this command has no effect.

Also in MIPSSim 4.0 the counter includes the cycles required to flush the pipeline when an MDI breakpoint or single-step causes execution to stop, and to restart the pipeline when resuming execution. So there will be an overhead per breakpoint or step command which you will need to subtract.

`mdi cycles clear`

Clears the MIPSSim cycle counter to zero, and then enables cycle counting.

`mdi cycles disable`

Disables MIPSSim cycle counting. Has no effect with MIPSSim 4.0 and above.

`mdi cycles status`

Reports on whether MIPSSim cycle counting is available, and if so whether it is enabled or disabled.

`mdi reset` *`WHAT`*

It may sometimes be useful to start over from the reset vector when debugging system firmware. The optional argument can be one of the following:

`full`

Reset the entire target system, if possible. This is the default action if no argument is given, and is often the only action supported by the hardware. The CPU will exit the reset state and halt before fetching the first instruction from the memory location at the reset vector.

`device`

If the device consists of a CPU plus peripherals, reset both if possible.

`periph`

If the device consists of a CPU plus peripherals, reset just the peripherals if possible.

`cpu`

If the device consists of a CPU plus peripherals, reset just the CPU if possible.

`mdi regsync`

Forces *gdb* to write back any modified register values to the target CPU. Normally this only occurs when *gdb* is about to restart execution of the application.

`monitor` *`COMMAND...`*

Sends the command line to the MDI library's "do command" interface. The command line is not interpreted by *gdb.*

3.1.3.4 MDI Troubleshooting

If your MDI-connected probe or simulator appears to be misbehaving then it will help us to help you if you collect a log file which shows the MDI calls which occur between GDB and the MDI library. You may be able to work out what's going wrong for yourself, by looking at this file, but if not then please send it to us along with a log of your GDB session.

You can create a log file by switching on remote debug mode **before** issuing the target command, and then repeating whatever commands cause your problem, e.g.

```
(gdb) set mdi logfile mdilog.txt
(gdb) set debug remote 2
(gdb) target mdi
...
(gdb) quit
```

3.2 Debugging with MIPS® MT ASE

To better understand the rest of this chapter, it will help if we first describe a couple of the fundamental terms defined in the MIPS MT (multi-threading) ASE:

- T*hread Context (TC)*: The hardware state necessary to support a single thread of execution within a multi-threaded CPU device. This includes a set of general purpose registers, multiplier registers, a program counter (PC) and a small amount of privileged state.

- *VPE*: A virtual processing element (VPE) is an instantiation of the full privileged CPU state on a multi-threaded CPU, sufficient to run an independent per-processor OS image - it can be thought of as a virtual CPU. Each VPE must have at least one TC bound to it in order to execute instructions and be debuggable, but it may contain more than one TC when running an explicitly multi-threaded OS or application. A conventional single-threaded CPU could be thought of as implementing a a single TC bound to a single VPE.

These components of the MT ASE may be used within a variety of programming models, with different debugging methodologies:

- *LLMT*: Low-Level Multi-Threading (LLMT) describes programs which make explicit use of the hardware TCs to run multi-threaded code, but generally with no more software threads than there are hardware TCs. This may be a self-contained threaded application, or a simple RTOS kernel. When debugging such software you will want

to track the behaviour of the hardware TC states as they execute threads within your program. See Section 3.2.1 “Debugging LLMT Applications” below.

Be aware that LLMT debugging only provides visibility of threads which are assigned to hardware TCs. Neither the probe, simulator, nor GDB have the OS-specific knowledge to find and interpret a stored thread context in target memory. So if you are debugging a threaded application which has more threads than TCs to run them, presumably with a micro-kernel or RTOS to context switch the threads between TCs, then you will need to use the debugging facilities or thread-aware remote debugging protocol provided by the OS to debug application-level threads. The LLMT model may however be used to bring-up and debug the kernel.

- *AP*/*RP*: The term AP/RP describes a programming model where the VPEs are treated as independent loosely coupled cores, with one acting as "Application Processor" (AP), running a complex operating system such as Linux; while the other acts as the "Realtime Processor" (RP), running dedicated real-time code without interference from the AP operating system's scheduling and interrupt handling. This mechanism has sometimes been known as AP/SP and AMVP.

  Debugging an application program on the AP side use the standard OS application debugger, but debugging a program running on the RP side requires the *mips-sde-elf-gdb* debugger, with something like a remote serial or EJTAG probe connection to the RP VPE, as described in Section 3.2.3 “Debugging AP/RP Applications” below.

- *SMVP*: Describes the execution of a largely unmodified symmetric multi-processing (SMP) operating system by multiple VPEs. In this environment application programs - including multi-threaded applications - will be debugged using the OS's usual debugger, and the underlying MT hardware will typically be "invisible" to the application programmer. See Section 3.2.4 “Debugging SMVP/SMTC Programs”.

- *SMTC*: An extension of the SMVP model which requires more significant modifications to an SMP operating system, so that it can schedule multiple software threads and/or processes to run on the hardware TCs. As with SMVP, the OS's normal application debugger will typically be used for debugging threaded applications; for kernel debugging the LLMT model may be applicable.

3.2.1 Debugging LLMT Applications

When you debug low-level or operating system kernel code which makes explicit use of hardware TCs (the "LLMT" model described above) you can use *mips-sde-elf-gdb* in conjunction with an MDI library that supports the multi-threading extensions. At the time of writing this means a recent version of 34K MIPSSim or the FS2 EJTAG probe. The following section assumes that you have successfully connected GDB to your target via a suitable MDI library, as described in Section 3.1 “MDI Debugging”.

When you connect *mips-sde-elf-gdb* to a MT-capable CPU via an MT-aware MDI library, then the hardware TCs can be accessed using GDB's thread debugging facilities - for full details of these commands see the "Debugging programs with multiple threads" section of the GNU GDB manual (supplied as HTML and PDF with the toolchain). To illustrate how these facilities map onto hardware TCs, the critical features are also documented below.

3.2.1.1 Thread Status

Whenever execution stops and control returns to GDB, the debugger will display which TCs have been activated or deactivated since the last prompt and, if it has changed, the name of the current thread. One thread is always the "thread of interest" to which all GDB commands will apply by default, and this is the "current thread". When GDB first regains control from the application the current thread will be the TC which hit the breakpoint, or completed a single-step. In the case of asychronous stop (e.g. typing Ctrl-C in the command line GDB) then any TC may be chosen as the current thread.

You can display a list of all active TCs, and their program counters within the program, as follows:

```
(gdb) info thread
   2 Thread Context 4 in client_thread()
 * 3 Thread Context 2 in server_thread()
   ...
```

Note that there are two numbers on each line: first GDB's thread number, and secondly the hardware Thread Context (TC) number. All of the GDB thread commands work in terms of GDB thread numbers (the first number), not the hardware TC numbers (the second number).

So this particular example tells you that GDB's thread number 2 corresponds to hardware TC 4, and its program counter is within the `client_thread()` function; while GDB's thread number 3 corresponds to TC 2, and its program counter is within the `server_thread()` function. Thread 3 (TC 2) is marked with an asterisk to indicate that it is the current thread.

3.2.1.2 TC-specific Breakpoints

You can set an breakpoint that will only "trigger" when executed by a specific TC simply by appending the `thread` qualifier to a breakpoint command. For example:

```
(gdb) b send_message thread 2
```

This will set a breakpoint in the `send_message` function to be activated only when executed by "client" thread (TC 4) listed above. Beware that a software breakpoint exception will be taken by every TC which executes the breakpoint instruction, requiring communication between GDB and the target. GDB will quietly step over breakpoints which occur in the wrong TC, but performance will be substantially reduced.

It is not currently possible to specify a TC-specific hardware data watchpoint. A hardware watchpoint set up using GDB's `watch`, `rwatch` or `awatch` commands will trigger when any TC bound to the same VPE accesses that location in the specified manner.

3.2.1.3 Thread-specific Commands

You can switch GDB from one TC to another using the `thread` command, or its alias `t` e.g.

```
(gdb) thread 2      # switch to "client" thread
(gdb) bt            # do stack trace of "client"
(gdb) t 3           # switch to "server" thread
(gdb) info reg      # display registers of "server"
```

Alternatively you can perform the same operation on a number of threads at once, e.g.

```
(gdb) thread apply 1 2 7 4 bt         # apply backtrace cmd to threads 1,2,7,4
(gdb) thread apply 2-7 9 p foo        # apply p foo cmd to threads 2->7 & 9
(gdb) t apply all x/i $pc          # apply x/i $pc cmd to all threads
```

3.2.1.4 Resuming threaded execution

Normally when you issue a single-step command there is no guarantee which TCs will run in which order - they might even hit a breakpoint before your single-step request completes, "seizing the prompt" away from your original thread of interest. A sequence of single-step commands may switch you back and forth between your active TCs, or just advance the highest priority one.

To avoid this happening while single stepping you may disable execution of all other TCs on the same VPE, apart from the currently selected TC, by using this command:

```
(gdb) set scheduler-locking step
```

But beware that this can get you into situations where the TC which you are stepping cannot make any progress, because it is waiting for a semaphore or mutex to be unlocked by another TC - so it is not always the most appropriate behaviour.

Furthermore this command:

```
(gdb) set scheduler-locking on
```

will prevent other TCs on the same VPE from running in all cases, even when you resume execution using commands like `continue`, `until` or `finish`.

Finally, rather than locking out the other TCs altogether, you can request that all TCs should "gang step" together. This requires both GDB `scheduler-locking` and "`mdi threadstepall`" to be set. For example

```
(gdb) set scheduler-locking on
(gdb) set mdi threadstepall on
```

In summary:

Schedule-locking	MDI threadstepall	Single-step Behavior
`off`	`X`	Current TC single-steps, all other TCs run freely until the current TC completes an instruction, or one of the other TCs hits a breakpoint
`on|step`	`off`	Current TC single-steps, all other TCs are suspended
`on|step`	`on`	All TCs single-step together - the first to complete an instruction returns GDB to command mode, selected as the current thread

3.2.2 Debugging Multiple VPEs

Debugging multiple VPEs within a multi-threaded core is very similar to debugging multiple independent CPUs within a multi-core system.

3.2.2.1 Multiple VPEs with FS2 probe

The rest of this section assumes that you have installed, configured and selected EJTAG probe software as your current MDI target, as described in Section 3.1.2 “MDI Debugging with an EJTAG Probe”. For reliable multi-VPE debugging it is recommended that you use version 2.1.6.7 or higher of the FS2 probe software.

When you start GDB with the probe connected to a 34Kc core, you should see something like this in response to the `show mdi devices` command:

```
(gdb) show mdi devices
Targ 01: mips-single-core
    Dev 01: mips-single-core-root
Targ 02: mips-dual-cores
    Dev 01: mips-dual-cores-mips1
    Dev 02: mips-dual-cores-mips2
Targ 03: mips-34k
```

```
Dev 01: mips-34k-vpe1
Dev 02: mips-34k-vpe0
```

If that works as described, then you should now be able to connect to VPE0. Assuming the same numbering as above then use the "target mdi 3:2" command, which should result in output as follows (the last line, especially the address reported, will vary):

```
(gdb) target mdi 3:2
Selected device mips-34k-vpe0 on MIPS unknown
[New Thread Context 0]
Connected to MDI target
0x8010049c in ?? ()
```

You are now set up and ready for debugging.

General VPE Debugging with Probe

To access a VPE within an multi-threaded CPU the appropriate target group number must be used (e.g. target group 3 in the example above). Within that group the device number 1 corresponds to VPE1 and the device number 2 corresponds to VPE0. Thus to attach to VPE0 you need to use "target mdi" and then select group number 3 and then device number 2 interactively, or alternatively use "target mdi 3:2". Likewise for VPE1 you could use "target mdi 3:1". A given VPE can only be usefully connected to if it has at least one thread context (TC) bound to it. Therefore with the default configuration, VPE0 can be controlled straight from RESET, but VPE1 can only be once some code has been run to bind a TC to it. However, GDB may be attached to a disabled VPE and it will keep waiting until it has been activated.

GDB may be used to load a program to be debugged to the target. A typical session in this case is going to include the following commands in the given order:

```
$ mips-sde-elf-gdb program

    ... start GDB and load program's symbol table
(gdb) target mdi 3:2,rst=7

    ... connect to VPE0, and reinitialise the target
(gdb) load

    ... transfer the program to the target's memory
(gdb) break function

    ... set a software breakpoint on a function
(gdb) run

    ... start execution
```

Note the rst=7 option when connecting to VPE0. That tells GDB to reset the target CPU just after establishing the connection. Execution is then resumed and the target is allowed to run freely for seven seconds, after which it is halted. The intent is to let the firmware (e.g. YAMON) initialize board resources, in particular the caches and memory controller, so that the target can accept a program image. Note that this will reset the whole board and CPU, not just the selected VPE, so it would not make sense to use this option when connecting to VPE1.

Depending on your setup, to load a large program into memory you may either use the probe via the GDB load command, or it may be faster to use the system's firmware. For the latter, and a board like the Malta, that would be

YAMON (see the YAMON manual for how to do this); for other systems it would be system-specific. Sometimes you may be may be debugging the firmware itself. An example session may look like this:

```
$ mips-sde-elf-gdb program
    ... start GDB and load program's symbol table
(gdb) target mdi 3:2
     ... connect to VPE0 - VPE0 is halted
(gdb) break function
    ... set a software breakpoint on a function
(gdb) continue
    ... resume execution of already loaded program/firmware
```

If it's the firmware being debugged, it may sometimes be useful to start over from the reset vector. For this, the "`mdi reset`" command may be useful - this resets the target system entirely. The CPU will exit the reset state and stop before fetching the first instruction from the memory location at the reset vector. You normally really want to issue this command from a debugger connected to VPE0 as VPE1 will become inactive as a result.

When a debugging session is terminated, the VPE can either be left halted or execution may be resumed, therefore letting code that has been *previously* debugged run freely. Use the following commands to control that behaviour:

```
(gdb) set mdi continueonclose on
   ... to resume execution
(gdb) set mdi continueonclose off
   ... to keep the target halted
(gdb) show mdi continueonclose
   ... to retrieve the current setting
```

3.2.2.2 Multiple VPEs on the MIPSSim™ Simulator

The rest of this section assumes that you have installed, configured and selected the MIPSSim simulator as your current MDI target, as described in Section 3.1.1 “MDI Debugging with the MIPSSim™ Simulator”. For reliable multi-VPE debugging it is recommended that you use version 4.6.36 or higher of the MIPSSim software.

MIPSsim provides two ways of working with multiple VPEs. One uses a single MDI device to access the whole core: all thread contexts are accessible through a single connection to the device, regardless of the VPE to which they are bound. The other way uses a pair of separate MDI devices where each has access only to thread contexts bound to the corresponding VPE. The second method requires an auxiliary program called **`mipssimd`** that controls internal communication between the two MDI devices - this tool is supplied as part of the MIPSSim package, but is not currently available for Windows hosts.

Setting up mipssimd

Working with **`mipssimd`** requires additional settings to be present in environment variables. They are necessary for the program to create identifiable communication channels with clients connecting to VPE 0 and VPE 1 of the same simulated processor. System V IPC is used. The variables are as follows:

`$MIPS_MDI_IPC_KEY`

defines a file to be used as a key to identify this particular instance of a simulated processor. The file has to exist and be accessible. This variable is also used by GDB to select which instance of **`mipssimd`** to communicate with.

`$MIPS_MDI_IPC_CLIENTS`

defines the number of clients to be handled. For the 34K family this has to be set to "2" for the 2 VPEs the processor implements.

`$MIPS_MDI_IPC_CLIENT_ID`

defines the number of the communication channel to use between the debugger and a single instance of **mipssimd**, starting from '0'. For the 34K this can be either "0" or '1'. This variable is only used by GDB, and each instance of GDB should have a different value.

With `$MIPS_MDI_IPC_KEY` and `$MIPS_MDI_IPC_CLIENTS` set up you should be able to start **mipssimd**. But before that, it's generally a good idea to clean up any leftover state in IPC resources that may have been left from previous **mipssimd** runs. There is a dedicated program included with MIPSsim that does that. To run it, enter the "`mdiipcwatchdog cleanup`" command. You should get output like below (obviously the path to the key file will differ, depending on the value of $MIPS_MDI_IPC_KEY, as may the key and the seed). The following example assumes a Bourne-style shell, for a C shell use the `setenv` command.

```
$ touch /home/joe/.MIPS_MDI_IPC_KEY
$ export MIPS_MDI_IPC_KEY=/home/joe/.MIPS_MDI_IPC_KEY
$ export MIPS_MDI_IPC_CLIENTS=2
$ export MIPS_MDI_IPC_CLIENT_ID=0
$ mdiipcwatchdog cleanup
Destroying shared memory and semaphores.
Generated key '0x1157340' using key string
/home/joe/.MIPS_MDI_IPC_KEY' and key seed 0x1
```

This cleanup step is not required before running **mipssimd** for the first time, but as a side effect it also validates the setup, so running it anyway is a sensible idea.

Now to actually run **mipssimd**, you should see output as follows (again, the path to the key file will likely differ):

```
$ mipssimd -p -f
Starting up......

Establishing connection with debuggers using key
/home/joe/.MIPS_MDI_IPC_KEY'.
Support up to 2 debugging clients
Ready to handle IPC commands from debugger #0.
Ready to handle IPC commands from debugger #1.
```

If this is works, then you are ready to start working with **mipssimd**.

The options given to **mipssimd** above have the following meaning:

`-p` stands for "persistent" and makes **mipssimd** preserve the state of the simulated system between MDI connections

`-f` stands for "forever" and makes **mipssimd** keep running even when the last client disconnects.

The result is to make **mipssimd** behave like a real h/w CPU, allowing multiple debugger connections to be opened and closed, until it is terminated. Once you finish debugging, you may terminate **mipssimd** by sending it the SIGINT signal. It's done in a system-specific way, usually by typing <Ctrl>+<C>, also written as `^C - run "stty -a"` and see the entry marked `intr=` for what character is used in a given system - or by using the shell's `kill` command. With the latter, bear in mind **mipssimd** is multithreaded and all threads must be terminated.

General VPE Debugging with Simulator

MIPSsim provides two target group numbers for the 34K - number 21 is for the instruction-accurate simulator and number 22 is for the cycle-counting version. Within each of the groups six devices are defined as follows:

1	Whole CPU, little-endian
2	Whole CPU, big-endian
3	VPE0, little-endian
4	VPE0, big-endian
5	VPE1, little-endian
6	VPE1, big-endian

Thus to attach to VPE0 of a big-endian, cycle-counting 34K you need to use the "`target mdi`" command and select the group number 22 and then the device number 4 interactively or alternatively use "`target mdi 22:4`". Similarly for a whole CPU access to a little-endian, instruction-accurate 34K you may either select the group number 21 and then the device number 1 or use "`target mdi 21:1`".

GDB may be used to load a program to be debugged to the target. A typical session in this case is going to include the following commands in the given order:

```
$ mips-sde-elf-gdb program
    ... start GDB and load program's symbol table
(gdb) target mdi 21:2
    ... connect to whole 34K, instruction accurate, big-endian
(gdb) load
    ... transfer the program to the target's memory
(gdb) break function
    ... set a software breakpoint on a function
(gdb) run
    ... start execution
```

Normally GDB generates a MIPSSim configuration file on the fly from a template (installed as `<install_top_dir>/share/mipssim.cfg`) and uses this whenever a target is opened. If the default settings are unsuitable, then a custom configuration file may be used. Once such a file has been created, use the following command in GDB to use it instead of the default auto-generated file:

```
(gdb) set mdi configfile filename
```

Sometimes it's useful to start debugging a program that has already been loaded into MIPSSim memory; this can done using the `APP_FILE` setting in a MIPSSim configuration file. An example session may then look like this:

```
$ mips-sde-elf-gdb program
    ... start GDB and load program's symbol table
(gdb) set mdi configfile myconfigfile
    ... select custom configuration file
(gdb) target mdi 22:1
    ... connect to whole 34K, cycle-accurate, little-endian
  ... the simulator loads the application executable file
  ... the device is halted
(gdb) break function
    ... set a software breakpoint on a function
(gdb) continue
    ... resume execution under control of GDB
```

3.2.3 Debugging AP/RP Applications

The mechanism for debugging a program running on the RP side of an AP/RP system is similar to downloading and running a "bare-iron" program on a target board connected by a serial port or network. It is also possible to debug RP programs using an EJTAG probe.

3.2.3.1 Using the SP Debugging Daemon

This mechanism allows debugging of an RP program without use of a h/w EJTAG probe. The remote debug connection is via TCP/IP, with the GDB remote debug protocol transported between *mips-sde-elf-gdb* on the development host, through a network server running on the target CPU's AP Linux VPE, and then via a shared memory FIFO to the RP VPE.

In the current implementation the debug protocol is finally interpreted by a remote debug "stub" which is linked into your RP application, similar to the remote serial debugging of standalone programs described in Section 3.4.2 “Serial Debugging with SDE Debug Stub”.

The following example demonstrates how to debug an RP application running on a Malta board. Debugging an RP application on MIPSSim is not currently possible using this mechanism.

1. Build your application with the **RDEBUG** makefile variable set to `imm`, e.g.

   ```
   devhost$ cd $SDETOP/examples/minimon
   devhost$ make clean
   devhost$ make all RDEBUG=imm SBD=MALTA32LSP
   ```

2. If you haven't already done so, then start the SP debugging daemon on your Malta Linux target:

   ```
   aplinux$ spd &
   ```

3. Open a new connection to your Malta (e.g. using `telnet`, `ssh`, `rlogin`, etc) in another terminal window, and start the AP/RP `rtterm` (real-time terminal) application. This will allow you to communicate with the running RP program's virtual console:

   ```
   aplinux/2$ rtterm
   ```

4. Now transfer the `minirel` program to your Malta board and "download" it into the Realtime Processor by sending it to the `/dev/vpe1` device on the Application Processor Linux host, e.g. in your original window:

   ```
   aplinux$ cat minirel >/dev/vpe1
   ```

5. Now start *mips-sde-elf-gdb* (on your development host) and connect it via TCP/IP to the debug server running on the target board:

   ```
   devhost$ mips-sde-elf-gdb minirel
   (gdb) target remote aplinux:2222
   ```

   In the above *aplinux* represents the network hostname or IP address of your Malta board. GDB automatically determines the load address of the relocatable program, and relocates its symbol table data to match.

6. Now you can set breakpoints and enter the GDB `c` or `continue` command to start the program running under the control of GDB. Don't use the `r` or `run` command to start execution, since this would restart the RP program from its entry point.

7. Note that the `spd` remote debugging daemon does not currently support interrupt requests from GDB, so it is not possible to break into a runaway RP application from GDB by typing Control-C . To diagnose such problems you will need to use other techniques such as breakpoints to find the problem; or use an EJTAG probe, which can interrupt any program, whatever its state.

3.2.3.2 AP/RP Debugging with EJTAG Probe

Refer to Section 3.2.2 "Debugging Multiple VPEs" for general information on using a probe to debug multiple VPEs.

The usual method of debugging an AP/RP Linux kernel with an EJTAG probe is to let the firmware load and start Linux and then attach to VPE0 (the Linux AP) which is already running. Similarly for VPE1 (the RP program), except that the Linux VPE loader is used to load and start the program.

Since the probe firmware does not know the load address of a relocatable RP program, and cannot tell GDB how to relocate its symbol table, it's usually easier to debug a fully linked RP executable (i.e. an executable called "`*ram`" rather than `*rel`). To build such a program, assuming that you have started your Linux kernel with the `memsize=30M` boot option, you would build your program something like this:

```
devhost$ cd $SDETOP/examples/minimon
devhost$ make clean
devhost$ make ram SBD=MALTA32LSP DLBASE_C=81e00000
```

Note that 81e00000 is the KSEG0 (untranslated, cacheable) mapping of the Linux maximum memory size (30MB = 0x1e00000). Also note that the RDEBUG option should not be used when debugging using an EJTAG probe.

If debugging of either AP or RP from the very beginning of the loaded program is required, then hardware execution breakpoints may be placed at the entry point. Use the GDB's `hbreak` for this. It accepts any syntax that is valid for the `break` command; in particular absolute numeric addresses may be specified after an asterisk. As the command uses a hardware breakpoint register in the debug port of the core it has to be issued to the correct VPE and will not affect the other VPE.

If the RP-side VPE to be debugged is inactive, then there is no way to set a hardware breakpoint since on an attempt to connect GDB will stall, waiting for the VPE to become activated. GDB will pass control to the user as soon as the VPE becomes active and before the first instruction of the program has been executed.

AP/RP Team Debugging

Sometimes when doing debugging it may be desired for the VPEs involved to be stopped and resumed synchronously, so that the state of the target system remains as stable as possible during debug accesses. GDB provides a way of doing that by grouping VPEs, and potentially any devices, into the so called teams. While a single instance of GDB can only fully control one device at a time, including the device in a team with other devices makes requests for stopping and resuming be propagated to all of them. If any of the other devices have instances of GDB attached to them, these requests are transparent to their controlling debuggers. Specifically a device in a team that has been stopped by another debugger, but not the controlling one, stops, but continues reporting the running state to the latter. Likewise a device that has been resumed by the controlling debugger starts reporting the running state, but resumes only after all the other debuggers resumed it.

The following commands are used to control teams:

```
target mdi <device>,team=<device>[,team=<device>...]
```

Open the device specified at the beginning and attach it together with ones listed as `"team="` arguments to the currently selected team.

`mdi team attach <device> [<device>...]`

Attach listed devices to the currently selected team.

`mdi team detach <device> [<device>...]`

Detach listed devices from the currently selected team.

`mdi team clear`

Destroy the currently selected team removing all members beforehand, the currently selected team is set to "0".

`mdi team list`

List identifiers and members of the currently existing teams.

`set mdi team <id>`

Select a team identifier for further team operations, "0" means a new team will be created for attachment operations.

`mdi team <id>`

Shorthand for "`set mdi team <id>`".

`show mdi team`

Print the identifier of the currently selected team.

`mdi team`

Shorthand for "show mdi team".

The use of these commands is incompatible with group debugging as described in Section 3.2.4.1 “SMVP/SMTC using MIPSSim® Simulator On MIPSSim”.

AP/RP Debugging with MIPSsim

Note that this feature is not supported by the current release of the AP/RP package for TimeSys Linux.

One way to debug such a setup is to use a custom MIPSSim configuration file to load and run the AP/RP Linux kernel. It can be used straight from GDB using the "`set mdi configfile`" command. In such a setup after opening the target, programs as referred to from the configuration files will have been loaded into MIPSSim memory and may be started just by issuing the `continue` command. Soon you will see Linux kernel messages being output. Depending on whether **`mipssimd`** is used or not, they will appear through **`mipssimd`** or GDB's window. This communication channel is actually the Linux console and once the user mode is reached will accept input as well.

Similarly with VPE1 (RP), the Linux VPE loader is the usual way of starting the program, rather than loading it through the MDI interface. It's usually easier to use a fully linked executable, as described above for the EJTAG probe. Memory space for loading such an executable has to be reserved in the MIPSSim device configuration file - one provided with the Linux AP/RP package may be used as the starting point (see the MIPSSim documentation for anything that is not immediately obvious in the file).

Since the simulator returns control to GDB after loading Linux, the kernel may also be debugged from the very beginning as is - rather than issuing `continue` you may use any commands, like `step` or `break` to set up debugging as required.

If debugging of the RP program from the very start of the loaded program is required, then a hardware execution breakpoint may be placed at the entry point. Use the `hbreak` command of GDB for that. If split per-VPE devices and **`mipssimd`** are used, then `hbreak` has to be issued to the correct VPE and it will not affect the other one. A connection to the VPE1 device has to be made and the breakpoint be set within, which will trigger as soon as VPE1 executes the instruction there.

3.2.4 Debugging SMVP/SMTC Programs

3.2.4.1 SMVP/SMTC using MIPSSim® Simulator On MIPSSim

the use of the "whole CPU" device to debug shared program image operating systems running across multiple VPEs is recommended - refer to Section 3.2.2.2 “Multiple VPEs on the MIPSSim™ Simulator” above. In this case TCs bound to any VPE all become visible and controllable as threads within GDB, as described in Section 3.2.1 “Debugging LLMT Applications” above. All active VPEs also halt and resume execution simultaneously. This is probably what you would expect anyway, when debugging SMP operating systems on a single 34K.

3.2.4.2 SMVP/SMTC using FS2 Probe and Group Debugging

Group debugging allows synchronous control of multiple devices by a single instance of GDB. All thread contexts of all open devices are seen as threads of a single running program. This is most useful for debugging SMP-style execution environments, though it is not strictly required for each of the devices to execute the same code. Internally the devices are synchronised to one another, that is, events causing one device to stop freeze all the other ones and if GDB decides to return control to the user, then all the threads have their state preserved as of the time of the event, subject to hardware or simulator limitations.

The FS2 MDI libraries prior to version 2.1.8.0 do not fully support device synchronization. When using them, GDB still permits doing group debugging, but there is no synchronisation between devices and the state preserved will only be a rough approximation of what the system would look like if a debugger was not attached. This may still be useful for debugging systems which have no strict timing restrictions.

Use the following command to debug a group of devices:

```
target mdi <device>,group=<device>[,group=<device>...]
```

Open all the devices requested at once. The use of this command is incompatible with team debugging as described in the Section “AP/RP Team Debugging”.

Here is an *example* session for SMTC Linux:

```
(gdb) file ./vmlinux
Reading symbols from /home/macro/linux/vmlinux...done.
(gdb) target mdi 2:2,group=2:1,rst=0
Selected device mips-dual-cores-mips2 on MIPS unknown
Selected device mips-dual-cores-mips1 on MIPS unknown
[New Thread Context 2:2:0]
Connected to MDI target
0xbfc00000 in ?? ()
(gdb) continue
Continuing.
```

[*Here Linux is started from YAMON.*]

```
^C
Quit received: Stopping target
[New Thread Context 2:2:0]
[New Thread Context 2:2:1]
[New Thread Context 2:2:2]
[New Thread Context 2:1:3]
[New Thread Context 2:1:4]

Program received signal SIGINT, Interrupt.
[Switching to Thread Context 2:1:3]
0x80101e6c in r4k_wait () at arch/mips/kernel/cpu-probe.c:48
48              __asm__(".set\tmips3\n\t"
(gdb) info threads
5 Thread Context 2:1:4  0x80101e6c in r4k_wait () at arch/mips/kernel/cpu-probe.c:48
4 Thread Context 2:1:3  0x80101e6c in r4k_wait () at arch/mips/kernel/cpu-probe.c:48
3 Thread Context 2:2:2  r4k_wait () at arch/mips/kernel/cpu-probe.c:48
2 Thread Context 2:2:1  r4k_wait () at arch/mips/kernel/cpu-probe.c:48
1 Thread Context 2:2:0  0x8035a5f4 in _spin_unlock_irqrestore (lock=0x80407774,
flags=1024) at kernel/spinlock.c:284
```

Notice how device numbers are reported prefixing thread context numbers above.

3.3 Debugging with the GNU Simulator

You can debug a program using the GNU MIPS simulator which is built into *mips-sde-elf-gdb*. It works very like any other remote debug mechanisms - in fact internally it looks to *gdb* like a remote board.

As supplied the GNU simulator does not simulate i/o devices[1], just a bare MIPS architecture CPU, RAM and a set of PROM monitor entry points. So you can't use the GNU simulator to run programs built for a real hardware target like a Malta board - you must build your programs specifically for the GNU simulator target, e.g. `SBD=GSIM32B`.

You can see your program's output in *gdb*'s console window. If your program attempts to read from its console, then you can input text through *gdb*'s console window when you see the `app> prompt`. Your program can also read and write files on your host computer.

3.4 Remote Serial Port Debugging

If you've got a MIPS Technologies evaluation board such as the Malta or SEAD-2 boards, but you haven't got an EJTAG probe, then you'll probably be debugging your programs using a remote debug protocol over the serial port. You also might need to use serial debugging in other cases, such as when you need to debug a multi-threaded application or RTOS, which requires a debug protocol that can handle software thread contexts - for example MDI can provide access to low-level hardware TCs on a multi-threaded CPU (see Section 3.2 “Debugging with MIPS® MT ASE”), but does not know how to find or interpret the state of a software thread which is not currently assigned to a hardware TC.

GDB Serial Ports

When you connect to a target using a serial (RS232) port, you have to tell *gdb* the name of the port device to use. In the examples which follow we've chosen to use the Linux device name `/dev/ttyS0`, but this is operating system

1. Actually, if you are brave, then it is possible to add device models to the GNU simulator by editing the source.

specific, and you'll have to use different names as appropriate for you host. Table 3.2 gives a list of possible names for different operating systems.

Table 3.2 Host O/S Serial Port Devices

Host	Device Names
Linux	`/dev/ttyS0, /dev/ttyS1`
Windows	`/dev/com1, /dev/com2`
Solaris	`/dev/ttya, /dev/ttyb`

GDB Serial Protocols

There are several different ways that a MIPS program can be debugged remotely, and the distinction often causes confusion.

1. 1) Using the default *gdb* serial remote debug protocol, support for which is built into the YAMON monitor on MIPS Technologies boards, or

2. Again using the default *gdb* serial remote debug protocol, but in this case connecting to the SDE remote debug stub, which can be linked into your program if you are building a rommable or "standalone" program, or

3. Using the historical MIPS Computer Systems remote debug protocol, as implemented in some PROM monitors (e.g. IDT/sim and PMON). But this mechanism is no longer documented in this manual. It is a completely different debug protocol, and requires different commands to get it started.

The amount of data passed back and forth between the board and *gdb* means that some operations can be quite slow at 38400 baud (the YAMON monitor's default speed). You can use *mips-sde-elf-gdb*'s **`-b`** option, its `"set remotebaud"` command, or the Target Selection dialog in the GUI, to raise the serial line speed to 57600 or 115200 baud, if the target board can handle it. Where the host/target link is slow it's quicker to set *gdb* temporary breakpoints (the `tbreak` command) and then `continue`, rather than doing repeated `step` commands. You can also speed things up by enabling *gdb*'s memory transfer cache using the `"set remotecache"` command, but don't do that if you plan to use *gdb* to access device registers or shared memory.

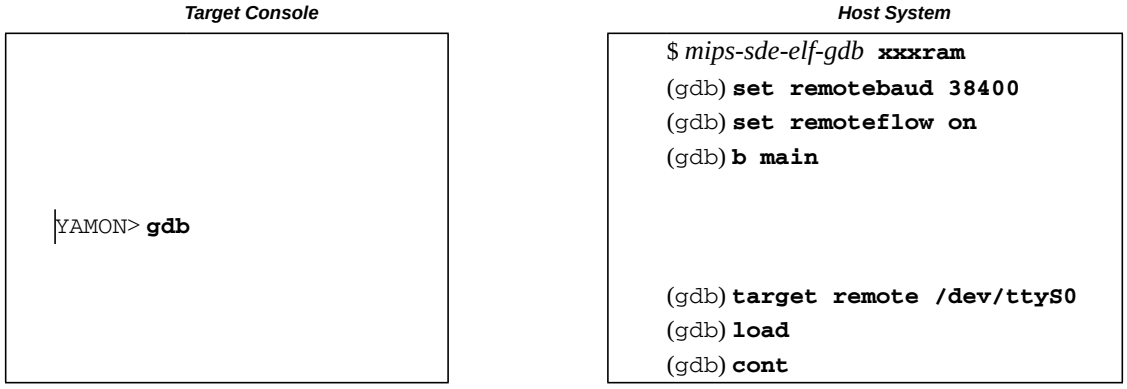
3.4.1 Serial Debugging with the YAMON™ Monitor

The YAMON PROM monitor supplied on MIPS Technologies' Atlas, Malta and SEAD-2 boards implements *gdb*'s default remote debug protocol. The YAMON *gdb* protocol is hardwired to use the board's second serial port (tty1), so you will usually need two serial connections between the host and the board: one connected to a terminal emulator for the console, and one used by *gdb* for the remote debug protocol.

The YAMON monitor runs its serial ports at a default 38400 baud, and in some cases (slow FPGA-based cores) may require hardware flow-control to avoid UART receive buffer overruns. This can be enabled by *gdb*'s `set remoteflow` command, or using the h/w flow control tickbox in the GUI's "File->Target\Settings..." dialog.

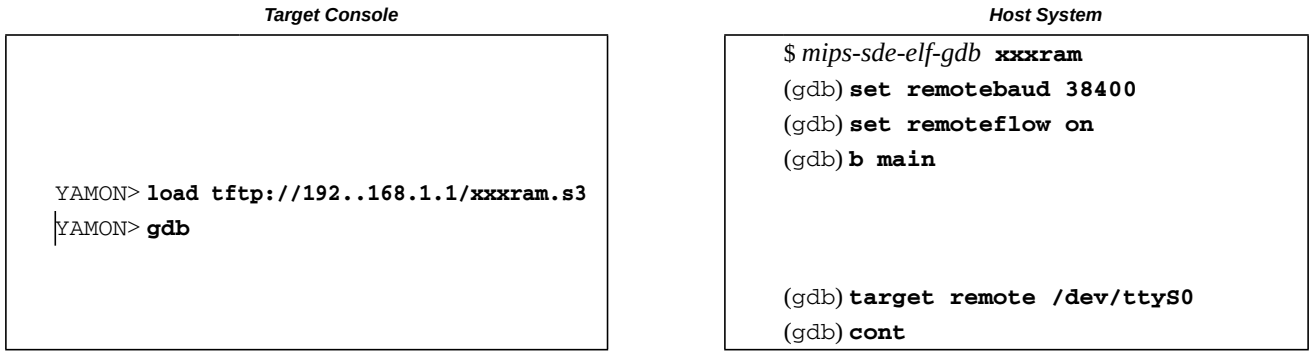
3.4.1.1 YAMON™ Monitor - Serial Download

Follow this example to load a program `xxxram` over a serial port to a board running the YAMON monitor (e.g. built with `SBD=MALTA32L`).



3.4.1.2 YAMON Monitor - TFTP Download

If you have an Ethernet connection to your board and a TFTP server on your host, then you can avoid a long serial download by downloading your program over Ethernet with the YAMON monitor's `load` command, and then starting *gdb* as follows:



To simplify this further you could set the YAMON $start environment variable to run the YAMON `load` and `gdb` commands after every reset.

3.4.2 Serial Debugging with SDE Debug Stub

The SDE run-time system includes a "remote debug stub", which implements the target monitor for *gdb*'s default remote debug protocol. This stub will only be linked into your application if the target board's PROM monitor does NOT include one of the supported remote debug protocols, or if you are building a standalone, rommable or Real-time Processor program. In both cases you must also define the **`RDEBUG`** makefile variable.

N.B. The **`RDEBUG`** variable is ignored when you build a program for a monitor which already supports *gdb* remote debugging. For example MIPS Technologies' YAMON monitor also uses the *gdb* default remote debug protocol, but you should be reading the previous section, which describes YAMON debugging.

Before starting *mips-sde-elf-gdb* you have to start your program running. For a RAM-based program this will mean downloading it to your board, using whatever facilities your board's monitor provides, and issuing some sort of "go" command. For a rommable program this might mean blowing an EPROM or Flash, plugging it into your board, and just switching it on!

Your program will now run until it gets an unexpected exception, at which point it displays a message on its console to indicate that it is waiting for the remote debugger to make contact. On your host system you can now start *mips-sde-elf-gdb* and perform post-mortem diagnosis as follows:

Target Console

```
<start program>
DSE General Exception, reason=...
Cause 00000008
etc.
Awaiting remote debugger...
```

Host System

```
$ mips-sde-elf-gdb xxxrom
(gdb) target remote /dev/ttyS0
(gdb) bt
```

If you want to set breakpoints before the program starts running, then define **`RDEBUG=immed`** when building it. The startup code will then stop and wait for *mips-sde-elf-gdb* just before entering your `main()` function. At this point you can connect *mips-sde-elf-gdb*, as above, set your breakpoints and continue. For example:

Target Console

```
<start program>
Awaiting remote debugger...
```

Host System

```
$ mips-sde-elf-gdb xxxrom
(gdb) target remote /dev/ttyS0
(gdb) b main
(gdb) c
```

Note that most SDE board kits do not support serial port interrupts, so it is not usually possible to interrupt a runaway application from GDB when using the remote debug protocol, e.g. by typing Control-C. To debug such problems you must use other techniques such as breakpoints to find the problem; or use an EJTAG probe which can interrupt any running program, even when interrupts are disabled.

If you wish to use a faster baud rate, then you will need to recompile the board-specific serial-port driver (i.e.`$SDETOP/kit/SBD/sbdser.sx`) with a larger value of the **`DBGSPEED`** constant defined (e.g. in the board's `sbd.mk` or `sbd.h` file). To run the debug protocol down the console port (i.e. sharing a single connection) define **`DBGPORT=0`** in `sbd.mk`; on boards which support a non-volatile environment the same effect can be achieved by setting either the `$dbgport` or `$hostport` board variable to `tty0`.

3.4.3 Serial Comms Fault Finding

If your target board is not quite capable of keeping up with the data rate from the host (which can happen if your UART doesn't have a FIFO), or if some error is occurring in the remote debug protocol code, then *mips-sde-elf-gdb* may run very slowly, or mysteriously time-out the connection. If this happens then you should try switching on serial port logging in *gdb* before issuing the target command, and then repeat whatever commands cause the problem, e.g.

```
(gdb) set remotelogfile log.txt
```

When you close the target connection, the named file will contain a trace of all data sent and received by *gdb*.

You can also try

```
(gdb) set debug remote 1
```

which tells the higher-level remote protocol code to output debug information about its activity.

With the YAMON monitor you can ask the remote end to output a debug protocol log to the console, by starting it up with the **-v** flag, like this:

```
YAMON> gdb -v
```

The debug trace information is naturally somewhat cryptic if you are not familiar with the protocols, but you may be able to identify dropped characters or other problems. If you need to contact us with a debug comms problem, then it will be helpful if you can email the trace information to us.

3.5 Debugging C++

Works as advertised in the GDB manual, so long as you use the default DWARF-2. The alternative “Stabs” format used in previous releases of SDE can also be used, but is deprecated. The DWARF-1 format does not support C++.

Chapter 4

Manual Downloading

After the linker has generated an executable object file you may want to download it manually to a PROM programmer, or an evaluation board.

4.1 Evaluation Board Download

Usually you'll download your code using *mips-sde-elf-gdb* as part of a debugging session, as described in the previous chapter. But sometimes you might need to download your program manually. There are usually two steps:

1. While some evaluation boards have an Ethernet interface which allows them to load object files directly at very high speed, most others require that the object file is first converted into some other format (ASCII or encoded binary). The *mips-sde-elf-conv* program performs the task of converting an executable object file into a number of different formats, including: Motorola S-records, LSI Logic PMON fast format, IDT/sim binary S-records, and Stag PROM programmer binary format. See [Conv] for full option details.

   Remember that the example makefiles automatically generate downloadable files as their final result.

2. Finally you can can perform the download via a serial or parallel port. It may also be possible to use the download features of your favourite terminal emulator, for which consult your board manual.

Note that when you download to an evaluation board, you will usually want the program and its data to be loaded at the load addresses assigned by the linker, so **DO NOT** use *mips-sde-elf-conv*'s **`-p`** (prom) option to create your downloadable file: this is what the example makefiles will do when building the ram and standalone versions of a program, as opposed to the rommable version.

The actual process of downloading to an evaluation board is highly dependent on the board and its PROM monitor.

4.2 PROM Programmer Download

The other situation when manual downloading is required is when blowing a PROM. In this case it is usually necessary for the code and data to be relocated from their linker-assigned addresses into offsets from the start of the ROM. The ROM startup code will then relocate the initialised data, and possibly the code too, from ROM to RAM.

The *mips-sde-elf-conv* **`-p`** (prom) option helps with this. It ensures that ROM resident code and read-only data is placed at its correct offset in the ROM image, and then places the initialised, writable data segment at the next 16-byte boundary following. This supports the behaviour of SDE's default ROM startup code (`romlow.sx`), which copies the initialised data to its final location in RAM before starting your application. *mips-sde-elf-conv* also contains facilities for splitting an object file into horizontal and/or vertical slices, including interleaving, to accommodate dumb programmers (the machines, not the people!).

The example *makefiles* automatically invoke mips-sde-elf-conv with the**`-p`** option when building *rom* versions of the program.

The physical process of downloading to the PROM programmer is device-dependent. You should refer to your PROM programmer's manual for instructions.

4.2.1 Other Techniques

Downloading large programs via a serial port is very slow and tedious. There is no reason why a faster technique cannot be used for downloading the program, and you may want to use some other high-speed mechanism on your own board (e.g. a Centronics parallel interface, a PCI bus, USB, or whatever).

To help with this process you may want to examine the sources of *convert* (aka *mips-sde-elf-conv*) programs in the source code tarball.

Chapter 5

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. Most simple, self-contained programs will port with no difficulty. The easiest approach to porting is as follows:

1. Create a new subdirectory in the `$SDETOP/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. For integer-only programs, copy the *dhrystone* makefile; if it uses any floating point arithmetic, then copy the *whetstone* makefile.

3. Edit the new *makefile* and change the definitions of **`PROG`** and **`SRCS`** to represent your final program name, and the list of object files which make it up. Note that object files have the `.o` extension, not `.obj` or anything else.

4. Check the other *makefile* variables, and in particular check that the **`FLOAT`** variable is set to either `yes` or `ieee` if your program performs any floating point arithmetic.

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. You could test it first with the GNU MIPS simulator. 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).

7. If you want to create a Makefile which is to some extent independent of the flexible, but complex SDE build system, then you can generate a “standalone” Makefile for your own program (or one of the examples), which is customised to your SBD setting. Do this by running **`make SDEmakefile SBD=xxx`**, and then use it by running **`make -f SDEmakefile`**, do this in one of the example directories. You can then edit `SDEmakefile` to customise it for your own project.

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 SDE’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 an MDI-interfaced probe or simulator, or the GNU simulator, or on boards with network hardware and suitably equipped PROM monitors. In other cases, you will have to compile the data into the program.

- *Time and date:* is returned by the ISO / ANSI `time(` function, but can return only the elapsed time on boards without a battery-backed real-time clock chip; on such boards the first call will return zero.

- *Elapsed time:* can be determined on all supported boards with the ISO / ANSI `time()` and `clock()`, or POSIX `gettimeofday()` functions. The `clock()` function is the easiest to use for benchmarking: it returns the elapsed time in units of 1 µs. But note that unlike POSIX, it measures elapsed *real* time, not *cpu* time; in other words, it **does** include time spent waiting for console input/output. Be careful to put calls to `clock()` around computational code only.

- *Signal handling:* is primitive. since the console is polled, the Ctrl-C interrupt **SIGINT** will only be detected while you are performing I/O.

- *POSIX termios functions:* and `ioctl` interface are supported, see the <*sys/termios.h*> header file. The older *termio* and *sgtty* interfaces are not supported.

5.1 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).

  SDE includes an exception catcher and emulator for unalignedloads and stores; you just have to call the function

  ```
  _mips_unaligned_init()
  ```

  at the start of your program to install the handler, or simply define "`FEATURES=unaligned`" if you are using the example makefiles. But it's not fast; *don't* use it for benchmarks, and don't use it for a real application unless the unaligned references are very infrequent.

- *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 uncacheable KSEG1 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 explict "*unsigned*" type qualifiers, or change GCC's default behaviour using its **-funsigned-bitfields** option.

- *Small variables of 8 bytes or less:* They 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).

Appendix A

References

[Sweet99]
See MIPS Run, Dominic Sweetman (of MIPS Technologies), 1999, Morgan Kaufman, ISBN 1-55860-410-3.

We have to give special mention to this comprehensive guide to the MIPS Architecture and programming; firstly because one of us wrote it, and secondly because if you read it carefully enough we'll save time on support work.

[Farq94]
The MIPS Programmers Handbook, Erin Farquhar & Philip Bunce, 1994, Morgan Kaufmann, ISBN 1-55860-297-6.

Example-based programming book aimed at small MIPS-based systems.

[SGI96]
MIPSpro™ Assembly Language Programmer's Guide, Silicon Graphics Inc.

[Kane92]
MIPS RISC Architecture, Gerry Kane and Joe Heinrich, 1992, Prentice Hall, ISBN 0-13-584210-7.

Reference manual to MIPS instructions, focussed on the machine instruction level.

[Kern88]
The C Programming Language (Second Edition), Brian W. Kernighan and Dennis M. Ritchie, 1988, Prentice Hall, ISBN 0-13-110362-8.

Throw away all those cheerfully coloured fat books with big letters and lots of pictures. If you want to program in C you need this and nothing else.

[Lewine91]
POSIX Programmer's Guide, Donald Lewine, 1991, O'Reilly, ISBN 0-937175-73-0.

An introduction to and complete set of manual pages for the POSIX.1 programming interface, of which the SDE run-time system implements a generous subset.

Then there are reference works; we need to put these in, but you won’t read them unless you have to:

[POSIX88]
IEEE Standard 1003.1-1988, Institute of Electrical and Electronics Engineers Inc., 1985.

[ABI]
System V Applications Binary Interface - Revised Edition, Unix System Laboratories, Prentice Hall, ISBN 0-13-877598-2.

[MIPSABI]
System V ABI MIPS Processor Supplement, Unix System Laboratories, Prentice Hall, ISBN 0-13-880170-3.

[ELF]
Understanding ELF Object Files and Debugging Tools, Mary Lou Nohr (Editor), Prentice Hall, ISBN 0-13-091109-7.

[MD00410]
MIPS® SDE for Linux Getting Started Guide, MIPS Technologies, Inc.

The document which describes the SDE toolchain configured for native development Linux/MIPS kernels and applications.

[MD00374]
MIPS32® Architecture for Programmers Volume IV-e: MIPS® DSP Application-Specific Extension to the MIPS32® Architecture, MIPS Technologies, Inc.

[MD00378]
MIPS32® Architecture for Programmers Volume IV-f: MIPS® MT Application-Specific Extension to the MIPS32® Architecture, MIPS Technologies, Inc.

You can't (so far as we know) buy the following GNU manuals, but they're provided as part of the toolchain:

[Binutils]
All the object-code tools except the linker itself, which gets a separate manual [Ld].

[Conv]
The SDE-specific ELF file conversion tool (sde-conv).

[Cpp]
The GNU C pre-processor; only for specialists.

[Gcc]
The compiler manual. Serious users should think about reading this through one time.

[Gdb]
The debugger manual. Probably for reference only.

[Gprof]
The profiler manual. Read this if you're planning to do performance analysis.

[Ld]
The linker manual. Read this if you need to go beyond the tricks used in SDE examples.

[Make]
Read this if you're keen to create makefiles even more exciting than those in the examples.

Appendix B

Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

This document may refer to Architecture specifications (for example, instruction set descriptions and EJTAG register definitions), and change bars in these sections indicate changes since the previous version of the relevant Architecture document.

Revision	Date	Description
1.00	April 8, 2008	• First external version
1.01	April 9, 2008	• changed .../SDE notation to $SDETOP
1.02	April 28, 2008	• predefined macros - some have __ instead of _ • added porting chapter
1.03	May 27, 2008	• More accurate description on how to set up MDI target.
1.04	May 27, 2008	• Compiler chapter Typo in mtune table - missing 34kf1_1
1.05	June 20, 2008	• mtune table - 24k and 24ke replaced by 24kc and 24kec