Embed With GNU

Using the GNU Tools on Embedded Systems

Winter 1996
Draft

Cygnus Support

Table of Contents

1 Supported targets

This document describes programming practices and options for several of the embedded targets supported by the Cygnus Developer's Kit. This document is currently draft status, as the tools themselves are evolving to meet the needs of our customers. It is also incomplete, as new targets are added to our matrix frequently (see section "Overview" in *Release Notes*).

1.1 Hitachi H8/300 targets

1.1.1 What to call the tools

Cross-development tools in the Cygnus Developer's Kit are normally installed with names that reflect the target machine, so that you can install more than one set of tools in the same binary directory.

The target name, constructed with the `--target` option to `configure`, is used as a prefix to the program name. For example, the compiler for the Hitachi H8/300 (called simply `gcc` in native configurations) is called:

```
h8300-hms-gcc
```

Likewise, the Hitachi H8/300-configured GDB is called:

```
h8300-hms-gdb
```

For DOS-hosted toolchains, the tools are simply called by their standard names, e.g., `gcc`, `gdb`, etc.

1.1.2 Compiling on Hitachi H8/300 targets

The Hitachi H8/300 target family toolchain controls variances in code generation directly from the command line.

When you run `GCC`, you can use command-line options to choose whether to take advantage of the extra Hitachi H8/300 machine instructions, and whether to generate code for hardware or software floating point.

Using C++ for the Hitachi H8/300

This special release includes support for the C++ language. This support may in certain circumstances add up to 5k to the size of your executables.

The new C++ support involves new startup code that runs C++ initializers before '`main()`' is invoked. If you have a replacement for the file '`crt0.o`' (or if you call '`main()`' yourself) you must call '`_main()`' before calling '`main()`'.

You may need to run these C++ initializers even if you do not write in C++ yourself. This could happen, for instance, if you are linking against a third-party library which itself was written in C++. You may not be able to tell that it was written in C++ because you are calling it with C entry points prototyped in a C header file. Without these initializers, functions written in C++ may malfunction.

If you are not using any third-party libraries, or are otherwise certain that you will not require any C++ constructors you may suppress them by adding the following definition to your program:

```
int __main() {}
```

1.1.2.1 Compiler options

When you run GCC, you can use command-line options to choose machine-specific details. For information on all the GCC command-line options, see section "GNU CC Command Options" in *Using GNU CC*.

General GCC options

`-mh` Generate code for the H8/300H chip.

`-mint32` Use 32-bit integers when compiling for the H8/300H.

`-g` The compiler debugging option '`-g`' is essential to see interspersed high-level source statements, since without debugging information the assembler cannot tie most of the generated code to lines of the original source file.

Floating point subroutines

The Hitachi H8/300 has no floating point support. Two kinds of floating point subroutines are useful with GCC:

1. Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.
2. General-purpose mathematical subroutines.

   The Developer's Kit from Cygnus Support includes an implementation of the standard C mathematical subroutine library. See section "Mathematical Functions" in *The Cygnus C Math Library*.

1.1.2.2 Predefined preprocessor macros

GCC defines the following preprocessor macros for the Hitachi H8/300 configurations:

Any Hitachi H8/300 architecture:

`__H8300__`

The Hitachi H8/300H architecture:

`__H8300H__`

1.1.2.3 Assembler options

To use the GNU assembler, GAS, to assemble GCC output, configure GCC with the '`--with-gnu-as`' switch (as it is in Cygnus distributions) or with the `-mgas` option below.

General GAS options

`-mgas` Compile using GAS to assemble GCC output.

`-Wa` If you invoke GAS through the GNU C compiler (version 2), you can use the '`-Wa`' option to pass arguments through to the assembler. One common use of this option is to exploit the assembler's listing features. Assembler arguments that you specify with `gcc -Wa` must be separated from each other (and the '`-Wa`') by commas.

`-L` The additional assembler option '`-L`' preserves local labels, which may make the listing output more intelligible to humans.

For example, in the following commandline:

```
$ h8300-hms-gcc -c -g -O -Wa,-alh,-L file.c
```

the assembler option `-ahl` requests a listing with high-level language and assembly language interspersed, `-L` preserves local labels, while the compiler debugging option `-g` gives the assembler the necessary debugging information.

GAS options for listing output

Use these options to enable *listing output* from the assembler (the letters after '`-a`' may be combined into one option, *e.g.*, '`-aln`'):

`-a` By itself, '`-a`' requests listings of high-level language source, assembly language, and symbols.

`-ah` Request a high-level language listing.

`-al` Request an output-program assembly listing.

`-as` Request a symbol table listing.

`-ad` *Omit* debugging directives from the listing.

High-level listings require that a compiler debugging option like '`-g`' be used, and that assembly listings ('`-al`') be requested also.

GAS listing-control directives

Use these *listing-control* assembler directives to control the appearance of the listing output (f you do not request listing output with one of the '`-a`' options, these listing-control directives have no effect):

`.list` Turn on listings from this point on.

`.nolist` Turn off listings from this point on.

`.psize` *`linecount`* `,` *`columnwidth`*
Describe the page size for your output (the default is 60, 200). GAS generates form feeds after printing each group of *`linecount`* lines. To avoid these automatic form feeds, specify `0` as the *`linecount`*.

`.eject` Skip to a new page (issue a form feed).

`.title` Use *`heading`* as the title (this is the second line of the listing output, directly after the source file name and pagenumber) when generating assembly listings.

`.sbttl` Use *`subheading`* as the subtitle (this is the third line of the listing output, directly after the title line) when generating assembly listings.

`-an` Turn off all forms processing.

1.1.2.4 Calling conventions

The Hitachi H8/300 passes the first three words of arguments in registers '`R0`' through '`R2`'. All remaining arguments are pushed onto the stack, last to first, so that the lowest numbered argument not passed in a register is at the lowest address in the stack. The registers are always filled, so a double word argument starting in '`R2`' would have the most significant word in '`R2`' and the least significant word on the stack.

Function return values are stored in '`R0`' and '`R1`'. Registers '`R0`' through '`R2`' can be used for temporary values.

When a function is compiled with the default options, it must return with registers '`R3`' through '`R6`' unchanged.

Note that functions compiled with different calling conventions cannot be run together without some care.

1.1.3 Debugging on Hitachi H8/300 targets

GDB needs to know these things to talk to your Hitachi H8/300:

1. that you want to use one of the following interfaces:

   'target remote', GDB's generic debugging protocol. Use this for the Hitachi low-cost evaluation board (LCEVB) running CMON.

   'target hms', the interface to H8/300 eval boards running the HMS monitor.

   'target e7000', the E7000 in-circuit emulator for the Hitachi H8/300.

   'target sim', the simulator, which allows you to run GDB remotely without an external device.

2. what serial device connects your host to your Hitachi board (the first serial device available on your host is the default).
3. if you are using a Unix host, what speed to use over the serial device.

Use one of these GDB commands to specify the connection to your target board:

target *interface port*
: To run a program on the board, start up GDB with the name of your program as the argument. To connect to the board, use the command 'target *interface port*', where *interface* is an interface from the list above and *port* is the name of the serial port connected to the board. If the program has not already been downloaded to the board, you may use the load command to download it. You can then use all the usual GDB commands.

  For example, this sequence connects to the target board through a serial port, and loads and runs a program called *prog* through the debugger:

```
host$ h8300-hms-gdb prog
GDB is free software and ...
(gdb) target remote /dev/ttyb
...
(gdb) load
...
(gdb) run
```

target *interface hostname*:*portnumber*
: You can specify a TCP/IP connection instead of a serial port, using the syntax *hostname*:*portnumber* (assuming your board is connected so that this makes sense; for instance, to a serial line managed by a terminal concentrator).

GDB also supports:

`set remotedebug` *n*
: You can see some debugging information about communications with the board by setting the `remotedebug` variable.

1.1.4 Loading on specific target architectures

Cygnus supports downloading to H8/300 boards and E7000 in-circuit emulators.

1.1.4.1 Hitachi H8/300 boards

You can use the GDB remote serial protocol to communicate with a Hitachi H8/300 board. See section "The GDB remote serial protocol" in *Debugging with GDB*, for more details. Note that the Hitachi LCEVB running CMON has the stub already built-in.

Use the special GDB command:

```
device port
```

if you need to explicitly set the serial device. The default *port* is the first available port on your host. This is only necessary on Unix hosts, where it is typically something like '`/dev/ttya`'.

The following sample session illustrates the steps needed to start a program under GDB control on an H8/300, using a DOS host. The example uses a sample H8 program called '`t.x`'. The procedure is the same for other Hitachi chips in the series.

1. Hook up your development board. In the full example below, we use a board attached to serial port `COM1`.
2. Call GDB with the name of your program as the argument.

   ```
   gdb filename
   ```

   GDB prompts you, as usual, with the prompt:

   ```
   (gdb)
   ```
3. Use two special commands to begin your debugging session:

   ```
   target hms port
   ```

   to specify cross-debugging to the Hitachi board, and

   ```
   load filename
   ```

   to download your program to the board. `load` displays the names of the program's sections. (If you want to refresh GDB data on symbols or on the executable file without downloading, use the GDB commands `file` or `symbol-file`).

These commands, and `load` itself, are described in section “Commands to specify files” in *Debugging with GDB*.

```
C:\H8\TEST> gdb t.x
GDB is free software and you are welcome to distribute copies
 of it under certain conditions; type "show copying" to see
 the conditions.
There is absolutely no warranty for GDB; type "show warranty"
for details.
GDB 4.15-96q1, Copyright 1994 Free Software Foundation, Inc...
(gdb) target hms com1
Connected to remote H8/300 HMS system.
(gdb) load t.x
.text   : 0x8000 .. 0xabde **********
.data   : 0xabde .. 0xad30 *
.stack  : 0xf000 .. 0xf014 *
```

At this point, you’re ready to run or debug your program. Now you can use all of the usual GDB commands:

`break` Set breakpoints.

`run` Start your program.

`print` Display data.

`continue` Resume execution after stopping at a breakpoint.

`help` Dislay full information about GDB commands.

Note: Remember that operating system facilities aren’t available on your development board. For example, if your program hangs, you can’t send an interrupt—but you can press the RESET switch.

Use the RESET button on the development board:

- to interrupt your program (don’t use `Ctrl-C` on the DOS host—it has no way to pass an interrupt signal to the development board).
- to return to the GDB command prompt after your program finishes normally. The communications protocol provides no other way for GDB to detect program completion.

In either case, GDB sees the effect of a RESET on the development board as a “normal exit” of your program.

1.1.4.2 E7000 in-circuit emulators

You can use the E7000 in-circuit emulator to develop code for either the Hitachi H8/300 or the H8/300H. Use one of these forms of the 'target e7000' command to connect GDB to your E7000:

`target e7000` *`port speed`*
: Use this form if your E7000 is connected to a serial port. The *`port`* argument identifies what serial port to use (for example, 'com2'). The third argument is the line speed in bits per second (for example, '9600').

`target e7000` *`hostname`*
: If your E7000 is installed as a host on a TCP/IP network, you can just specify its hostname; GDB uses `telnet` to connect.

The monitor command set makes it difficult to load large amounts of data over the network without using `ftp`. We recommend you try not to issue `load` commands when communicating over Ethernet; use the `ftpload` command instead.

1.1.5 Further documentation

The following manual provides extensive documentation on the Hitachi H8/300. They are produced by and available from Hitachi Microsystems; contact your Field Application Engineer for details.

H8/300 Microcomputer User's Manual
: Semiconductor Design & Development Center, 1992

1.2 Hitachi SH targets

1.2.1 What to call the tools

Cross-development tools in the Cygnus Developer's Kit are normally installed with names that reflect the target machine, so that you can install more than one set of tools in the same binary directory.

The target name, constructed with the `--target` option to `configure`, is used as a prefix to the program name. For example, the compiler for the Hitachi SH (called simply `gcc` in native configurations) is called:

```
sh-hms-gcc
```

Likewise, the Hitachi SH-configured GDB is called:

```
sh-hms-gdb
```

For DOS-hosted toolchains, the tools are simply called by their standard names, e.g., `gcc`, `gdb`, etc.

1.2.2 Compiling on Hitachi SH targets

The Hitachi SH target family toolchain controls variances in code generation directly from the command line.

When you run `GCC`, you can use command-line options to choose whether to take advantage of the extra Hitachi SH machine instructions, and whether to generate code for hardware or software floating point.

1.2.2.1 Compiler options

When you run `GCC`, you can use command-line options to choose machine-specific details. For information on all the `GCC` command-line options, see section "GNU CC Command Options" in *Using GNU CC*.

GCC options for architecture and code generation

`-g` The compiler debugging option '`-g`' is essential to see interspersed high-level source statements, since without debugging information the assembler cannot tie most of the generated code to lines of the original source file.

`-mshl` Generate little-endian Hitachi SH COFF output.

`-m1` Generate code for the Hitachi SH-1 chip. This is the default behavior for the Hitachi SH configuration.

`-m2` Generate code for the Hitachi SH-2 chip.

`-m3` Generate code for the Hitachi SH-3 chip.

`-mhitachi`
Use Hitachi's calling convention rather than that for `GCC`. The registers '`MACH`' and '`MACL`' are saved with this setting (see Section 1.2.2.4 "Calling conventions," page 13).

`-mspace` Generate small code rather than fast code. By default, GCC generates fast code rather than small code.

`-mb` Generate big endian code. This is the default.

`-ml` Generate little endian code.

`-mrelax` Do linker relaxation. For the Hitachi SH, this means the '`jsr`' instruction can be converted to the '`bsr`' instruction. '`-mrelax`' replaces the obsolete option '`-mbsr`'.

`-mbigtable`
Generate jump tables for switch statements using four-byte offsets rather than the standard two-byte offset. This option is necessary when the code within a switch statement is larger than 32k. If the option is needed and not supplied, the assembler will generate errors.

Floating point subroutines

Two kinds of floating point subroutines are useful with `GCC`:

1. Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.
2. General-purpose mathematical subroutines.

   The Developer's Kit from Cygnus Support includes an implementation of the standard C mathematical subroutine library. See section "Mathematical Functions" in *The Cygnus C Math Library*.

1.2.2.2 Predefined preprocessor macros

`GCC` defines the following preprocessor macros for the Hitachi SH configurations:

Any Hitachi SH architecture:
`__sh__`

Hitachi SH architecture with little-endian numeric representation:
`__LITTLE_ENDIAN__`

Big-endian numeric representation is the default in Hitachi SH architecture.

1.2.2.3 Assembler options

To use the GNU assembler, GAS, to assemble GCC output, configure GCC with the '`--with-gnu-as`' switch (as it is in Cygnus distributions) or with the `-mgas` option below.

General GAS options

`-mgas` Compile using GAS to assemble GCC output.

`-Wa` If you invoke GAS through the GNU C compiler (version 2), you can use the '`-Wa`' option to pass arguments through to the assembler. One common use of this option is to exploit the assembler's listing features. Assembler arguments that you specify with `gcc -Wa` must be separated from each other (and the '`-Wa`') by commas.

`-L` The additional assembler option '`-L`' preserves local labels, which may make the listing output more intelligible to humans.

For example, in the following commandline:

```
$ sh-hms-gcc -g -O -Wa,-alh,-L file.c
```

the assembler option `-ahl` requests a listing with high-level language and assembly language interspersed, `-L` preserves local labels, while the compiler debugging option `-g` gives the assembler the necessary debugging information.

GAS options for listing output

Use these options to enable *listing output* from the assembler (the letters after '`-a`' may be combined into one option, *e.g.*, '`-aln`'):

`-a` By itself, '`-a`' requests listings of high-level language source, assembly language, and symbols.

`-ah` Request a high-level language listing.

`-al` Request an output-program assembly listing.

`-as` Request a symbol table listing.

`-ad` *Omit* debugging directives from the listing.

High-level listings require that a compiler debugging option like '`-g`' be used, and that assembly listings ('`-al`') be requested also.

GAS listing-control directives

Use these *listing-control* assembler directives to control the appearance of the listing output (f you do not request listing output with one of the '`-a`' options, these listing-control directives have no effect):

`.list` Turn on listings from this point on.

`.nolist` Turn off listings from this point on.

`.psize` *`linecount`* `,` *`columnwidth`*
Describe the page size for your output (the default is 60, 200). GAS generates form feeds after printing each group of *`linecount`* lines. To avoid these automatic form feeds, specify `0` as the *`linecount`*.

`.eject` Skip to a new page (issue a form feed).

`.title` Use *`heading`* as the title (this is the second line of the listing output, directly after the source file name and pagenumber) when generating assembly listings.

`.sbttl` Use *`subheading`* as the subtitle (this is the third line of the listing output, directly after the title line) when generating assembly listings.

`-an` Turn off all forms processing.

1.2.2.4 Calling conventions

The Hitachi SH passes the first four words of arguments in registers '`R4`' through '`R7`'. All remaining arguments are pushed onto the stack, last to first, so that the lowest numbered argument not passed in a register is at the lowest address in the stack. The registers are always filled, so a double word argument starting in '`R7`' would have the most significant word in '`R7`' and the least significant word on the stack.

Function return values are stored in '`R0`'. Register '`R15`' has a reserved use. Registers '`R0`' through '`R7`', '`T`', '`MACH`' and '`MACL`' can be used for temporary values.

When a function is compiled with the default options, it must return with registers '`R8`' through '`R14`' unchanged.

The '`-mhitachi SH`' switch makes the '`MACH`' and '`MACL`' registers caller-saved, which is compatible with the Hitachi SH tool chain at the expense of performance.

Note that functions compiled with different calling conventions cannot be run together without some care.

1.2.3 Debugging on Hitachi SH targets

GDB needs to know these things to talk to your Hitachi SH:

1. that you want to use one of the following:

   'target remote', GDB's generic debugging protocol. Use 'src/gdb/config/sh/stub.c' to connect to the SH chip. See section "Using and Porting GNU GCC" in *Using and Porting GNU GCC*.

   'target e7000', the E7000 in-circuit emulator for the Hitachi SH.

   'target hms', the HMS rom monitor on SH and H8/300 boards.

   'target sim', the simulator, which allows you to run GDB remotely without an external device.

2. what serial device connects your host to your Hitachi SH board (the first serial device available on your host is the default).
3. what speed to use over the serial device.

The last two pieces of information are not needed for 'target sim', as the simulator is built in.

Use one of these GDB commands to specify the connection to your target board:

target hms *port*

To run a program on the board, start up GDB with the name of your program as the argument. To connect to the board, use the command 'target hms *port*', where *port* is the name of the serial port connected to the board. If the program has not already been downloaded to the board, you may use the load command to download it. You can then use all the usual GDB commands.

For example, this sequence connects to the target board through a serial port, and loads and runs a program called *prog* through the debugger:

```
host$ sh-hms-gdb prog
GDB is free software and ...
(gdb) target hms /dev/ttyb
...
(gdb) load
...
(gdb) run
```

target hms *hostname*:*portnumber*

You can specify a TCP/IP connection instead of a serial port, using the syntax *hostname*:*portnumber* (assuming your board is connected so that this makes sense; for instance, to a serial line managed by a terminal concentrator).

GDB also supports:

`set remotedebug` *n*
: You can see some debugging information about communications with the board by setting the `remotedebug` variable.

1.2.4 Further documentation

The following manuals provide extensive documentation on the Hitachi SH. They are produced by and available from Hitachi SH Microsystems; contact your friendly Field Application Engineer for details.

SH Microcomputer User's Manual
: Semiconductor Design & Development Center, 1992

Hitachi SH2 Programming Manual
: Semiconductor and Integrated Circuit Division, 1994

1.3 MIPS targets

Cygnus currently supports the IDT/MIPS, both R3xxx and R4xxx.

1.3.1 What to call the tools

Cross-development tools in the Cygnus Developer's Kit are normally installed with names that reflect the target machine, so that you can install more than one set of tools in the same binary directory.

The target name, constructed with the `--target` option to `configure`, is used as a prefix to the program name. For example, the compiler for the IDT/MIPS (called simply `gcc` in native configurations) is called by one of these names, depending on which configuration you have installed:

`mips-idt-ecoff-gcc`
If configured for big-endian byte ordering.

`mipsel-idt-ecoff-gcc`
If configured for little endian byte ordering.

Likewise, the mips-configured GDB is called:

`mips-idt-gdb`

For DOS-hosted toolchains, the tools are simply called by their standard names, e.g., `gcc`, `gdb`, etc.

1.3.2 Compiling on MIPS targets

The MIPS target family toolchain controls variances in code generation directly from the command line.

When you run `GCC`, you can use command-line options to choose whether to take advantage of the extra MIPS machine instructions, and whether to generate code for hardware or software floating point.

1.3.2.1 Compiler options

When you run `GCC`, you can use command-line options to choose machine-specific details. For information on all the `GCC` command-line options, see section "GNU CC Command Options" in *Using GNU CC*.

There are a great many compiler options for specific MIPS targets. The following are those options that can be used on all MIPS targets.

Note: The compiler options '`-mips2`', '`-mips3`' and '`-mips4`' cannot be used on the MIPS R3000.

GCC options for architecture and code generation

`-g` The compiler debugging option '`-g`' is essential to see interspersed high-level source statements, since without debugging information the assembler cannot tie most of the generated code to lines of the original source file.

`-mcpu=r3000`
`-mcpu=`*`cputype`*
Since most IDT boards are based on the MIPS R3000, the default for this particular configuration is '`-mcpu=r3000`'.

In the general case, use this option on any MIPS platform to assume the defaults for the machine type *`cputype`* when scheduling instructions. The default *`cputype`* on other MIPS configurations is '`default`', which picks the longest cycle times for any of the machines, in order that the code run at reasonable rates on any MIPS CPU. Other choices for *`cputype`* are '`r2000`', '`r3000`', '`r4000`', '`r6000`', '`r4400`', '`r4600`', '`r4650`', '`r8000`', and '`orion`'. While picking a specific *`cputype`* will schedule things appropriately for that particular chip, the compiler will not generate any code that does not meet level 1 of the MIPS ISA (Instruction Set Architecture) unless you use the '`-mips2`', '`-mips3`', or '`mips4`' switch.

`-mips1` Generate code that meets level 1 of the MIPS ISA.

`-mips2` Generate code that meets level 2 of the MIPS ISA.

`-mips3` Generate code that meets level 3 of the MIPS ISA.

`-mips4` Generate code that meets level 4 of the MIPS ISA.

`-meb` Generate big endian code.

`-mel` Generate little endian code.

`-mad` Generate multiply-add instructions, which are part of the MIPS `4650`.

`-m4650` Generate multiply-add instructions along with single-float code.

`-mfp64` Select the 64-bit floating point register size.

`-mfp32` Select the 32-bit floating point register size.

`-mgp64` Select the 64-bit general purpose register size.

`-mfp32` Select the 32-bit general purpose register size.

`-mlong64` Make long integers 64 bits long, rather than the default of 32 bits long. This works only if you're generating 64-bit code.

`-G` *num* Put global and static items less than or equal to *num* bytes into the small `.data` or `.bss` sections instead of into the normal `.data` and `.bss` sections. This allows the assembler to emit one-word memory reference instructions based on the global pointer (*gp* or *$28*),instead of on the normal two words used.

By default, *num* is 8. When you specify another value, GCC also passes the '`-G` *num*' switch to the assembler and linker.

GCC options for floating point

These options select software or hardware floating point.

`-msoft-float`

Generate output containing library calls for floating point. The '`mips-idt-ecoff`' configuration of '`libgcc`' (an auxiliary library distributed with the compiler) include a collection of subroutines to implement these library calls.

In particular, this GCC configuration generates subroutine calls compatible with the US Software "GOFAST R3000" floating point library, giving you the opportunity to use either the '`libgcc`' implementation or the US Software version. IDT includes the GOFAST library in their IDT C 5.0 package; you can also order libraries separately from IDT as the "IDT KIT". of how to use GCC to link with the GOFAST library.

To use the '`libgcc`' version, you need nothing special; GCC links with '`libgcc`' automatically after all other object files and libraries.

Because the calling convention for MIPS architectures depends on whether or not hardware floating-point is installed, '`-msoft-float`' has one further effect: GCC looks for subroutine libraries in a subdirectory '`soft-float`', for any library directory in your search path. (*Note*: This does not apply to directories specified using the '`-l`' option.) With the Cygnus Developer's Kit, you can select the standard libraries as usual with '`-lc`' or '`-lm`', because the '`soft-float`' versions are installed in the default library search paths.

> *Warning:* Treat '`-msoft-float`' as an "all or nothing" proposition. If you compile any module of a program with '`-msoft-float`', it's safest to compile all modules of the program that way—and it's essential to use this option when you link.

`-mhard-float`
: Generate output containing floating point instructions, and use the corresponding MIPS calling convention. This is the default.

`-msingle-float`
: Generate code for a target that only has support for single floating point values, such as the MIPS `4650`.

Floating point subroutines

Two kinds of floating point subroutines are useful with `GCC`:

1. Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.

   When you indicate that no hardware floating point is available (with the `GCC` option '`-msoft-float`', `GCC` generates calls compatible with the US Software GOFAST library. If you do not have this library, you can still use software floating point; '`libgcc`', the auxiliary library distributed with `GCC`, includes compatible—though slower—subroutines.
2. General-purpose mathematical subroutines.

   The Developer's Kit from Cygnus Support includes an implementation of the standard C mathematical subroutine library. See section "Mathematical Functions" in *The Cygnus C Math Library*.

1.3.2.2 Predefined preprocessor macros

`GCC` defines the following preprocessor macros for the IDT/MIPS configurations:

Any MIPS *architecture:*
: `__mips__`

MIPS *architecture with big-endian numeric representation:*
: `__MIPSEB__`

MIPS *architecture with little-endian numeric representation:*
: `__MIPSEL__`

1.3.2.3 Assembler options

To use the GNU assembler, GAS, to assemble GCC output, configure GCC with the '`--with-gnu-as`' switch (as it is in Cygnus distributions) or with the `-mgas` option below.

GAS for MIPS architectures supports the MIPS R2000, R3000, and R4000 processors.

General GAS options

`-mgas` Compile using GAS to assemble GCC output.

`-Wa` If you invoke GAS through the GNU C compiler (version 2), you can use the '`-Wa`' option to pass arguments through to the assembler. One common use of this option is to exploit the assembler's listing features. Assembler arguments that you specify with `gcc -Wa` must be separated from each other (and the '`-Wa`') by commas.

`-L` The additional assembler option '`-L`' preserves local labels, which may make the listing output more intelligible to humans.

For example, in the following commandline:

```
$ mips-idt-ecoff-gcc -c -g -O -Wa,-alh,-L file.c
```

the assembler option `-ahl` requests a listing with high-level language and assembly language interspersed, `-L` preserves local labels, while the compiler debugging option `-g` gives the assembler the necessary debugging information.

GAS options for listing output

Use these options to enable *listing output* from the assembler (the letters after '`-a`' may be combined into one option, *e.g.*, '`-aln`'):

`-a` By itself, '`-a`' requests listings of high-level language source, assembly language, and symbols.

`-ah` Request a high-level language listing.

`-al` Request an output-program assembly listing.

`-as` Request a symbol table listing.

`-ad` *Omit* debugging directives from the listing.

High-level listings require that a compiler debugging option like '`-g`' be used, and that assembly listings ('`-al`') be requested also.

GAS listing-control directives

Use these *listing-control* assembler directives to control the appearance of the listing output (f you do not request listing output with one of the '`-a`' options, these listing-control directives have no effect):

`.list` Turn on listings from this point on.

`.nolist` Turn off listings from this point on.

`.psize` *`linecount`*`,` *`columnwidth`*
: Describe the page size for your output (the default is 60, 200). GAS generates form feeds after printing each group of *`linecount`* lines. To avoid these automatic form feeds, specify `0` as the *`linecount`*.

`.eject` Skip to a new page (issue a form feed).

`.title` Use *`heading`* as the title (this is the second line of the listing output, directly after the source file name and pagenumber) when generating assembly listings.

`.sbttl` Use *`subheading`* as the subtitle (this is the third line of the listing output, directly after the title line) when generating assembly listings.

`-an` Turn off all forms processing.

GAS options for MIPS

The MIPS configurations of GAS support three special options, and accept one other for command-line compatibility. See section "Command-Line Options" in *Using the GNU Assembler as*, for information on the command-line options available with all configurations of the GNU assembler.

`-G` *`num`* This option sets the largest size of an object that will be referenced implicitly with the gp register. It is only accepted for targets that use ECOFF format. The default value is 8.

`-EB`
`-EL` Any MIPS configuration of GAS can select big-endian or little-endian output at run time (unlike the other GNU development tools, which must be configured for one or the other). Use '`-EB`' to select big-endian output, and '`-EL`' for little-endian.

`-nocpp` This option is ignored. It is accepted for command-line compatibility with other assemblers, which use it to turn off C style preprocessing. With GAS, there is no need for '`-nocpp`', because the GNU assembler itself never runs the C preprocessor.

GAS directives for debugging information

MIPS ECOFF GAS supports several directives used for generating debugging information which are not supported by traditional MIPS assemblers:

```
.def            .endef          .dim
.file           .scl            .size
.tag            .type           .val
.stabd          .stabn          .stabs
```

The debugging information generated by the three `.stab` directives can only be read by GDB, not by traditional MIPS debuggers (this enhancement is required to fully support C++ debugging). These directives are primarily used by compilers, not assembly language programmers. See section "Assembler Directives" in *Using as*, for full information on all GAS directives.

MIPS ECOFF object code

The assembler supports some additional sections for a MIPS ECOFF target besides the usual `.text`, `.data` and `.bss`. The additional sections are:

`.rdata` For readonly data

`.sdata` For small data

`.sbss` For small common objects

When assembling for ECOFF, the assembler uses the `$gp` (`$28`) register to form the address of a *small object*. Any object in the `.sdata` or `.sbss` section is considered small in this sense.

Using small ECOFF objects requires linker support, and assumes that the `$gp` register has been correctly initialized (normally done automatically by the startup code).

Note: MIPS ECOFF assembly code must *not* modify the `$gp` register.

Options for MIPS ECOFF object code

`GCC -G` For external objects, or for objects in the `.bss` section, you can use the GCC '`-G`' option to control the size of objects addressed via `$gp`; the default value is `8`, meaning that a reference to any object eight bytes or smaller will use `$gp`.

`-G 0` Passing '`-G 0`' to GAS prevents GAS from using the `$gp` register on the basis of object size (the assembler uses `$gp` for objects in `.sdata` or `sbss` in any case).

Directives for MIPS ECOFF object code

`.comm`
`.lcomm` The size of an object in the `.bss` section is set by the `.comm` or `.lcomm` directive that defines it.

`.extern` The size of an external object may be set with the `.extern` directive. For example:

```
.extern sym,4
```

declares that the object at `sym` is 4 bytes in length, while leaving `sym` otherwise undefined.

1.3.2.4 Calling conventions

Arguments on MIPS architectures are not split, so that if a double word argument starts in 'R7', the entire word gets pushed onto the stack instead of being split between 'R7' and the stack.

The following calling convention for MIPS architectures depends on whether or not hardware floating-point is installed. Even if it is, the MIPS uses the registers for integer arguments *whenever* the first argument is an integer. The MIPS uses the registers for floating-point arguments only for floating-point arguments and only if the *first* argument is a floating point.

The following calling convention for the MIPS also depends on whether you're using standard 32-bit mode or Cygnus Support's 64-bit mode; 32-bit mode only allows the MIPS to use even numbered registers, while 64-bit mode allows the MIPS to use both odd and even numbered registers.

Note that functions compiled with different calling conventions cannot be run together without some care.

Registers used for integer arguments

If the first argument is an integer, the MIPS uses these registers for all arguments:

- The MIPS passes the first four words of arguments in registers 'R4' through 'R7', which are also called registers 'A0' through 'A3'.

If the function return values are integers, they're stored in 'R2' and 'R3'.

Registers used for floating-point arguments

If the first argument is a floating-point, the MIPS uses these registers for floating-point arguments:

- In 32-bit mode, the MIPS passes the first four words of arguments in registers 'F12' and 'F14'.
- In 64-bit mode, the MIPS passes the first four words of arguments in registers 'F12' through 'F15'.

If the function return value is a floating-point, it's stored in 'F0'.

Calling conventions used for integer arguments

These conventions apply to integer arguments:

'R0' is hardwired to the value 0. 'R1', which is also called 'AT', is reserved as the assembler's temporary register. 'R26' through 'R29' and 'R31' have reserved uses. Registers 'R2' through 'R15', 'R24', and 'R25' can be used for temporary values.

When a function is compiled with the default options, it must return with 'R16' through 'R23' and 'R30' unchanged.

Calling conventions used fo floating-point arguments

These conventions apply to floating-point arguments:

None of the registers has a reserved use.

- In 32-bit mode, 'F0' through 'F18' can be used for temporary values. When a function is compiled with the default options it must return with 'F20' through 'F30' unchanged.
- In 64-bit mode, 'F0' through 'F19' can be used for temporary values. When a function is compiled with the default options it must return with 'F20' through 'F31' unchanged.

1.3.3 Debugging on MIPS targets

GDB needs to know these things to talk to your MIPS:

1. what serial device connects your host to your MIPS board (the first serial device available on your host is the default).
2. what speed to use over the serial device.

`mips-idt-ecoff-gdb` uses the MIPS remote serial protocol to connect your development host machine to the target board. On the target board itself, the IDT program `IDT/sim` implements the same protocol. (`IDT/sim` runs automatically whenever the board is powered up.)

Use one of these GDB commands to specify the connection to your target board:

`target mips` *`port`*

To run a program on the board, start up GDB with the name of your program as the argument. To connect to the board, use the command '`target mips` *`port`*', where *`port`* is the name of the serial port connected to the board. If the program has not already been downloaded to the board, you may use the `load` command to download it. You can then use all the usual GDB commands.

For example, this sequence connects to the target board through a serial port, and loads and runs a program called *`prog`* through the debugger:

```
host$ mips-idt-ecoff-gdb prog
GDB is free software and ...
(gdb) target mips /dev/ttyb
...
(gdb) load
...
(gdb) run
```

`target mips` *`hostname`*`:`*`portnumber`*

You can specify a TCP/IP connection instead of a serial port, using the syntax *`hostname`*`:`*`portnumber`* (assuming your IDT board is connected so that this makes sense; for instance, to a serial line managed by a terminal concentrator).

GDB also supports these special commands for IDT/MIPS targets:

`set mipsfpu off`

If your target board does not support the MIPS floating point coprocessor, you should use the command '`set mipsfpu off`' (you may wish to put this in your '`.gdbinit`' file). This tells GDB how to find the return value of functions which return floating point values. It also allows GDB to avoid saving the floating point registers when calling functions on the board.

If you neglect to do this, calls into your program, such as '`print strlen("abc")`', will fail.

`set remotedebug` *n*

You can see some debugging information about communications with the board by setting the `remotedebug` variable. If you set it to 1 using '`set remotedebug 1`' every packet will be displayed. If you set it to 2 every character will be displayed. You can check the current value at any time with the command '`show remotedebug`'.

1.3.4 I/O for specific target architectures

Before you can use the Cygnus Developer's Kit to build your programs for IDT boards, you need a C library and C run-time initialization code. Unless you already have suitable libraries of your own, you must integrate the Cygnus C libraries with low-level code supplied by IDT. This low-level code initializes the C run-time environment, and describes the hardware interface to the Cygnus C libraries.

To begin with, make sure you have the following C and assembly source files from IDT:

C source files:

```
drv_8254.c          sys.c
idt_int_hand.c      syscalls.c
idtfpip.c           timer_int_hand.c
sbrk.c
```

C header files:

```
dpac.h              idtio.h
excepthdr.h         idtmon.h
fpip.h              iregdef.h
i8254.h             saunder.h
idt_entrypt.h       setjmp.h
idtcpu.h
```

Assembler files:

```
idt_csu.S           lnkexit.S
idt_except.S        lnkhelp.S
idtfpreg.S          lnkinstal.S
idtmem.S            lnkio.S
idttlb.S            lnkioctl.S
idtwbf.S            lnkjmp.S
lnkatb.S            lnkmem.S
lnkcach.S           lnknimp.S
lnkchar.S           lnkprint.S
lnkcio.S            lnksbrk.S
lnkcli.S            lnkstr.S
```

Note: For concreteness, these example commands assume the `mips` (big-endian) variant of the configuration; if you ordered tools configured

for little-endian object code, type '`mipsel`' wherever the examples show '`mips`'.

Follow these steps to integrate the low-level IDT code with your Cygnus Developer's Kit:

1. IDT supplies the C run-time initialization code in the file '`idt_csu.S`'. Since GNU CC expects to find the initialization module under the name `crt0.o`, rename the source file to match:

   ```
   $ mv idt_csu.S crt0.S
   ```

2. Edit the contents of '`crt0.S`'. A few more instructions are needed to ensure correct initialization, and to ensure that your programs exit cleanly. At the end of the file (after a comment including the text '`END I/O initialization`'), look for these lines:

   ```
            jal     main

   ENDFRAME(start)
   ```

   Insert '`move ra,zero`' before '`jal main`' to mark the top of the stack for the debugger, and add two lines after the call to `main` to call the `exit` routine (before the '`ENDFRAME(start)`'), so that the end of the file looks like this:

   ```
            move    ra,zero
            jal     main

            move    a0,v0
            jal     exit

   ENDFRAME(start)
   ```

3. Edit '`syscalls.c`', the interface to the low-level routines required by the C library, to remove the leading underbar from two identifiers:
   a. Rename `_kill` to `kill`;
   b. Rename `_getpid` to `getpid`.

4. Edit '`lnksbrk.S`' to remove the definition of `_init_sbrk`; this definition is not needed, since it is available in '`sbrk.c`'. Delete the lines marked with '`-`' at the left margin below:

   ```
            .text

   -FRAME(_init_sbrk,sp,0,ra)
   -        j       ra
   -ENDFRAME(_init_sbrk)
   -
   -
   -
    FRAME(_init_file,sp,0,ra)
            j       ra
    ENDFRAME(_init_file)
   ```

5. Use your Cygnus Developer's Kit to assemble the '`.S`' files, like this (use the compiler driver `gcc` to permit C preprocessing).

```
$ mips-idt-ecoff-gcc -g -c *.S
```

6. Compile the '`.c`' files.

   One particular C source file, '`drv_8254.c`' requires two special preprocessor symbol definitions: '`-DCLANGUAGE -DTADD=0xBF800000`'. *Be careful to type the constant value for* '`TADD`' *accurately;* the correct value is essential to allow the IDT board to communicate over its serial port.

   The two special preprocessor definitions make no difference to the other C source files, so you can compile them all with one call to the compiler, like this:

```
$ mips-idt-ecoff-gcc -g -O \
   -DCLANGUAGE -DTADD=0xBF800000 -c *.c
```

   (The example is split across two lines simply due to formatting constraints; you can type it on a single line instead of two lines linked by a '\', of course.)

7. Add the new object files to the C library archive, '`libc.a`', from your Cygnus Developer's Kit. Assuming you installed the Kit in '`/usr/cygnus/`' as we recommend:

```
$ mips-idt-ecoff-ar rvs /usr/cygnus/progressive-94q1/\
H-host/mips-idt-ecoff/lib/libc.a *.o
```

   As before, you can omit the '\' and type a single line. '`H-`*`host`*' stands for the string that identifies your host configuration; for example, on a SPARC computer running SunOS 4.1.3, you'd actually type '`H-sparc-sun-sunos4.1.3`'.

Linking with the GOFAST library

The GOFAST library is available with two interfaces; `GCC` '`-msoft-float`' output places all arguments in registers, which (for subroutines using `double` arguments) is compatible with the interface identified as "Interface 1: all arguments in registers" in the GOFAST documentation. For full compatibility with all GOFAST subroutines, you need to make a slight modification to some of the subroutines in the GOFAST library.

If you purchase and install the GOFAST library, you can link your code to that library in a number of different ways, depending on where and how you install the library.

To focus on the issue of linking, the following examples assume you've already built object modules with appropriate options (including '`-msoft-float`').

This is the simplest case; it assumes that you've installed the GOFAST library as the file '`fp.a`' in the same directory where you do development, as shown in the GOFAST documentation:

```
$ mips-idt-ecoff-gcc -o prog prog.o ... -lc fp.a
```

In a shared development environment, this example may be more realistic; it assumes you've installed the GOFAST library as '*ussdir*`/libgofast.a`', where *ussdir* is any convenient directory on your development system:

```
$ mips-idt-ecoff-gcc -o program program.o ... \
  -lc -Lussdir -lgofast
```

Finally, you can eliminate the need for a '`-L`' option with a little more setup, using an environment variable like this (the example assumes you use a command shell compatible with the Bourne shell):

```
$ LIBRARY_PATH=ussdir; export LIBRARY_PATH
$ mips-idt-ecoff-gcc -o program program.o ... -lc -lgofast
```

As for the previous example, the GOFAST library here is installed in '*ussdir*`/libgofast.a`'. The environment variable `LIBRARY_PATH` instructs `GCC` to look for the library in *ussdir*. (The syntax shown here for setting the environment variable is the Unix Bourne Shell, '`/bin/sh`', syntax; adjust as needed for your system.)

Notice that all the variations on linking with the GOFAST library explicitly include '`-lc`' *before* the GOFAST library. '`-lc`' is the standard C subroutine library; normally, you don't have to specify this, since linking with that library is automatic.

When you link with an alternate software floating-point library, however, the order of linking is important. In this situation, specify '`-lc`' *to the left* of the GOFAST library, to ensure that standard library subroutines also use the GOFAST floating-point code.

Full compatibility with the GOFAST library

The `GCC` calling convention for functions whose first and second arguments have type `float` is not completely compatible with the definitions of those functions in the GOFAST library, as shipped.

These functions are affected:

```
fpcmp   fpadd   fpsub   fpmul   fpdiv   fpfmod
fpacos  fpasin  fpatan  fpatan2 fppow
```

Since the GOFAST library is normally shipped with source, you can make these functions compatible with the `GCC` convention by adding this instruction to the beginning of each affected function, then rebuilding the library:

```
        move    $5,$6
```

1.3.5 Further documentation

For information about the MIPS instruction set, see *MIPS RISC Architecture*, by Kane and Heindrich (Prentice-Hall).

For information about IDT's `IDT/sim` board monitor program, see *IDT/sim 4.0 Reference Manual* (IDT#703-00146-001/A).

For information about US Software's floating point library, see *U S Software GOFAST R3000 Floating Point Library* (United States Software Corporation).

1.4 Motorola m68k targets

1.4.1 What to call the tools

Cross-development tools in the Cygnus Developer's Kit are normally installed with names that reflect the target machine, so that you can install more than one set of tools in the same binary directory.

The target name, constructed with the `--target` option to `configure`, is used as a prefix to the program name. For example, the compiler for the Motorola `m68k` (called simply `gcc` in native configurations) is called by one of these names, depending on which configuration you have installed:

```
m68k-coff-gcc
m68k-aout-gcc
```

Likewise, the `m68k`-configured GDB is called by one of these names:

```
m68k-coff-gdb
m68k-aout-gdb
```

For DOS-hosted toolchains, the tools are simply called by their standard names, e.g., `gcc`, `gdb`, etc.

1.4.2 Compiling on Motorola m68k targets

The Motorola `m68k` target family toolchain controls variances in code generation directly from the command line.

When you run `GCC`, you can use command-line options to choose whether to take advantage of the extra Motorola `m68k` machine instructions, and whether to generate code for hardware or software floating point.

1.4.2.1 Compiler options

When you run `GCC`, you can use command-line options to choose machine-specific details. For information on all the `GCC` command-line options, see section "GNU CC Command Options" in *Using GNU CC*.

GCC options for architecture and code generation

`-g` The compiler debugging option '`-g`' is essential to see interspersed high-level source statements, since without debugging information the assembler cannot tie most of the generated code to lines of the original source file.

`-m68000` Generate code for the Motorola `m68000`.

`-m68020` Generate code for the Motorola `m68020`.

`-m68030` Generate code for the Motorola `m68030`.

`-m68040` Generate code for the Motorola `m68040`. Also enables code generation for the `68881` FPU by default.

`-m68332` Generate code for the Motorola `cpu32` family, of which the Motorola `m68332` is a member.

GCC options for floating point

`-msoft-float`
Generate output containing library calls for floating point. The Motorola configurations of '`libgcc`' include a collection of subroutines to implement these library calls.

`-m68881` Generate code for the Motorola `m68881` FPU. See compiler option '`-m68040`' above.

Floating point subroutines

Two kinds of floating point subroutines are useful with `GCC`:

1. Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.
2. General-purpose mathematical subroutines.

   The Developer's Kit from Cygnus Support includes an implementation of the standard C mathematical subroutine library. See section "Mathematical Functions" in *The Cygnus C Math Library*.

1.4.2.2 Predefined preprocessor macros

GCC defines the following preprocessor macros for the Motorola m68k configurations:

Any Motorola m68k *architecture:*
__mc68000__

Any Motorola m68010 *architecture:*
__mc68010__

Any Motorola m68020 *architecture:*
__mc68020__

Any Motorola m68030 *architecture:*
__mc68030__

Any Motorola m68040 *architecture:*
__mc68040__

Any Motorola m68332 *architecture:*
__mc68332__

Any Motorola m68881 *architecture:*
__HAVE_68881__

1.4.2.3 Assembler options

To use the GNU assembler, GAS, to assemble GCC output, configure GCC with the '--with-gnu-as' switch (as it is in Cygnus distributions) or with the -mgas option below.

General GAS options

-mgas Compile using GAS to assemble GCC output.

-Wa If you invoke GAS through the GNU C compiler (version 2), you can use the '-Wa' option to pass arguments through to the assembler. One common use of this option is to exploit the assembler's listing features. Assembler arguments that you specify with gcc -Wa must be separated from each other (and the '-Wa') by commas.

-L The additional assembler option '-L' preserves local labels, which may make the listing output more intelligible to humans.

For example, in the following commandline:

```
$ m68k-coff-gcc -c -g -O -Wa,-alh,-L file.c
```

the assembler option `-ahl` requests a listing with high-level language and assembly language interspersed, `-L` preserves local labels, while the compiler debugging option `-g` gives the assembler the necessary debugging information.

GAS options for listing output

Use these options to enable *listing output* from the assembler (the letters after '`-a`' may be combined into one option, *e.g.*, '`-aln`'):

`-a` By itself, '`-a`' requests listings of high-level language source, assembly language, and symbols.

`-ah` Request a high-level language listing.

`-al` Request an output-program assembly listing.

`-as` Request a symbol table listing.

`-ad` *Omit* debugging directives from the listing.

High-level listings require that a compiler debugging option like '`-g`' be used, and that assembly listings ('`-al`') be requested also.

GAS listing-control directives

Use these *listing-control* assembler directives to control the appearance of the listing output (f you do not request listing output with one of the '`-a`' options, these listing-control directives have no effect):

`.list` Turn on listings from this point on.

`.nolist` Turn off listings from this point on.

`.psize` *`linecount`*`,` *`columnwidth`*
Describe the page size for your output (the default is 60, 200). GAS generates form feeds after printing each group of *`linecount`* lines. To avoid these automatic form feeds, specify `0` as the *`linecount`*.

`.eject` Skip to a new page (issue a form feed).

`.title` Use *`heading`* as the title (this is the second line of the listing output, directly after the source file name and pagenumber) when generating assembly listings.

`.sbttl` Use *`subheading`* as the subtitle (this is the third line of the listing output, directly after the title line) when generating assembly listings.

`-an` Turn off all forms processing.

1.4.2.4 Calling conventions

The Motorola m68k pushes all arguments onto the stack, last to first, so that the lowest numbered argument not passed in a register is at the lowest address in the stack.

Function return values for integers are stored in 'D0' and 'D1'. 'A7' has a reserved use. Registers 'A0', 'A1', 'D0', 'D1' 'F0', and 'F1' can be used for temporary values.

When a function is compiled with the default options, it must return with registers 'D2' through 'D7' and registers 'A2' through 'A6' unchanged. If you have floating-point registers, then registers 'F2' through 'F7' must also be unchanged.

Note that functions compiled with different calling conventions cannot be run together without some care.

1.4.3 Debugging on Motorola m68k targets

GDB needs to know these things to talk to your Motorola m68k:

1. that you want to use one of the following interfaces:

   'target rom68k', the rom monitor for the IDP board.

   'target cpu32bug', the rom monitor for other Motorola boards, such as the Motorola Business Card Computer, BCC.

   'target est', the EST Net/300 emulator.

   'target remote', GDB's generic debugging protocol.
2. what serial device connects your host to your m68k board (the first serial device available on your host is the default).
3. what speed to use over the serial device.

Use these GDB commands to specify the connection to your target board:

target *interface serial-device*

To run a program on the board, start up GDB with the name of your program as the argument. To connect to the board, use the command 'target *interface serial-device*', where *interface* is an interface from the list above and *serial-device* is the name of the serial port connected to the board. If the program has not already been downloaded to the board, you may use the load command to download it. You can then use all the usual GDB commands.

For example, this sequence connects to the target board through a serial port, and loads and runs a program called *prog* through the debugger:

```
host$ m68k-coff-gdb prog
GDB is free software and ...
(gdb) target cpu32bug /dev/ttyb
...
(gdb) load
...
(gdb) run
```

`target m68k` *`hostname`*`:`*`portnumber`*

You can specify a TCP/IP connection instead of a serial port, using the syntax *`hostname`*`:`*`portnumber`* (assuming your board is connected so that this makes sense; for instance, to a serial line managed by a terminal concentrator).

GDB also supports:

`set remotedebug` *`n`*

You can see some debugging information about communications with the board by setting the `remotedebug` variable.

1.5 Sparc targets

The Cygnus Developer's Kit supports both the SPARC and the Fujitsu SPARClite families. For the compiler in particular, special configuration options allow you to use special software floating-point code for the SPARC MB86930 chip, as well as defaulting command-line options to use special Fujitsu SPARClite features.

For the Fujitsu SPARClite, the CDK currently supports boards ex930, ex931, ex932, ex933, ex934, and ex936.

1.5.1 What to call the tools

Cross-development tools in the Cygnus Developer's Kit are normally installed with names that reflect the target machine, so that you can install more than one set of tools in the same binary directory.

The target name, constructed with the `--target` option to `configure`, is used as a prefix to the program name. For example, the compiler for the SPARC (called simply `gcc` in native configurations) is called by one of these names, depending on which configuration you have installed:

```
sparc-aout-gcc
sparc-coff-gcc
```

Likewise, the SPARC-configured GDB is called:

```
sparc-aout-gdb
```

The compiler for the Fujitsu SPARClite is called by one of these names, depending on which configuration you have installed:

```
sparclite-aout-gcc
sparclite-coff-gcc
```

Likewise, the SPARClite-configured GDB is called by one of these names:

```
sparclite-aout-gdb
sparclite-coff-gdb
```

For DOS-hosted toolchains, the tools are simply called by their standard names, e.g., `gcc`, `gdb`, etc.

1.5.2 Compiling on Sparc targets

The SPARC target family toolchain controls variances in code generation directly from the command line.

When you run GCC, you can use command-line options to choose whether to take advantage of the extra SPARC machine instructions, and whether to generate code for hardware or software floating point.

1.5.2.1 Compiler options

When you run GCC, you can use command-line options to choose machine-specific details. For information on all the GCC command-line options, see section "GNU CC Command Options" in *Using GNU CC*.

GCC options for architecture and code generation

The CDK supports the machine-dependent options for SPARC in addition to special compiler command-line options available for Fujitsu SPARClite. Both kinds of options are described in section "SPARC Options" in *Using GNU CC*.

`-g` The compiler debugging option '`-g`' is essential to see interspersed high-level source statements, since without debugging information the assembler cannot tie most of the generated code to lines of the original source file.

`-mv8` '`-mv8`' gives you SPARC v8 code. The only difference from v7 code is that the compiler emits the integer multiply (`smul` and `umul`) and integer divide (`sdiv` and `udiv`) instructions which exist in SPARC v8 but not in SPARC v7.

`-mf930` Generate code specifically intended for the SPARC MB86930, a Fujitsu SPARClite chip without an FPU. This option is equivalent to the combination '`-msparclite -mno-fpu`'.

'`-mf930`' is the default when the compiler is configured specifically for Fujitsu SPARClite.

`-mf934` Generate code specifically for the SPARC MB86934, a Fujitsu SPARClite chip *with* an FPU. This option is equivalent to '`-msparclite`'.

`-mflat` Does not use register windows in function calls.

`-msparclite`
The SPARC configurations of GCC generate code for the common subset of the instruction set: the v7 variant of the SPARC architecture.

'`-msparclite`' (which is on automatically for any of the Fujitsu SPARClite configurations) gives you SPARClite code. This adds the integer multiply (`smul` and `umul`, just as in SPARC v8), integer divide-step (`divscc`), and scan (`scan`) instructions which exist in SPARClite but not in SPARC v7.

Using '`-msparclite`' when you run the compiler does *not*, however, give you floating point code that uses the entry points for US Software's GOFAST library.

The following command line options are available for both the SPARC and the Fujitsu SPARClite configurations of the compiler. See section "SPARC Options" in *Using GNU CC*.

GCC options for floating point

When you run the compiler, you can specify whether to compile for hardware or software floating point configurations with these GCC command-line options:

`-mfpu`
`-mhard-float`
: Generate output containing floating point instructions. This is the default.

`-msoft-float`
`-mno-fpu`
: Generate output containing library calls for floating point. The SPARC configurations of 'libgcc' include a collection of subroutines to implement these library calls.

  In particular, the Fujitsu SPARClite GCC configurations generate subroutine calls compatible with the US Software 'goFast.a' floating point library, giving you the opportunity to use either the 'libgcc' implementation or the US Software version.

  To use the US Software library, simply include '-lgoFast' on the GCC command line.

  To use the 'libgcc' version, you need nothing special; GCC links with 'libgcc' automatically after all other object files and libraries.

Floating point subroutines

Two kinds of floating point subroutines are useful with GCC:

1. Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.

   When you indicate that no hardware floating point is available (with either of the GCC options '-msoft-float' or '-mno-fpu'), the Fujitsu SPARClite configurations of GCC generate calls compatible with the U.S. Software GOFAST library. If you do not have this library, you can still use software floating point; 'libgcc', the auxiliary library distributed with GCC, includes compatible—though slower—subroutines.

2. General-purpose mathematical subroutines.

   The Developer's Kit from Cygnus Support includes an implementation of the standard C mathematical subroutine library. See section "Mathematical Functions" in *The Cygnus C Math Library*.

1.5.2.2 Predefined preprocessor macros

GCC defines the following preprocessor macros for the SPARC configurations:

Any SPARC *architecture:*

```
__sparc__
```

Any Fujitsu SPARClite *architecture:*

```
__sparclite__
```

1.5.2.3 Assembler options

To use the GNU assembler, GAS, to assemble GCC output, configure GCC with the '`--with-gnu-as`' switch (as it is in Cygnus distributions) or with the `-mgas` option below.

General GAS options

`-mgas` Compile using GAS to assemble GCC output.

`-Wa` If you invoke GAS through the GNU C compiler (version 2), you can use the '`-Wa`' option to pass arguments through to the assembler. One common use of this option is to exploit the assembler's listing features. Assembler arguments that you specify with `gcc -Wa` must be separated from each other (and the '`-Wa`') by commas.

`-L` The additional assembler option '`-L`' preserves local labels, which may make the listing output more intelligible to humans.

For example, in the following commandline:

```
$ sparc-coff-gcc -c -g -O -Wa,-alh,-L file.c
```

the assembler option `-ahl` requests a listing with high-level language and assembly language interspersed, `-L` preserves local labels, while the compiler debugging option `-g` gives the assembler the necessary debugging information.

GAS options for listing output

Use these options to enable *listing output* from the assembler (the letters after '`-a`' may be combined into one option, *e.g.*, '`-aln`'):

`-a` By itself, '`-a`' requests listings of high-level language source, assembly language, and symbols.

`-ah` Request a high-level language listing.

`-al` Request an output-program assembly listing.

`-as` Request a symbol table listing.

`-ad` *Omit* debugging directives from the listing.

High-level listings require that a compiler debugging option like '`-g`' be used, and that assembly listings ('`-al`') be requested also.

GAS listing-control directives

Use these *listing-control* assembler directives to control the appearance of the listing output (f you do not request listing output with one of the '`-a`' options, these listing-control directives have no effect):

`.list` Turn on listings from this point on.

`.nolist` Turn off listings from this point on.

`.psize` *`linecount`*`,` *`columnwidth`*
Describe the page size for your output (the default is 60, 200). GAS generates form feeds after printing each group of *`linecount`* lines. To avoid these automatic form feeds, specify `0` as the *`linecount`*.

`.eject` Skip to a new page (issue a form feed).

`.title` Use *`heading`* as the title (this is the second line of the listing output, directly after the source file name and pagenumber) when generating assembly listings.

`.sbttl` Use *`subheading`* as the subtitle (this is the third line of the listing output, directly after the title line) when generating assembly listings.

`-an` Turn off all forms processing.

GAS options for the Fujitsu SPARClite

When configured for SPARC, GAS recognizes the additional Fujitsu SPARClite machine instructions that GCC can generate:

`-Asparclite`
: A flag to the GNU assembler (configured for SPARC) explicitly selects this particular SPARC architecture. The SPARC assembler automatically selects the Fujitsu SPARClite architecture whenever it encounters one of the SPARClite-only instructions (`divscc` or `scan`).

1.5.2.4 Calling conventions

The SPARC passes the first six words of arguments in registers 'R8' through 'R13'. All remaining arguments are stored in a reserved block on the stack, last to first, so that the lowest numbered argument not passed in a register is at the lowest address in the stack. The registers are always filled, so a double word argument starting in 'R13' would have the most significant word in 'R13' and the least significant word on the stack.

Function return values are stored in 'R8'. 'R0' is hardwired so that it always has the value 0. 'R14' and 'R15' have reserved uses. Registers 'R1' through 'R7' can be used for temporary values.

When a function is compiled with the default options, it must return with registers 'R16' through 'R29' unchanged.

Note that functions compiled with different calling conventions cannot be run together without some care.

1.5.3 Debugging on Sparc targets

GDB needs to know these things to talk to your SPARC or Fujitsu SPARClite:

1. that you want to use:

   'target remote', GDB's generic debugging protocol.
2. what serial device connects your host to your SPARC board (the first serial device available on your host is the default).
3. what speed to use over the serial device.

Use one of these GDB commands to specify the connection to your sparc target board:

`target sparclite` *`serial-device`*

To run a program on the board, start up GDB with the name of your program as the argument. To connect to the board, use the command '`target sparclite` *`serial-device`*', where *`serial-device`* is the name of the serial port connected to the board. If the program has not already been downloaded to the board, use the `load` command to download it.

For example, this sequence connects to the target board through a serial port, and loads and runs a program called *`prog`* through the debugger:

```
(gdb) target sparclite com1
[SPARClite appears to be alive]
(gdb) load
[Loading section .text at 0x40000000 (9160 bytes)]
[Loading section .data at 0x400023c8 (96 bytes)]
[Starting hello at 0x40000020]
```

`target sparclite` allows loading, but no other operations. This sequence uses `target remote` to debug:

```
(gdb) target remote com1
Remote debugging using com1
breakinst () ../sparcl-stub.c:975
975     }
(gdb) s
main () hello.c:50
50        writez(1, "Got to here\n");
(gdb)
```

`target sparclite` *`hostname`*`:`*`portnumber`*

You can specify a TCP/IP connection instead of a serial port, using the syntax *`hostname`*`:`*`portnumber`* (assuming your SPARClite board is connected so that this makes sense; for instance, to a serial line managed by a terminal concentrator).

GDB also supports:

`set remotedebug` *`n`*

You can see some debugging information about communications with the board by setting the `remotedebug` variable.

1.5.4 Loading on specific target architectures

The SPARC eval boards use a host-based terminal program to load and execute programs on the target. This program, `pciuh`, is relatively new and it replaces the previous `ROM` monitor, which had the shell in the `ROM`.

To use the GDB remote serial protocol to communicate with a Fujitsu `SPARClite` board, link your programs with the "stub" module '`sparc-stub.c`'; this module manages the communication with GDB. See section "The GDB remote serial protocol" in *Debugging with GDB*, for more details.

1.5.5 Further documentation

See *SPARClite User's Manual* (Fujitsu Microelectronics, Inc. Semiconductor Division, 1993) for full documentation of the `SPARClite` family, architecture, and instruction set.

Index

L

M

N

O

P

R

S

T