GNUPRO™ TOOLKIT

GNUPro Tools for Embedded Systems

July, 1998
98r1

CYGNUS

This documentation has been prepared by Cygnus Technical Publications; contact the Cygnus Technical Publications staff: **doc@cygnus.com**.

Part #: 300-400-101000047

GNUPro Warranty

The GNUPro Toolkit is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. This version of GNUPro Toolkit is supported for customers of Cygnus.

For non-customers, GNUPro Toolkit software has NO WARRANTY.

Because this software is licensed free of charge, there are no warranties for it, to the extent permitted by applicable law. Except when otherwise stated in writing, the copyright holders and/or other parties provide the software “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event, unless required by applicable law or agreed to in writing, will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

How to Contact Cygnus

Use the following means to contact Cygnus.

Cygnus Headquarters
1325 Chesapeake Terrace
Sunnyvale, CA 94089 USA
Telephone (toll free): +1 800 CYGNUS-1
Telephone (main line): +1 408 542 9600
Telephone (hotline): +1 408 542 9601
FAX: +1-408 542 9699
(Faxes are answered 8 a.m.–5 p.m., Monday through Friday.)
email: `info@cygnus.com`
Website: `www.cygnus.com`.

Cygnus United Kingdom
36 Cambridge Place
Cambridge CB2 1NS
United Kingdom
Telephone: +44 1223 728728
FAX: +44 1223 728728
email: `info@cygnus.co.uk/`

Cygnus Japan
Nihon Cygnus Solutions
Madre Matsuda Building
4-13 Kioi-cho Chiyoda-ku
Tokyo 102-0094
Telephone: +81 3 3234 3896
FAX: +81 3 3239 3300
email: `info@cygnus.co.jp`
Website: `http://www.cygnus.co.jp/`

Use the hotline (+1 408 542 9601) to get help, although the most reliable way to resolve problems with GNUPro Toolkit is by using email:
`bugs@cygnus.com`.

Contents

1 Using GNU tools on embedded systems

The following GNUPro tools can be run on embedded targets.

- `gcc`, the GNUPro Toolkit compiler (see “gcc, the GNU compiler” on page 2)
- `ccp`, the GNU C preprocessor (see “cpp, the GNU preprocessor” on page 2)
- `gas`, the GNUPro Toolkit assembler (see “gas, the GNU assembler” on page 3)
- `ld`, the GNUPro Toolkit linker (see “ld, the GNU linker” on page 3)
- `binutils`, the GNUPro Toolkit directory of utilities (see “binutils, the GNU binary utilities” on page 3)
- `gdb`, the GNUPro Toolkit debugger (see “gdb, the debugging tool” on page 5)
- `libgloss`, the support library for embedded targets and `newlib`, the C library developed by Cygnus (see “libgloss, newlib and libstd++, the GNU libraries” on page 5)

See the following documentation for more discussion on using the GNU tools.

- “Invoking the GNU tools” on page 2
- “crt0, the main startup file” on page 7
- “The linker script” on page 11
- “I/O support code” on page 14
- “Memory support” on page 15
- “Miscellaneous support routines” on page 16

Invoking the GNU tools

`gcc` invokes all the required GNU passes for you with the following utilities.

- `cpp`
  The preprocessor which processes all the header files and macros that your target requires.
- `gcc`
  The compiler which produces assembly language code from the processed C files. For more information, see *Using GNU CC* in ***GNUPro Compiler Tools***.
- `gas`
  The assembler which produces binary code from the assembly language code and puts it in an object file.
- `ld`
  The linker which binds the code to addresses, links the startup file and libraries to the object file, and produces the executable binary image.

There are several machine-independent compiler switches, among which are, notably, `-fno-exceptions` (for C++), `-fritti` (for C++) and `-T` (for linking).

You have four implicit file extensions: `.c`, `.C`, `.s`, and `.S`. For more information, see *Using GNU CC* in ***GNUPro Compiler Tools***.

gcc, the GNU compiler

When you compile C or C++ programs with gnu C, the compiler quietly inserts a call at the beginning of `main` to a `gcc` support subroutine called `__main`. Normally this is invisible—you may run into it if you want to avoid linking to the standard libraries, by specifying the compiler option, `-nostdlib`. Include `-lgcc` at the end of your compiler command line to resolve this reference. This links with the compiler support library `libgcc.a`. Putting it at the end of your command line ensures that you have a chance to link first with any of your own special libraries.

`__main` is the initialization routine for C++ constructors. Because GNU C is designed to interoperate with GNU C++, even C programs must have this call: otherwise C++ object files linked with a C main might fail. For more information on `gcc`, see *Using GNU CC* in ***GNUPro Compiler Tools***.

cpp, the GNU preprocessor

`cpp` merges in the `#include` files, expands all macros definitions, and processes the `#ifdef` sections. To see the output of `cpp`, invoke `gcc` with the `-E` option, and the preprocessed file will be printed on `stdout`.

There are two convenient options to assemble handwritten files that require C-style preprocessing. Both options depend on using the compiler driver program, `gcc`, instead of calling the assembler directly.

- Name the source file using the extension `.S` (capitalized) rather than `.s`. `gcc` recognizes files with this extension as assembly language requiring C-style preprocessing.
- Specify the "source language" explicitly for this situation, using the `gcc` option, `-xassembler-with-cpp`.

For more information on `cpp`, see *The C Preprocessor* in ***GNUPro Compiler Tools***.

`gas`, the GNU assembler

`gas` can be used as either a compiler pass or a source-level assembler.

When used as a source-level assembler, it has a companion assembly language preprocessor called `gasp`. `gasp` has a syntax similar to most other assembly language macro packages.

`gas` emits a relocatable object file from the assembly language source code. The object file contains the binary code and the debug symbols.

For more information on `gas`, see *Using AS* in ***GNUPro Utilities***.

`ld`, the GNU linker

`ld` resolves the code addresses and debug symbols, links the startup code and additional libraries to the binary code, and produces an executable binary image.

For more information on `ld`, see *Using LD* in ***GNUPro Utilities***.

`.coff` for object file formats

`.coff` is the main object file format when using the tools on embedded target systems. For more information on object files and object file formats, see *The GNU Binary Utilities* in ***GNUPro Utilities***.

`binutils`, the GNU binary utilities

The following are the binary utilities, although they are not included on all hosts: `ar`, `nm`, `objcopy`, `objdump`, `ranlib`, `size`, `strings`, and `strip`.

For more information on `binutils`, see *The GNU Binary Utilities* in ***GNUPro Utilities***.

The most important of these utilities are `objcopy` and `objdump`.

`objcopy`

A few ROM monitors, such as `a.out`, load executable binary images, and, consequently, most load an S-record. An S-record is a printable ASCII representation of an executable binary image.

S-records are suitable both for building ROM images for standalone boards and for downloading images to embedded systems. Use the following example's input for this process.

```
objcopy -O srec infile outfile
```

`infile` in the previous example's input is the executable binary filename, and *`outfile`* is the filename for the S-record.

Most PROM burners also read S-records or some similar format. Use the following example's input to get a list of supported object file types for your architecture.

```
objdump -i
```

For more information on S-records, see the discussions for `FORMAT` *`output-format`* in the documentation for "MRI comptible files" and the discussion for "BFD" in *Using LD* in ***GNUPro Utilities***. For more discussion of making an executable binary image, see "`objcopy`" in *The GNU Binary Utilities* in ***GNUPro Utilities***.

`objdump`

`objdump` displays information about one or more object files. The options control what particular information to display. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work.

When specifying archives, `objdump` shows information on each of the member object files. `objfile...` designates the object files to be examined.

A few of the more useful options for commands are: `-d`, `-disassemble` and `--prefix-addresses`.

`-d`
`--disassemble`

Displays the assembler mnemonics for the machine instructions from *`objfile`*. This option only disassembles those sections that are expected to contain instructions.

`--prefix-addresses`

For disassembling, prints the complete address on each line, starting each output line with the address it's disassembling. This is the older disassembly format. Otherwise, you only get raw opcodes.

gdb, the debugging tool

To run gdb on an embedded execution target, use a gdb backend with the gdb standard remote protocol or a similar protocol. The most common are the following two types of gdb backend.

- A gdb stub
  This is an exception handler for breakpoints, and it must be linked to your application. gdb stubs use the gdb standard remote protocol.
- An existing ROM monitor used as a gdb backend
  The most common approach means using the following processes.
  - With a similar protocol to the gdb standard remote protocol.
  - With an interface that uses the ROM monitor directly. With such an interface, gdb only formats and parses commands.

For more information on debugging tools, see *Debugging with GDB* in ***GNUPro Debugging Tools***.

Useful debugging routines

The following routines are always useful for debugging a project in progress.

- `print()`
  Runs standalone in `libgloss` with no `newlib` support. Many times `print()` works when there are problems that make `printf()` cause an exception.
- `outbyte()`
  Used for low-level debugging.
- `putnum()`
  Prints out values in hex so they are easier to read.

libgloss, newlib and libstd++, the GNU libraries

GNUPro Toolkit has three libraries: `libgloss`, `newlib` and `libstd++`.

`libgloss`

`libgloss`, the library for GNU Low-level OS Support, contains the startup code, the I/O support for `gcc` and `newlib` (the C library), and the target board support packages that you need to port the GNU tools to an embedded execution target.

The C library used throughout this manual is `newlib`, however `libgloss` could easily be made to support other C libraries. Because `libgloss` resides in its own tree, it's able to run standalone, allowing it to support GDB's remote debugging and to be included in other GNU tools.

Several functions that are essential to gcc reside in libgloss. These include the following functions.

- crt0, the main startup script (see “crt0, the main startup file” on page 7)
- ld, the linker script (see “The linker script” on page 11)
- I/O support code (see “I/O support code” on page 14)

newlib
: The Cygnus libraries, including the C library, libc, and the C math library, libm.

libstd++
: The C++ library in development by Cygnus.

crt0, the main startup file

The crt0 (C RunTime 0) file contains the initial startup code.

Cygnus provides a crt0 file, although you may want to write your own crt0 file for each target. The crt0 file is usually written in assembler as 'crt0.S', and its object gets linked in first and bootstraps the rest of your application. The crt0 file defines a special symbol like _start, which is both the default base address for the application and the first symbol in the executable binary image.

If you plan to use any routines from the standard C library, you'll also need to implement the functions on which libgloss depends. The crt0 file accomplishes the following results. See "I/O support code" on page 14.

- **crt0** ***initializes everything in your program that needs it.***
  This initialization section varies. If you are developing an application that gets downloaded to a ROM monitor, there is usually no need for special initialization because the ROM monitor handles it for you. If you plan to burn your code in a ROM, the crt0 file typically does all of the hardware initialization required to run an application. This can include things like initializing serial ports and running a memory check; however, results vary depending on your hardware.

  The following is a typical basic initialization of crt0.S.

  1. Set up concatenation macros.

     ```
     #define CONCAT1(a, b) CONCAT2(a, b)
     #define CONCAT2(a, b) a ## b
     ```

     Later, you'll use these macros.

  2. Set up label macros, using the following example's input.

     ```
     #ifndef __USER_LABEL_PREFIX__
     #define __USER_LABEL_PREFIX__ _
     #endif
     #define SYM(x) CONCAT1 (__USER_LABEL_PREFIX__, x)
     ```

     These macros make the code portable between coff and a.out. coff always has an __ (underline) prepended to the front of its global symbol names. a.out has none.

  3. Set up register names (with the right prefix), using the following example's input.

     ```
     #ifndef __REGISTER_PREFIX__
     #define __REGISTER_PREFIX__
     #endif
     /* Use the right prefix for registers. */
     #define REG(x) CONCAT1 (__REGISTER_PREFIX__, x)
     ```

```
#define d0 REG (d0)
#define d1 REG (d1)
#define d2 REG (d2)
#define d3 REG (d3)
#define d4 REG (d4)
#define d5 REG (d5)
#define d6 REG (d6)
#define d7 REG (d7)
#define a0 REG (a0)
#define a1 REG (a1)
#define a2 REG (a2)
#define a3 REG (a3)
#define a4 REG (a4)
#define a5 REG (a5)
#define a6 REG (a6)
#define fp REG (fp)
#define sp REG (sp)
```

Register names are for portability between assemblers. Some register names have a `%` or `$` prepended to them.

4. Set up space for the stack and grab a chunk of memory.

```
.set stack_size, 0x2000 .
comm SYM (stack), stack_size
```

This can also be done in the linker script, although it typically gets done at this point.

5. Define an empty space for the environment, using the following example's input.

```
      .data
      .align 2
SYM (environ):
      .long 0
```

This is bogus on almost any ROM monitor, although it's best generally set up as a valid address, then passing the address to `main()`. This way, if an application checks for an empty environment, it finds one.

6. Set up a few global symbols that get used elsewhere.

```
.align 2
.text
.global SYM (stack)
.global SYM (main)
.global SYM (exit)
.global __bss_start
```

This really should be `__bss_start`, not `SYM (__bss_start`.

`__bss_start` needs to be declared this way because its value is set in the linker script.

7. Set up the global symbol, `start`, for the linker to use as the default address for the `.text` section. This helps your program run.

```
SYM (start):
link a6, #-8
moveal #SYM (stack) + stack_size, sp
```

- **`crt0` *zeroes the* `.bss` *section***

  Make sure the `.bss` section is cleared for uninitialized data, using the following example's input. All of the addresses in the `.bss` section need to be initialized to zero so programs that forget to check new variables' default values will get predictable results.

```
moveal #__bss_start, a0
moveal #SYM (end), a1
1:
movel #0, (a0)
leal 4(a0), a0
cmpal a0, a1
bne 1b
```

  Applications can get wild side effects from the `.bss` section being left uncleared, and it can cause particular problems with some implementations of `malloc()`.

- **`crt0` *calls* `main()`**

  If your ROM monitor supports it, set up `argc` and `argv` for command line arguments and an environment pointer before the call to `main()`, using the following example's input.

  For `g++`, the code generator inserts a branch to `__main` at the top of your `main()` routine. `g++` uses `__main` to initialize its internal tables and then returns control to your `main()` routine.

  For `crt0` to call your `main()` routine, use the following example's input. First, set up the environment pointer and jump to `main()`. Call the main routine from the application to get it going, using the following example's input with `main (argc, argv, environ)`, using `argv` as a pointer to NULL.

```
            pea 0
            pea SYM (environ)
            pea sp@(4)
            pea 0
jsr SYM (main)
movel d0, sp@-4
```

- **`crt0` *calls* `(exit)`**

  After `main()` has run, the `crt0` file cleans things up and returns control of the hardware from the application. On some hardware there is nothing to return to—especially if your program is in ROM— and if that's the case, you need to do a hardware reset or branch back to the original start address.

If you're using a ROM monitor, you can usually call a user trap to make the ROM take over. Pick a safe vector with no sides effects. Some ROM's have a built-in trap handler just for this case.

Implementing `(exit)` here is easy.. First, with `_exit`, exit from the application. Normally, this causes a user trap to return to the ROM monitor for another run. Then, using the following example's input, you proceed with the call.

```
SYM (exit):
trap #0
```

Both `rom68k` and `bug` can handle a user-caused exception of `0` with no side effects. Although the `bug` monitor has a user-caused trap that returns control to the ROM monitor, the `bug` monitor is more portable.

The linker script

The linker script accomplishes the following processes to result.

- Sets up the memory map for the application.

  When your application is loaded into memory, it allocates some RAM, some disk space for I/O, and some registers. The linker script makes a memory map of this memory allocation which is important to embedded systems because, having no OS, you have the ability then to manage the behavior of the chip.
- For `g++`, sets up the constructor and destructor tables.

  The actual section names vary depending on your object file format. For `a.out` and `coff`, the three main sections are `.text`, `.data` and `.bss`.
- Sets the default values for variables used elsewhere.

  These default variables are used by `sbrk()` and the `crt0` file, typically called by `_bss_start` and `_end`.

There are two ways to ensure the memory map is correct.

- By having the linker create the memory map by using the option, `-Map`.
- By, after linking, using the `nm` utility to check critical addresses like `start`, `bss_end` and `_etext`.

The following is an example of a linker script for an `m68k`-based target board.

1. Use the `STARTUP` command, which loads the file so that it executes first.

```
STARTUP(crt0.o)
```

The `m68k-coff` configuration default does not link in `crt0.o` because it assumes that a developer has `crt0`. This behavior is controlled in the `config` file for each architecture in a macro called `STARTFILE_SPEC`. If `STARTFILE_SPEC` is set to NULL, `gcc` formats its command line and doesn't add `crt0.o`. Any filename can be specified with `STARTUP`, although the default is always `crt0.o`.

If you use only `ld` to link, you control whether or not to link in `crt0.o` on the command line.

If you have multiple `crt0` files, you can leave `STARTUP` out, and link in `crt0.o` in the makefile or use different linker scripts. Sometimes this option is used to initialize floating point values or to add device support.

2. Using `GROUP`, load the specified file.

```
GROUP(-lgcc-liop-lc)
```

In this case, the file is a relocated library that contains the definitions for the low-level functions needed by `libc.a`. The file to load could have also been specified on the command line, but as it's always needed, it might as well be here

as a default.

3. `SEARCH_DIR` specifies the path in which to look for files.

```
SEARCH_DIR(.)
```

4. Using `_DYNAMIC`, specify whether or not there are shared dynamic libraries. In the following example's case, there are no shared libraries.

```
__DYNAMIC = 0;
```

5. Set `_stack`, the variable for specifying RAM for the ROM monitor.
6. Specify a name for a section that can be referred to later in the script. In the following example's case, it's only a pointer to the beginning of free RAM space with an upper limit at 2M. If the output file exceeds the upper limit, `MEMORY` produces an error message. First, in this case, we'll set up the memory map of the board's stack for high memory for both the `rom68k` and `mon68k` monitors.

```
MEMORY
{
    ram     :     ORIGIN = 0x10000, LENGTH = 2M
}
```

***Setting up constructor and destructor tables for* g++**

1. Set up the `.text` section, using the following example's input.

```
SECTIONS
{
   .text :
   {
      CREATE_OBJECT_SYMBOLS
      *(.text)
      etext = .;
      __CTOR_LIST__ = .;
      LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
      *(.ctors)
      LONG(0)
      __CTOR_END__ = .;
      __DTOR_LIST__ = .;

      LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
      *(.dtors)
      LONG(0)
      __DTOR_END__ = .;
      *(.lit)
      *(.shdata) }
   > ram
   .shbss SIZEOF(.text) + ADDR(.text) : {
      *(.shbss)
   }
```

In a `coff` file, all the actual instructions reside in `.text` for also setting up the

constructor and destructor tables for g++. Notice that the section description redirects itself to the RAM variable that was set up in Step 5 of the earlier process for the crt0 file, “Set _stack, the variable for specifying RAM for the ROM monitor.” on page 12.

2. Set up the .data section.

```
        .talias : { } > ram
        .data : {
        *(.data)
        CONSTRUCTORS
        _edata = .;
    } > ram
```

In a coff file, this is where all of the initialized data goes. CONSTRUCTORS is a special command used by ld.

Setting default values for variables, _bss_start and _end

Set up the .bss section:

```
        .bss SIZEOF(.data) + ADDR(.data) :
        {
        __bss_start = ALIGN(0x8);
        *(.bss)
        *(COMMON)
           end = ALIGN(0x8);
           _end = ALIGN(0x8);
           __end = ALIGN(0x8);
        }
        .mstack : { } > ram
        .rstack : { } > ram
        .stab . (NOLOAD) :
        {
           [ .stab ]
        }
        .stabstr . (NOLOAD) :
        {
           [ .stabstr ]
        }
    }
```

In a coff file, this is where uninitialized data goes. The default values for _bss_start and _end are set here for use by the crt0 file when it zeros the .bss section.

I/O support code

Most applications use calls to the standard C library. However, when you initially link `libc.a`, several I/O functions are undefined. If you don't plan on doing any I/O, you're OK; otherwise, you need to create two I/O functions: `open()` and `close()`. These don't need to be fully supported unless you have a file system, so they are normally stubbed out, using `kill()`.

`sbrk()` is also a stub, since you can't do process control on an embedded system, only needed by applications that do dynamic memory allocation. It uses the variable, `_end`, which is set in the linker script.

The following routines are also used for optimization.

`-inbyte`
: Returns a single byte from the console.

`-outbyte`
: Used for low-level debugging, takes an argument for `print()` and prints a byte out to the console (typically used for ASCII text).

Memory support

The following routines are for dynamic memory allocation.

`sbrk()`
: The functions, `malloc()`, `calloc()`, and `realloc()` all call `sbrk()` at their lowest levels. `sbrk()` returns a pointer to the last memory address your application used before more memory was allocated.

`caddr_t`
: Defined elsewhere as `char *`.

`RAMSIZE`
: A compile-time option that moves a pointer to heap memory and checks for the upper limit.

Miscellaneous support routines

The following support routines are called by `newlib`, although they don't apply to the embedded environment.

`isatty()`
: Checks for a terminal device.

`kill()`
: Simply exits.

`getpd()`
: Can safely return any value greater than 1, although the value doesn't effect anything in `newlib`.

2

Overview of supported targets for cross-development

The following documentation describes programming practices and options for several of the embedded targets that GNUPro Toolkit supports. Since, by their very nature, the tools are evolving to meet the needs of Cygnus customers, new targets are frequently added (see the current matrix of supported embedded targets in *Introduction* in ***Getting Started with GNUPro Toolkit***).

The supported targets that are discussed can be found in the following documentation.

3

Hitachi H8/300, H8S, H8/300H development

The following documentation discusses cross-development with the Hitachi H8/300 H8S and H8/300 processors. All the H8 tools (300, 300H and S) are part of the same toolchain; older versions will not support the other two tools.

For more extensive documentation on the Hitachi H8/300, Hitachi Microsystems makes available the ***H8/300 Microcomputer User's Manual*** (Semiconductor Design & Development Center, 1992); contact your Field Application Engineer for details.

Cross-development tools in the GNUPro Toolkit 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 with the following input.

```
h8300-hms-gcc
```

Compiling for H8/300, H8S and H8/300H

The Hitachi 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 machine instructions, and whether to generate code for hardware or software floating point.

Using C++

There is 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()`), 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() {}
```

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 “GNU CC Command Options” in *Using GNU CC* in ***GNUPro Compiler Tools***.

Compiler options for H8/300

The following documentation discusses the compiler options.

`-ms`
: Generate code for the H8S processor.

`-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`:

- Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.
- An implementation of the standard C mathematical subroutine library. See "Mathematical Functions (math.h)" in *GNUPro Math Library* in ***GNUPro Libraries***.

Predefined preprocessor macros

gcc defines the following preprocessor macros for the Hitachi configurations:

Any Hitachi H8/300 architecture:

`__H8300__`

The Hitachi H8/300H architecture:

`__H8300H__`

Assembler options for H8/300, H8S and H8/300H

To use the GNU assembler to assemble `gcc` output, configure `gcc` with the switch, `--with-gnu-as` (in GNUPro Toolkit distributions) or with the `-mgas` option.

`-mgas`
: Compile using `as` to assemble `gcc` output.

`-Wa`
: If you invoke `as` 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 by commas like the options, `-alh` and `-L`, in the following example input separate from `-Wa`.

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

`-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, the assembler option, `-ahl`, requests a listing interspersed with high-level language and assembly language.

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

  '`-L`' preserves local labels, while the compiler debugging option , `-g`, gives the assembler the necessary debugging information.

Assembler options for listing output

Use the following options to enable *listing output from the assembler* (the letters after '`-a`' may be combined into one option, such as `-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`) also be requested.

Assembler listing-control directives

Use the following listing-control assembler directives to control the appearance of the listing output (if you do not request listing output with one of the '`-a`' options, the following listing-control directives have no effect).

`.list`
Turn on listings for further input.

`.nolist`
Turn off listings for further input.

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

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

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

`.sbttl`
Use 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.

Calling conventions for H8/300, H8S and H8/300H

The Hitachi family 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: Functions compiled with different calling conventions cannot be run together without some care.

Debugging for H8/300, H8S and H8/300H

The Hitachi-configured `gdb` is called with the following input.

```
h8300-hms-gdb
```

`gdb` needs to know the following specifications.

- Specifications for one of the following interfaces:

  `target remote`
  : GDB's generic debugging protocol, for using with the Hitachi low-cost evaluation board (**LCEVB**) running **CMON**.

  `target hms`
  : Interface to H8/300 `eval` boards running the HMS monitor.

  `target e7000`
  : E7000 in-circuit emulator for the Hitachi H8/300.

  `target sim`
  : Simulator, which allows you to run `gdb` remotely without an external device.

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

Use one of the following `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 previous list 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, the following example's sequence connects to the target board through a serial port, and loads and runs a program (designated as *`prog`* for variable-dependent input in the following example) through the debugger.

```
host$ h8300-hms-gdb prog
(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, designated here as *`hostname`*, is connected so that this makes sense; for instance, the connection may use a serial line, designated by your variable *`portnumber`* input, 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 variable, *n*, with the command, `remotedebug`.

In comparison to the H8/300, the H8S has the following improvements.

- Eight 16-bit expanded registers, and one 8-bit control register.
- Normal mode supports the 64K-byte address space.
- Advanced mode supports a maximum 16M-byte address space.
- Addressing modes of bit-manipulation instructions improved.
- Signed multiply and divide instructions.
- Two-bit shift instructions.
- Instructions for saving and restoring multiple registers.
- A test and set instruction.
- Basic instructions executing doublespeed.
- The H8S uses a two-channel on-chip PC break controller (PBC) for debugging programs with high-performance self-monitoring, without using an in-circuit emulator.
- The ROM is connected to the CPU by a 16-bit data bus, enabling both byte data and word data to be accessed in one state. This makes possible rapid instruction high-speed processing.
- The H8S has eight 32-bit *general registers*, all functionally alike for both address registers and data registers. When a general register is used as a data register, it can be accessed as a 32-bit, 16-bit, or 8-bit register.

  When the general registers are used as 32-bit registers or address registers, they use the letters, `ER` (`ER0` to `ER7`).

  The `ER` registers divide into 16-bit general registers designated by the letters, `E` (`E0` to `E7`) and `R` (`R0` to `R7`). These registers are functionally equivalent, providing a maximum 16 6-bit registers.

  The `E` registers (`E0` to `E7`) are also referred to as *extended registers*.

  The `R` registers divide into 8-bit general registers, using the letters, `RH` (`R0H` to `R7H`) and `RL` (`R0L` to `R7L`). These registers are functionally equivalent, providing a

maximum 16 8-bit registers.

- The *control registers* are the 24-bit program counter (`PC`), 8-bit extended control register (`EXR`), and 8-bit condition-code register (`CCR`).
- The H8S supports eight addressing modes. See Table 1.

Table 1: Addressing Modes

#	*Addressing Mode*	*Symbol*
1	Register direct	`Rn`
2	Register indirect	`@ERn`
3	Register indirect with displacement	`@(d:16,ERn)` `@(d:32,ERn)`
4	Register indirect with post-increment	`@ERn+`
	Register indirect with pre-decrement	`@-ERn`

5	Absolute address	`@aa:8`
		`@aa:16`
		`@aa:24`
		`@aa:32`

6	Immediate	`#xx:8`
		`#xx:16`
		`#xx:32`

7	Program-counter relative	`@(d:8,PC)` `@(d:16,PC)`
8	Memory indirect	`@@aa:8`

The upper 8 bits of the effective address are ignored, giving a 16-bit address.

- H8S initiates *exception handling* by a reset, a trap instruction, or an interrupt. Simultaneously generated exceptions are handled in order of priority. Exceptions originate from various sources. Trap instruction exception handling is always accepted in the program execution state. Trap instructions and interrupts are handled as in the following sequence.

  1. The program counter (`PC`), condition code register (`CCR`), and extend register (`EXR`) are pushed onto the stack.
  2. The interrupt mask bits are updated. The `T` bit is cleared to 0.
  3. A vector address corresponding to the exception source is generated, and program execution starts from that address.

  For a reset exception, use Step 2 and Step 3.

Loading on specific targets

With GNUPro Toolkit, downloading is possible to H8/300 boards and E7000 in-circuit emulators.

To communicate with a Hitachi H8/300 board, you can use the `gdb` remote serial protocol. See “The `gdb` remote serial protocol” in *Debugging with GDB* in ***GNUPro Debugging Tools*** for more details.

NOTE: The Hitachi **LCEVB** running **CMON** has the stub already built-in.

Use the following `gdb` command if you need to explicitly set the serial device.

```
device port
```

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 tutorial illustrates the steps needed to start a program under `gdb` control on an H8/300. The example uses a sample H8 program called ‘`t.x`’. The procedure is the same for other Hitachi chips in the series. First, hook up your development board. In the example that follows, we use a board attached to serial port, designated as `COM1`.

1. Call `gdb` with the name of your program as the argument, *`filename`*.
   ```
   gdb filename
   ```
2. `gdb` prompts you, as usual, with the following prompt.
   ```
   (gdb)
   ```
3. Use the following two special commands to begin your debugging session.

   `target hms` *`port`*
   Specify cross-debugging to the Hitachi board, and then use with the next input to download your program to the board.

   `load` *`filename`*
   `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`).

   The previous commands, specifically, `load`, are described in “Commands to specify files” in *Debugging with GDB* in ***GNUPro Debugging Tools***.
4. The following message for this `t.x` file then appears.

```
C:\H8\TEST> gdb t.x
GDB is free software and you are welcome to distribute copies
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 following `gdb` commands.

`break`
: Set breakpoints.

`run`
: Start your program.

`print`
: Display data.

`continue`
: Resume execution after stopping at a breakpoint.

`help`
: Display 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 to interrupt your program. Return to your program's process with the `(gdb)` command prompt after your program finishes its hanging. 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`" command to the program

To use the E7000 in-circuit emulator to develop code for either the Hitachi H8/300 or the H8/300H, use one of the following forms of the `target e7000` command to connect `gdb` to your E7000.

`target e7000` *`port speed`*
: Use this command if your E7000 is connected to a serial port. The *`port`* argument identifies what serial port to use (for example, `COM2`). The third argument, *`speed`*, is the line speed in bits per second (for example, input might be `9600`).

`target e7000` *`hostname`*
: If your E7000 is installed as a host on a TCP/IP network, substitute the network name for *`hostname`* during the connection. `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; instead, use the `ftpload` command.

4

Hitachi SH development

The following documentation discusses cross-development with the Hitachi SH processor.

Cross-development targets using the GNUPro Toolkit normally install 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 (calling `gcc` in native configurations) is named `sh-hms-gcc`.

For more documentation on the Hitachi SH, see ***SH Microcomputer User’s Manual*** (Semiconductor Design & Development Center, 1992) and ***Hitachi SH2 Programming Manual*** (Semiconductor and Integrated Circuit Division, 1994), from Hitachi SH Microsystems; contact your Field Application Engineer for details.

Compiling on 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.

Compiler options for SH

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 "GNU CC Command Options" in *Using GNU CC* in ***GNUPro Compiler Tools***.

Compiler options for architecture/code generation for SH

`-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.

`-m3e`
Generate code for the Hitachi SH-3E chip.

`-mhitachi`
Use Hitachi's calling convention rather than that for `gcc`. The registers, `MACH` and `MACL`, are saved with this setting (see "Calling conventions for SH targets" on page 37).

`-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 for SH

Two kinds of floating point subroutines are useful with `gcc`.

- Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.
- General-purpose mathematical subroutines.
  The GNUPro Toolkit from Cygnus includes an implementation of the standard C mathematical subroutine library. See “Mathematical Functions (`math.h`)” in *GNUPro Math Library* in ***GNUPro Libraries***.

Preprocessor macros for SH targets

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

Any Hitachi SH architecture:

`__sh__`

Any Hitachi SH1 architecture:

`__sh1__`

Any Hitachi SH2 architecture:

`__sh2__`

Any Hitachi SH3 architecture:

`__sh3__`

Any Hitachi SH3E architecture:

`__sh3e__`

Hitachi SH architecture with little-endian numeric representation:

`__little_endian__`

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

Assembler options for SH targets

The following documentation discusses the assembler options for the Hitachi SH processor.

General assembler options for SH

To use the GNU assembler to assemble `gcc` output, configure `gcc` with the switch, `--with-gnu-as` (in GNUPro Toolkit distributions) or with the `-mgas` option.

`-mgas`
Compile using `as` to assemble `gcc` output.

`-Wa`
If you invoke `as` 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 by commas like the options, `-alh` and `-L`, in the following example input separate from `-Wa`.

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

`-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, the assembler option, `-ahl`, requests a listing interspersed with high-level language and assembly language.

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

'`-L`' preserves local labels, while the compiler debugging option , `-g`, gives the assembler the necessary debugging information.

Assembler options for listing output for SH

Use the following options to enable *listing output from the assembler* (the letters after '`-a`' may be combined into one option, such as `-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`) also be requested.

Assembler listing-control directives for SH

Use the following listing-control Hitachi SH assembler directives to control the appearance of the listing output (if you do not request listing output with one of the '`-a`' options, the following listing-control directives have no effect).

`.list`
Turn on listings for further input.

`.nolist`
Turn off listings for further input.

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

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

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

`.sbttl`
Use 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.

Calling conventions for SH targets

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` and `R7`. Registers, `R0` through `R7`, as well as `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 `R1`, unchanged.

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

NOTE: Functions compiled with different calling conventions cannot be run together without some care.

Debugging on SH targets

The Hitachi SH-configured debugger, `gdb`, is called `sh-hms-gdb`.

`gdb` needs to know the following specifications to talk to your Hitachi SH.

- Specifications for one of the following interfaces:

  `target remote`
  : gdb's generic debugging protocol, for using with the Hitachi low-cost evaluation board (**LCEVB**) running **CMON**.

  `target hms`
  : Interface to SH `eval` boards running the HMS monitor.

  `target e7000`
  : E7000 in-circuit emulator for the Hitachi SH.

  `target sim`
  : Simulator, which allows you to run `gdb` remotely without an external device.

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

Use one of the following `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 previous list 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, the following example's sequence connects to the target board through a serial port, and loads and runs a program (designated as *`prog`* for variable-dependent input in the following example) through the debugger.

```
host$ sh-hms-gdb prog
(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, designated here as *`hostname`*, is connected so that this makes sense; for instance, the connection may use a serial line, designated by your variable *`portnumber`* input, 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 variable, *n*, with the command, `remotedebug`.

5

MIPS development

The following documentation discusses cross-development with the MIPS family of processors.

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

Cross-development tools in the GNUPro Toolkit 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 MIPS (using `gcc` in native configurations) is called by one of the following names, depending on which configuration you installed: `mips-ecoff-gcc` , if configured for big-endian byte ordering, and `mipsel-ecoff-gcc` , if configured for little-endian byte ordering.

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.

Compiler options for MIPS

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 "GNU CC Command Options" in *Using GNU CC* in ***GNUPro Compiler Tools***. There are a great many compiler options for specific MIPS targets. Options for architecture and code generation are for all MIPS targets (see "Options for architecture and code generation for MIPS").

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

Options for architecture and code generation for MIPS

The following options for architecture and code generation can be used on all MIPS targets.

`-g`

The compiler debugging option, `-g`, is essential to locate 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 MIPS boards are based on the MIPS R3000.

The default for this particular configuration is `-mcpu=r3000`.

In the general case, use `-mcpu=r3000` on any MIPS platform to assume the defaults for the machine type, *`cputype`*, when scheduling instructions.

The default, *`cputype`*, on other MIPS configurations is `r3000`, which picks the longest cycle times for any of the machines, in order that the code run at reasonable rates on any MIPS processor.

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, not 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.

Compiler options for floating point for MIPS

The following options select software or hardware floating point.

`-msoft-float`

Generate output containing library calls for floating point. The `mips-ecoff` configuration of `libgcc` (an auxiliary library distributed with the compiler) includes 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.

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 sub-routine 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 GNUPro Toolkit, you can select the standard libraries as usual with the options, '`-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 program's module 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`:

- ***Software implementations of the basic functions***
  Floating-point functionality for *multiply*, *divide*, *add*, *subtract* usage, used 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.

- ***General-purpose mathematical subroutines***
  GNUPro Toolkit includes an implementation of the standard C mathematical subroutine library. See “Mathematical Functions” in *GNUPro Math Library* in ***GNUPro Libraries***.

Preprocessor macros for MIPS targets

`gcc` defines the following preprocessor macros for the MIPS configurations.

Any MIPS architecture:

`__mips__`

MIPS architecture with big-endian numeric representation:

`__MIPSEB__`

MIPS architecture with little-endian numeric representation:

`__MIPSEL__`

Assembler options for MIPS targets

To use the GNU assembler to assemble `gcc` output, configure `gcc` with the `--with-gnu-as` or the `-mgas` option.

`-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 by commas like the options, `-alh` and `-L`, in the following example input separate from `-Wa`.

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

`-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, the assembler option, `-ahl`, requests a listing interspersed with high-level language and assembly language.

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

  '`-L`' preserves local labels, while the compiler debugging option , `-g`, gives the assembler the necessary debugging information.

Assembler options for listing output for MIPS

Use the following options to enable *listing output from the assembler* (the letters after '`-a`' may be combined into one option, such as `-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`) also be requested.

Assembler listing-control directives for MIPS

Use the following listing-control assembler directives to control the appearance of the listing output (if you do not request listing output with one of the '`-a`' options, the following listing-control directives have no effect).

`.list`
: Turn on listings for further input.

`.nolist`
: Turn off listings for further input.

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

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

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

`.sbttl`
: Use 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.

Special assembler options for MIPS

The MIPS configurations of `gas` support three special options, accepting one other for command-line compatibility. See "Command-Line Options" in *Using AS* in ***GNUPro Utilities*** 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 for *`num`* 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 the GNU assembler, there is no need for `-nocpp`, because the GNU assembler itself never runs the C preprocessor.

Assembler directives for debugging information

MIPS ECOFF using `gas` supports several directives for generating debugging information that 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 “Assembler Directives” in *Using AS* in ***GNUPro Utilities*** for full information on all GNU assembler 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 have the following definitions.

`.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 using `$gp`; the default value is `8`, meaning that a reference to any object eight bytes or smaller will use `$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. Use the following input, for example.

```
.extern sym, 4
```

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

Registers used for integer arguments for MIPS

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. If the first argument is an integer, MIPS uses the following registers for all arguments. The following calling convention for MIPS architectures depends on whether or not hardware floating-point is installed. Even if it is, MIPS uses the registers for integer arguments whenever the *first* argument is an integer. 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 MIPS also depends on whether standard 32-bit mode or Cygnus 64-bit mode is in use; 32-bit mode only allows MIPS to use even numbered registers, while 64-bit mode allows MIPS to use both odd and even numbered registers.

NOTE: Functions compiled with different calling conventions cannot be run together without some care.

- 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 are stored in `R2` and `R3`.

Registers used for floating-point arguments for MIPS

If the first argument is a floating-point, MIPS uses the following registers for floating-point arguments.

- In 32-bit mode, MIPS passes the first four words of arguments in registers `F12` and `F14`.
- In 64-bit mode, 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 for integer arguments for MIPS

The following 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 for floating-point arguments for MIPS

The following 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.

Debugging on MIPS targets

The MIPS-configured `gdb` uses the calling convention, `mips-ecoff-gdb`.

`gdb` needs to know the following things to talk to your MIPS target.

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

`mips-ecoff-gdb` uses the MIPS remote serial protocol to connect your development host machine to the target board.

Use one of the following `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, the following example's sequence connects to the target board through a serial port, and loads and runs a program (designated as *`prog`* for variable-dependent input in the following example) through the debugger.

```
host$ mips-ecoff-gdb prog
(gdb) target remote /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 board, designated here as *`hostname`*, is connected so that this makes sense; for instance, the connection may use a serial line, designated by your variable *`portnumber`* input, managed by a terminal concentrator).

`gdb` also supports the special command, `set mipsfpu off`, for MIPS targets.

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

If you neglect to use the command, `set mipsfpu off`, some calls will fail, such as `print strlen ("abc")`.

`set remotedebug` *n*

You can locate 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`.

Linking MIPS 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 information about US Software's floating point library, read ***US Software GOFAST R3000 Floating Point Library*** (United States Software Corporation).

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-ecoff-gcc -o prog prog.o...-lc fp.a
```

In a shared development environment, the following example may be more realistic.

IMPORTANT! The following documentation assumes you've installed the GOFAST library as *`user-dir`*`/libgofast.a`, where '`userdir`' is an apporpriate directory on your development system.

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

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

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

The GOFAST library is installed in the directory, *`userdir`*`/libgofast.a`, and the environment variable, `LIBRARY_PATH`, instructs `gcc` to look for the library in *`userdir`*. (The syntax shown here for setting the environment variable is the Unix Bourne Shell, `/bin/sh`, syntax; adjust as needed for your system.)

NOTE: 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 subroutine, since linking with the GOFAST 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 for MIPS

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. The following 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 the following instruction to the beginning of each affected function, then rebuilding the library.

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

6

Motorola `m68k` development

The following documentation discusses cross-development with the Motorola `m68k` targets.

Cross-development tools in the GNUPro Toolkit 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` (`gcc` in native configurations) is called, depending on which configuration you have installed, by `m68k-coff-gcc` or `m68k-aout-gcc`.

Compiling for 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.For information on all the gcc command-line options, see “GNU CC Command Options” in *Using GNU CC* in ***GNUPro Compiler Tools***.

`-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.

`-m68060`
: Generate code for the Motorola `m68060`. 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.

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.

Floating point subroutines

The following two kinds of floating point subroutines are useful with GCC.

- Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.
- General-purpose mathematical subroutines, included with implementation of the standard C mathematical subroutine library. See “Mathematical Functions” in *GNUPro Math Library* in ***GNUPro Libraries***.

Preprocessor macros for `m68k` targets

`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 `m68060` architecture:
  `__mc68060__`
- Any Motorola `m68332` architecture:
  `__mc68332__`
- Any Motorola `m68881` architecture:
  `__HAVE_68881__`

Assembler options for m68k targets

To use the GNU assembler, gas, to assemble gcc output, configure gcc with the --with-gnu-as switch or with the -mgas option.

-mgas
: Compile using **as** 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, like the options, -alh and -L, in the following example input, separate from -Wa.

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

-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, the assembler option ,-ahl, requests a listing with interspersed high-level language and assembly language.

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

  -L preserves local labels, while the compiler debugging option, -g, gives the assembler the necessary debugging information.

Assembler options for listing output

Use the following options to enable listing output from the assembler. The letters after '-a' may be combined into one option, such as '-al'.

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

-ah
: Requests a high-level language listing.

-al
: Request an output-program assembly listing.

-as
: Requests a symbol table listing.

-ad
: Omits debugging directives from listing. High-level listings require a compiler debugging option like -g, and assembly listings (such as -al) requested.

Assembler listing-control directives

Use the following listing-control assembler directives to control the appearance of the listing output (if you do not request listing output with one of the '`-a`' options, the following listing-control directives have no effect).

`.list`
Turn on listings for further input.

`.nolist`
Turn off listings for further input.

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

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

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

`.sbttl`
Use 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.

Calling conventions for `m68k` targets

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: Functions compiled with different calling conventions cannot be run together without some care.

Debugging on m68k targets

The m68k-configured gdb is called by m68k-coff-gdb or m68k-aout-gdb.

gdb needs to know the following specifications to talk to your Motorola m68k.

- Specifications for wanting to use one of the following interfaces:

  `target rom68k`
  : ROM monitor for the IDP board.

  `target cpu32bug`
  : ROM monitor for other Motorola boards, such as the Motorola Business Card Computer, BCC.

  `target est`
  : EST Net/300 emulator.

  `target remote`
  : gdb's generic debugging protocol.

- Specifications for what serial device connects your host to your m68k board (the first serial device available on your host is the default).
- Specifications for what speed to use over the serial device.

Use the following 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 previous list of specifications 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, the following sequence connects to the target board through a serial port, and loads and runs programs, designated here as *`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, designated here as *`hostname`*, is connected, for instance, to use a serial line, designated by *`portnumber`*, 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 variable, `remotedebug`.

7

PowerPC development

The following documentation discusses cross-development with the PowerPC targets.

Cross-development tools in the GNUPro Toolkit 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 PowerPC (`gcc` in native configurations) is called, depending on which configuration you have installed, by `powerpc-eabi-gcc`.

The following processors are supported for the PowerPC targets.

```
403Gx           603(e)
505             604
601             604(e)
602             821
603             860
```

Compiling for PowerPC targets

The PowerPC 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 PowerPC machine instructions, and whether to generate code for hardware or software floating point.

When you run gcc, you can use command-line options to choose machine-specific details.

These **-m** options are defined for the and PowerPC.

-mpower
-mno-power
-mpower2
-mno-power2
-mpowerpc
-mno-powerpc
-mpowerpc-gpopt
-mno-powerpc-gpopt
-mpowerpc-gfxopt
-mno-powerpc-gfxopt

GNU CC supports two related instruction set architectures for the IBM RS/6000 and PowerPC. The *POWER* instruction set are those instructions supported by the **rios** chip set used in the original RS/6000 systems and the *PowerPC* instruction set is the architecture of the Motorola MPC5*xx*, MPC6*xx*, MCP8*xx* and the IBM 4*xx* microprocessors. The PowerPC architecture defines 64-bit instructions, but they are not supported by any current processors.

Neither architecture is a subset of the other. However there is a large common subset of instructions supported by both. An MQ register is included in processors supporting the POWER architecture.

You use these options to specify which instructions are available on the processor you are using. The default value of these options is determined when configuring GNU CC. Specifying the '**-mcpu=***cpu_type*' overrides the specification of these options.

We recommend you use the '**-mcpu=***cpu_type*' option rather than any of these options.

The '**-mpower**' option allows GNU CC to generate instructions that are found only in the POWER architecture and to use the MQ register. Specifying '**-mpower2**' implies '**-power**' and also allows GNU CC to generate instructions that are present in the POWER2 architecture but not the original POWER architecture.

The '**-mpowerpc**' option allows GNU CC to generate instructions that are found

only in the 32-bit subset of the PowerPC architecture. Specifying '**-mpowerpc-gpopt**' implies '**-mpowerpc**' and also allows GNU CC to use the optional PowerPC architecture instructions in the General Purpose group, including floating-point square root. Specifying '**-mpowerpc-gfxopt**' implies '**-mpowerpc**' and also allows GNU CC to use the optional PowerPC architecture instructions in the Graphics group, including floating-point select.

If you specify both '**-mno-power**' and '**-mno-powerpc**', GNU CC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register. Specifying both '**-mpower**' and '**-mpowerpc**' permits GNU CC to use any instruction from either architecture and to allow use of the MQ register; specify this for the Motorola MPC601.

-mnew-mnemonics
-mold-mnemonics

Select which mnemonics to use in the generated assembler code.

'**-mnew-mnemonics**' requests output that uses the assembler mnemonics defined for the PowerPC architecture, while '**-mold-mnemonics**' requests the assembler mnemonics defined for the POWER architecture. Instructions defined in only one architecture have only one mnemonic; GNU CC uses that mnemonic irrespective of which of these options is specified.

PowerPC assemblers support both the old and new mnemonics, as will later POWER assemblers. Current POWER assemblers only support the old mnemonics. Specify '**-mnew-mnemonics** if you have an assembler that supports them, otherwise specify '**-mold-mnemonics**'.

The default value of these options depends on how GNU CC was configured. Specifying '**-mcpu=***cpu_type*' sometimes overrides the value of these option. Unless you are building a cross-compiler, you should normally not specify either '**-mnew-mnemonics**' or '**-mold-mnemonics**', but should instead accept the default.

-mcpu=*cpu_type*

Set architecture type, register usage, choice of mnemonics, and instruction scheduling parameters for machine type *cpu_type*. Supported values for *cpu_type* are '**rs6000**', '**rios1**', '**rios2**', '**rsc**', '**601**', '**602**', '**603**', '**603e**', '**604**', '**604e**', '**620**', '**power**', '**power2**', '**powerpc**', '**403**', '**505**', '**801**', '**821**', '**823**', '**860**' and '**common**'.

The '**-mcpu=power**', '**-mcpu=power2**', and '**-mcpu=powerpc**' specify generic POWER, POWER2 and pure PowerPC (i.e., not MPC601) architecture machine types, with an appropriate, generic processor model assumed for scheduling purposes.

Specifying '**-mcpu=rios1**', '**-mcpu=rios2**', '**-mcpu=rsc**', '**-mcpu=power**', or '**-mcpu=power2**' enables the '**-mpower**' option and disables the '**-mpowerpc**' option; '**-mcpu=601**' enables both the '**-mpower**' and '**-mpowerpc**' options; '**-mcpu=602**', '**-mcpu=603**', '**-mcpu=603e**', '**-mcpu=604**', '**-mcpu=620**'; '**-mcpu=403**', '**-mcpu=505**', '**-mcpu=821**', '**-mcpu=860**' and '**-mcpu=powerpc**'

enable the '**-mpowerpc**' option and disable the '**-mpower**' option; '**-mcpu=common**' disables both the '**-mpower**' and '**-mpowerpc**' options.

IBM AIX versions 4 or greater selects '**-mcpu=common**' by default, so that code will operate on all members of the IBM RS/6000 and PowerPC families. In that case, GNU CC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register. GNU CC assumes a generic processor model for scheduling purposes.

Specifying '**-mcpu=rios1**', '**-mcpu=rios2**', '**-mcpu=rsc**', '**-mcpu=power**', or '**-mcpu=power2**' also disables the '**new-mnemonics**' option.

Specifying '**-mcpu=601**', '**-mcpu=602**', '**-mcpu=603**', '**-mcpu=603e**', '**-mcpu=604**', '**-mcpu=620**', '**-mcpu=403**', or '**-mcpu=powerpc**' also enables the '**new-mnemonics**' option.

Specifying '**-mcpu=403**', '**-mcpu=821**', or '**-mcpu=860**' also enables the '**-msoft-float**' option.

-mtune=*cpu_type*
Set the instruction scheduling parameters for machine type, *cpu_type*, but do not set the architecture type, register usage, choice of mnemonics like '**-mcpu=***cpu_type*' would. The same values for *cpu_type* are used for '**-mtune=***cpu_type*' as for '**-mcpu=***cpu_type*'. The '**-mtune=***cpu_type*'option overrides the '**-mcpu=***cpu_type*' option in terms of instruction scheduling parameters.

-mfull-toc
-mno-fp-in-toc
-mno-sum-in-toc
-mminimal-toc
Modify generation of the TOC (Table Of Contents), which is created for every executable file. The '**-mfull-toc**' option is selected by default. In that case, GNU CC will allocate at least one TOC entry for each unique non-automatic variable reference in your program. GNU CC will also place floating-point constants in the TOC. However, only 16,384 entries are available in the TOC.

If you receive a linker error message that saying you have overflowed the available TOC space, you can reduce the amount of TOC space used with the '**-mno-fp-in-toc**' and '**-mno-sum-in-toc**' options.

'**-mno-fp-in-toc**' prevents GNU CC from putting floating-point constants in the TOC and '**-mno-sum-in-toc**' forces GNU CC to generate code to calculate the sum of an address and a constant at run-time instead of putting that sum into the TOC. You may specify one or both of these options. Each causes GNU CC to produce very slightly slower and larger code at the expense of conserving TOC space.

If you still run out of space in the TOC even when you specify both of these options, specify '**-mminimal-toc**' instead. This option causes GNU CC to make only one TOC entry for every file. When you specify this option, GNU CC will

produce code that is slower and larger but which uses extremely little TOC space. You may wish to use this option only on files that contain less frequently executed code.

-msoft-float
-mhard-float

Generate code that does not use or does use the floating-point register set. Software floating point emulation is provided if you use the '**-msoft-float**' option, and pass the option to GNU CC when linking.

-mmultiple
-mno-multiple

Generate code that uses (does not use) the load multiple word instructions and the store multiple word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use '**-mmultiple**' on little endian PowerPC systems, since those instructions do not work when the processor is in little endian mode.

-mstring
-mno-string

Generate code that uses (does not use) the load string instructions and the store string word instructions to save multiple registers and do small block moves. These instructions are generated by default on POWER systems, and not generated on PowerPC systems.

WARNING: Do not use **-mstring** on little endian PowerPC systems, since those instructions do not work when the processor is in little endian mode.

-mupdate
-mno-update

Generate code that uses (or does not use) the load or store instructions that update the base register to the address of the calculated memory location. These instructions are generated by default.

If you use '**-mno-update**', there is a small window between the time that the stack pointer is updated and the address of the previous frame is stored, which means code that walks the stack frame across interrupts or signals may get corrupted data.

-mfused-madd
-mno-fused-madd

Generate code that uses (does not use) the floating point multiply and accumulate instructions. These instructions are generated by default if hardware floating is used.

-mno-bit-align
-mbit-align

On System V.4 and embedded PowerPC systems do not and do force structures and unions containing bit fields aligned to the base type of the bit field. For example, by default a structure containing nothing but 8 unsigned bitfields of

length 1 would be aligned to a 4 byte boundary and have a size of 4 bytes. By using **-mno-bit-align**, the structure would be aligned to a 1 byte boundary and be one byte in size.

-mno-strict-align
-mstrict-align
On System V.4 and embedded PowerPC systems do not (do) assume that unaligned memory references will be handled by the system.

-mrelocatable
-mno-relocatable
On embedded PowerPC systems generate code that allows (does not allow) the program to be relocated to a different address at runtime. If you use **-mrelocatable** on any module, all objects linked together must be compiled with **-mrelocatable** or **-mrelocatable-lib**.

-mrelocatable-lib
-mno-relocatable-lib
On embedded PowerPC systems generate code that allows (does not allow) the program to be relocated to a different address at runtime. Modules compiled with '**-mreloctable-lib**' can be linked with either modules compiled without '**-mrelocatable**' and '**-mrelocatable-lib**' or with modules compiled with the '**-mrelocatable**' options.

-mno-toc
-mtoc
On System V.4 and embedded PowerPC systems do not (do) assume that register 2 contains a pointer to a global area pointing to the addresses used in the program.

-mno-traceback
-mtraceback
On embedded PowerPC systems do not (do) generate a trace-back tag before the start of the function. This tag can be used by the debugger to identify where the start of a function is.

-mlittle
-mlittle-endian
On System V.4 and embedded PowerPC systems compile code for the processor in little endian mode. The '**-mlittle-endian**' option is the same as '**-mlittle**'.

-mbig
-mbig-endian
On System V.4 and embedded PowerPC systems compile code for the processor in big endian mode. The '**-mbig-endian**' option is the same as '**-mbig**'.

-mcall-sysv
On System V.4 and embedded PowerPC systems compile code using calling conventions that adheres to the March 1995 draft of the System V Application Binary Interface, PowerPC processor supplement. This is the default unless you configured GCC using '**powerpc-*-eabiaix**'.

-mcall-sysv-eabi
Specify both '**-mcall-sysv**' and '**-meabi**' options.

-mcall-sysv-noeabi
Specify both '**-mcall-sysv**' and '**-mnoeabi**' options.

-mcall-aix
On System V.4 and embedded PowerPC systems compile code using calling conventions that are similar to those used on AIX. This is the default if you configured GCC using '**powerpc-*-eabiaix**'.

-mcall-solaris
On System V.4 and embedded PowerPC systems, compile code for the Solaris operating system.

-mcall-linux
On System V.4 and embedded PowerPC systems, compile code for the Linux operating system.

-mprototype
-mno-prototype
On System V.4 and embedded PowerPC systems assume that all calls to variable argument functions are properly prototyped. Otherwise, the compiler must insert an instruction before every non prototyped call to set or clear bit 6 of the condition code register (*CR*) to indicate whether floating point values were passed in the floating point registers in case the function takes a variable arguments.

With '**-mprototype**', only calls to prototyped variable argument functions will set or clear the bit.

-msim
On embedded PowerPC systems, assume that the startup module is called **sim-crt0.o** and the standard C libraries are **libsim.a** and **libc.a**. This is default for '**powerpc-*-eabisim**' configurations.

-mmvme
On embedded PowerPC systems, assume that the startup module is called **mvme-crt0.o** and the standard C libraries are '**libmvme.a**' and '**libc.a**'.

-memb
On embedded PowerPC systems, set the **PPC_EMB** bit in the ELF flags header to indicate that **eabi** extended relocations are used.

-mads
On embedded PowerPC systems, assume that the startup module is called '**crt0.o**' and the standard C libraries are '**libads.a**' and '**libc.a**'.

-myellowknife
On embedded PowerPC systems, assume that the startup module is called '**crt0.o**' and '**libyk.a**' and '**libc.a**' are the standard C libraries.

-meabi
-mno-eabi
On System V.4 and embedded PowerPC systems do (do not) adhere to the Embedded Applications Binary Interface (EABI) which is a set of modifications

to the System V.4 specifications. Selecting **-meabi** means that the stack is aligned to an 8 byte boundary, a function **__eabi** is called to from **main** to set up the EABI environment, and the '**-msdata**' option can use both **r2** and **r13** to point to two separate small data areas.

Selecting **-mno-eabi** means that the stack is aligned to a 16 byte boundary, do not call an initialization function from main, and the '**-msdata**' option will only use **r13** to point to a single small data area. The '**-meabi**' option is on by default if you configured GCC using one of the '**powerpc*-*-eabi***' options.

-msdata=eabi
On System V.4 and embedded PowerPC systems, put small initialized const global and static data in the '**.sdata2**' section, which is pointed to by register **r2**. Put small initialized non-const global and static data in the '**.sdata**' section, which is pointed to by register **r13**. Put small uninitialized global and static data in the '**.sbss**' section, which is adjacent to the '**.sdata**' section. The '**-msdata=eabi**' option is incompatible with the '**-mrelocatable**' option. The '**-msdata=eabi**' option also sets the '**-memb**' option.

-msdata=sysv
On System V.4 and embedded PowerPC systems, put small global and static data in the '**.sdata**' section, which is pointed to by register **r13**. Put small uninitialized global and static data in the '**.sbss**' section, which is adjacent to the '**.sdata**' section. The '**-msdata=sysv**' option is incompatible with the '**-mrelocatable**' option.

-msdata=default
-msdata
On System V.4 and embedded PowerPC systems, if '**-meabi**' is used, compile code the same as '**-msdata=eabi**', otherwise compile code the same as '**-msdata=sysv**'.

-msdata-data
On System V.4 and embedded PowerPC systems, put small global and static data in the '**.sdata**' section. Put small uninitialized global and static data in the '**.sbss**' section. Do not use register **r13** to address small data however.

This is the default behavior unless other '**-msdata**' options are used.

-msdata=none
-mno-sdata
On embedded PowerPC systems, put all initialized global and static data in the '**.data**' section, and all uninitialized data in the '**.bss**' section.

-G *num*
On embedded PowerPC systems, put global and static items less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss section. By default, *num* is 8. The '**-G** *num*' switch is also passed to the linker. All

modules should be compiled with the same
'**-G** *num*' value.

-mregnames
-mno-regnames

On System V.4 and embedded PowerPC systems, do (do not) emit register names in the assembly language output using symbolic forms.

Floating point subroutines for PowerPC

The following two kinds of floating point subroutines are useful with `gcc`.

- Software implementations of the basic functions (floating-point multiply, divide, add, subtract), for use when there is no hardware floating-point support.
- General-purpose mathematical subroutines, included with implementation of the standard C mathematical subroutine library. See "Mathematical Functions" in *GNUPro Math Library* in ***GNUPro Libraries***.

Preprocessor macros for PowerPC targets

`gcc` defines the following preprocessor macros for the PowerPC configurations.

- Any PowerPC architecture:
  `__powerpc-eabi__`

Assembler options for PowerPC targets

To use the GNU assembler, `gas`, to assemble `gcc` output, configure `gcc` with the `--with-gnu-as` switch or with the `-mgas` option.

`-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, like the options, `-alh` and `-L`, in the following example input, separate from `-Wa`.

```
$ powerpc-eabi-gcc -c -g -O -Wa,-alh, -L file.c
```

`-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, the assembler option ,`-ahl`, requests a listing with interspersed high-level language and assembly language.

```
$ powerpc-eabi-gcc -c -g -O -Wa,-alh,-L file.c
```

  `-L` preserves local labels, while the compiler debugging option, `-g`, gives the assembler the necessary debugging information.

Use the following options to enable listing output from the assembler. The letters after '`-a`' may be combined into one option, such as '`-al`'.

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

`-ah`
: Requests a high-level language listing.

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

`-as`
: Requests a symbol table listing.

`-ad`
: Omits debugging directives from listing. High-level listings require a compiler debugging option like `-g`, and assembly listings (such as `-al`) requested.

Use the following listing-control assembler directives to control the appearance of the listing output (if you do not request listing output with one of the ‘`-a`’ options, the following listing-control directives have no effect).

`.list`
: Turn on listings for further input.

`.nolist`
: Turn off listings for further input.

`.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 *`linecount`*. The variable input for *`columnwidth`* uses the same descriptive option.

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

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

`.sbttl`
: Use 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.

Debugging PowerPC targets

The `powerpc`-configured `gdb` is called by `powerpc-eabi-gdb`.

`gdb` needs to know the following specifications to talk to PowerPC targets.

- Specifications for what you want to use one, such as `target remote`, `gdb`'s generic debugging protocol.
- Specifications for what serial device connects your PowerPC board (the first serial device available on your host is the default).
- Specifications for what speed to use over the serial device.

Use the following `gdb` commands to specify the connection to your target board.

`target powerpc` *`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 previous list of specifications 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, the following sequence connects to the target board through a serial port, and loads and runs programs, designated here as *`prog`*, through the debugger.

```
(gdb) target powerpc com1
...
breakinst () ../sparc-stub.c:975
975     }
(gdb) s
main    ()   hello.c:50
50       writer(1,    "Got to here\n");
(gdb)
```

`target powerpc` *`hostname`*`:` *`portnumber`*

You can specify a TCP/IP connection instead of a serial port, using the syntax, *`hostname`*`:` *`portnumber`* (assuming your board, designated here as *`hostname`*, is connected, for instance, to use a serial line, designated by *`portnumber`*, managed by a terminal concentrator).

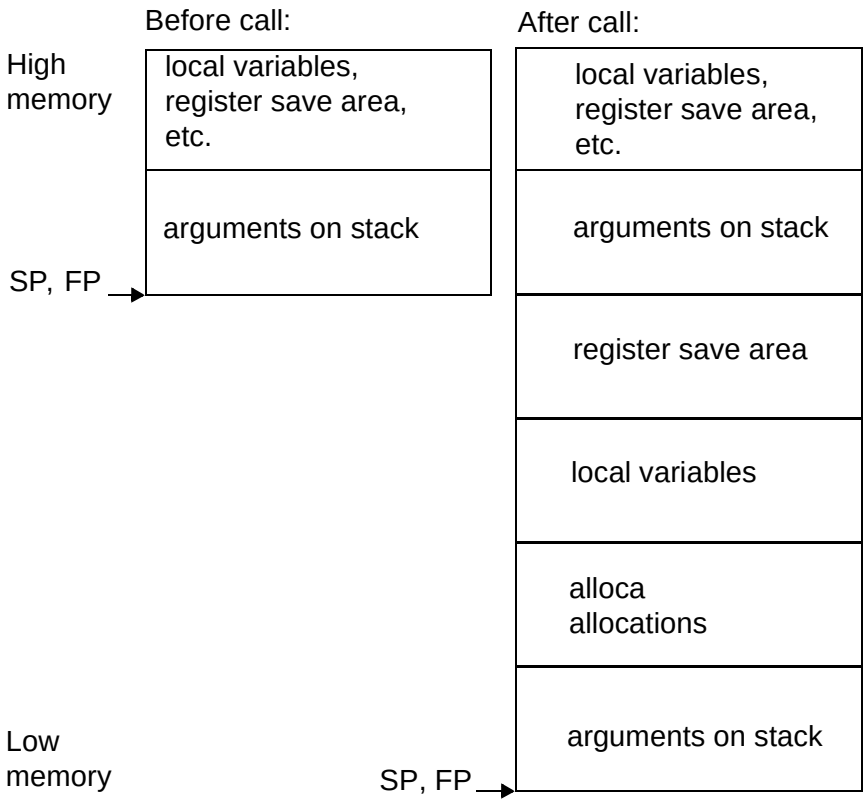
`gdb` also supports `set remotedebug` *`n`*. You can see some debugging information about communications with the board by setting the variable, `remotedebug`.

The stack frame

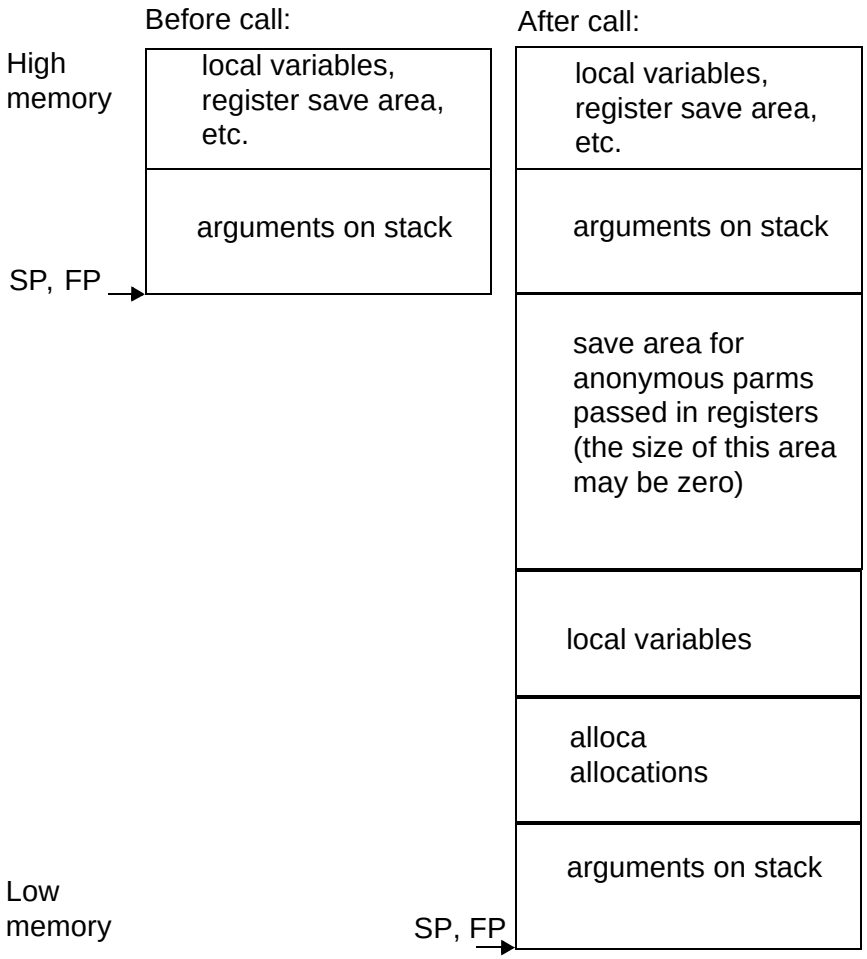
The following information applies to the stack frame for the PowerPC.

- The stack grows downwards from high addresses to low addresses.
- A leaf function need not allocate a stack frame if it does not need one.
- A frame pointer need not be allocated.
- The stack pointer shall always be aligned to 4 byte boundaries.
- The register save area shall be aligned to a 4 byte boundary.

Stack frames for functions taking a fixed number of arguments use the definitions in the following chart. FP points to the same location as SP.



Stack frames for functions that take a variable number of arguments use the following definitions.



Argument passing

The following table shows the general purpose registers, floating point registers, and the stack frame offset.

Figure 1: Parameter Passing Example Register

General Purpose Registers	*Floating-Point Registers*	*Stack Frame Offset*
r3: c	f1: ff	08: ptr to t
r4: d	f2: gg	0c: (padding)
r5: e	f3: hh	10: nn(lo)
r6: f	f4: ii	14: nn(hi)
r7: g	f5: jj	
r8: h	f6: kk	
r9: ptr to ld	f7: ll	
r10: ptr to s	f8: mm	

Function return values

Integers, floating point values, and aggregates of 8 bytes or less are returned in register ‘`r0`’ (and ‘`r1`’ if necessary).

Aggregates larger than 8 bytes are returned by having the caller pass the address of a buffer to hold the value in ‘`r0`’ as an “invisible” first argument. All arguments are then shifted down by one. The address of this buffer is returned in ‘`r0`’.

8

SPARC, SPARClite development

The following documentation discusses cross-development with the SPARC and SPARClite targets. For the `gcc` compiler in particular, special configuration options allow use of special software floating-point code for the SPARC MB86930 processor, as well as defaulting commnd-line options using special Fujitsu SPARClite features. For the FUjitsu SPARClite, there is support for the ex930, ex932, ex933, ex934, and the ex936 boards.

See the following documentation for more specific discussion concerning the SPARC and SPARClite targets.

Cross-development tools in the GNUPro Toolkit 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 (`gcc` in native configurations) is called, depending on which configuration you have installed, by `sparc-coff-gcc` or `sparc-aout-gcc`. The compiler for the SPARClite (`gcc` in native configurations) is called, depending on which configuration

you have installed, by `sparclite-coff-gcc` or `sparclite-aout-gcc`.

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

Compiling for 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.

Compiler options for SPARC

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 "GNU CC Command Options" in *Using GNU CC* in ***GNUPro Compiler Tools***.

`-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.

`-mvh`
: Generate code for the SPARC version 8. The only difference from version 7 code is the compiler emits the integer multiply (`smul` and `umul`) and integer divide (`sdiv` and `udiv`) instructions that exist in SPARC version 8 and not version 7.

`-mf930`
: Generate code for the Fujitsu SPARClite chip, MB86930. This chip is equivalent to the combination, `-msparclite -mno-fpu`. `-mf930` is the default when the compiler configures specifically to the Fujitsu SPARClite processor.

`-mf934`
: Generate code specifically intended for the SPARC MB86934, a Fujitsu SPARClite chip with a floating point .

  This option is equivalent to `-msparclite`.

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

`-msparclite`
: The SPARC configurations of GCC generate code for the common subset of the instruction set: the version 7 variant of the SPARC architecture.

  `-msparclite`, 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 version 8), the integer divide-step (`divscc`), and scan (`scan`) instructions that exist in SPARClite but not in SPARC version 7.

  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.

Options for floating point for SPARC and SPARClite

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

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

`-msoft-float`
`-mno-sfpu`
: 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 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, include the appropriate call 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 for SPARC and SPARClite

The following two kinds of floating point subroutines are useful with `gcc`.

- 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` 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, although slower, subroutines.
- General-purpose mathematical subroutines, included with implementation of the standard C mathematical subroutine library. See "Mathematical Functions" in *GNUPro Math Library* in ***GNUPro Libraries***.

Preprocessor macros for SPARC targets

`gcc` defines the following preprocessor macros for the SPARC configurations.

- Any SPARC architecture:
  `__sparc__`
- Any Fujitsu SPARClite architecture:
  `__sparclite__`

Assembler options for SPARC targets

To use the GNU assembler, `gas`, to assemble `gcc` output, configure `gcc` with the `--with-gnu-as` switch or with the `-mgas` option.

`-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, like the options, `-alh` and `-L`, in the following example input, separate from `-Wa`.

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

`-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, the assembler option ,`-ahl`, requests a listing with interspersed high-level language and assembly language.

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

`-L` preserves local labels, while the compiler debugging option, `-g`, gives the assembler the necessary debugging information.

Assembler options for listing output for SPARC, SPARClite

Use the following options to enable listing output from the assembler. The letters after '`-a`' may be combined into one option, such as '`-al`'.

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

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

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

`-as`
Requests a symbol table listing.

`-ad`
Omits debugging directives from listing. High-level listings require a compiler debugging option like `-g`, and assembly listings (such as `-al`) requested.

Assembler listing-control directives for SPARC, SPARClite

Use the following listing-control assembler directives to control the appearance of the listing output (if you do not request listing output with one of the '`-a`' options, the following listing-control directives have no effect).

`.list`
Turn on listings for further input.

`.nolist`
Turn off listings for further input.

`.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 *`linecount`*. The variable input for *`columnwidth`* uses the same descriptive option.

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

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

`.sbttl`
Use 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.

Assembler options for the SPARClite

When configured for SPARC, the assembler recognizes the additional Fujitsu SPARClite machine instructions that `gcc` generates: `-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`.

Calling conventions for SPARC and SPARClite

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`. Register `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: Functions compiled with different calling conventions cannot be run together without some care.

Debugging SPARC and SPARClite targets

The `sparc`-configured `gdb` is called by `sparc-coff-gdb` or `sparc-aout-gdb`.

The `sparclite`-configured `gdb` is called by `sparclite-coff-gdb` or `sparclite-aout-gdb`.

`gdb` needs to know the following specifications to talk to your SPARC or Fujitsu SPARClite.

- Specifications for what you want to use one, such as `target remote`, `gdb`'s generic debugging protocol.
- Specifications for what serial device connects your SPARC board (the first serial device available on your host is the default).
- Specifications for what speed to use over the serial device.

Use the following `gdb` commands to specify the connection to your 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` *`interface serial-device`*, where *`interface`* is an interface from the previous list of specifications 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, the following sequence connects to the target board through a serial port, and loads and runs programs, designated here as *`prog`*, through the debugger.

```
(gdb) target sparclite com1
[SPARClite appears to be alive]
breakinst () ../sparc-stub.c:975
975     }
(gdb) s
main    ()   hello.c:50
50       writer(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 board, designated here as *`hostname`*, is connected, for instance, to use a serial line, designated by *`portnumber`*, 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 variable, `remotedebug`.

Loading on specific targets for SPARC, SPARClite

The SPARC `eval` boards use a host-based terminal program to load and execute programs on the target. This program, `pciuh`, replaced the earlier 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 “The GDB remote serial protocol” in *Debugging with GDB* in ***GNUPro Debugging Tools*** for more details.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V