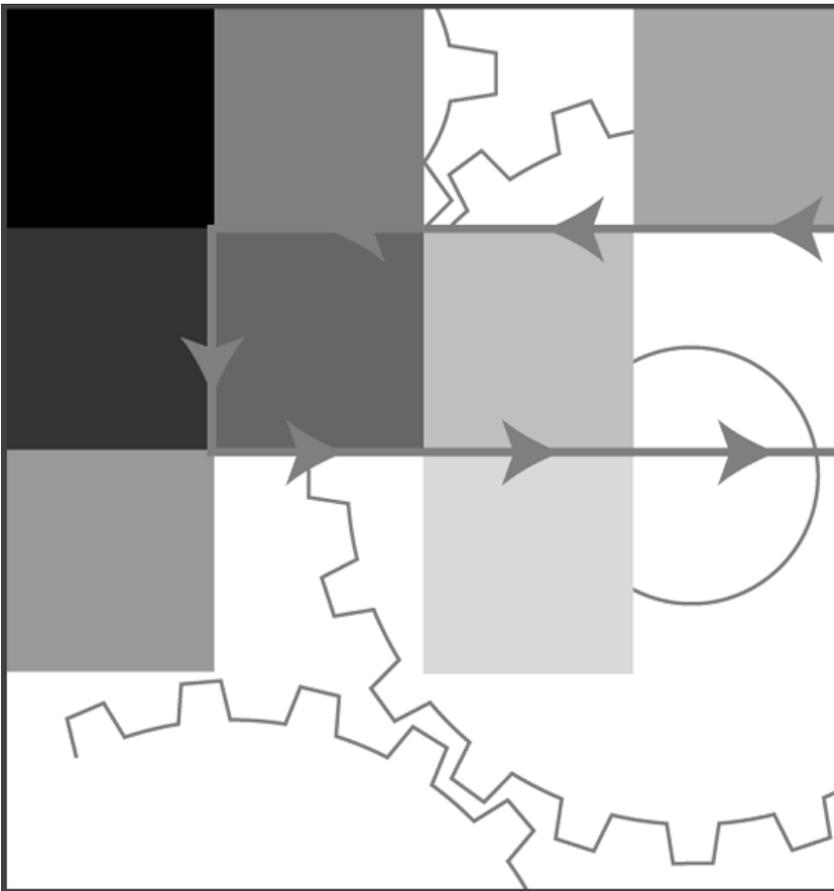




# **GNUPro<sup>®</sup> Toolkit User's Guide for Altera for ARM<sup>®</sup> and ARM/ Thumb<sup>®</sup> Development**



---

Copyright © 2002 Red Hat<sup>®</sup>, Inc. All rights reserved.

Red Hat<sup>®</sup>, GNUPro<sup>®</sup>, the Red Hat Shadow Man logo<sup>®</sup>, Insight<sup>™</sup>, Cygwin<sup>™</sup>, eCos<sup>™</sup>, RedBoot<sup>™</sup>, and Red Hat Embedded DevKit<sup>™</sup> are all trademarks of Red Hat, Inc.

ARM<sup>®</sup>, Thumb<sup>®</sup>, and ARM Powered<sup>®</sup>, SA<sup>™</sup>, SA-110<sup>™</sup>, SA-1100<sup>™</sup>, SA-1110<sup>™</sup>, SA-1500<sup>™</sup>, SA-1510<sup>™</sup> are trademarks of ARM Limited. All other brands or product names are the property of their respective owners. “ARM” is used to represent any or all of ARM Holdings plc (LSE; ARM: NASDAQ; ARMHY), its operating company, ARM Limited, and the regional subsidiaries ARM INC., ARM KK, and ARM Korea Ltd.

AT&T<sup>®</sup> is a registered trademark of AT&T, Inc.

Hitachi<sup>®</sup>, SuperH<sup>®</sup>, and H8<sup>®</sup> are registered trademarks of Hitachi, Ltd.

HP<sup>®</sup> and HP-UX<sup>®</sup> are registered trademarks of Hewlett-Packard, Ltd.

Intel<sup>®</sup>, Pentium<sup>®</sup>, StrongARM<sup>®</sup>, and XScale<sup>™</sup> are trademarks of Intel Corporation.

Linux<sup>®</sup> is a registered trademark of Linus Torvalds.

Microsoft<sup>®</sup> Windows<sup>®</sup> CE, Microsoft<sup>®</sup> Windows NT<sup>®</sup>, Microsoft<sup>®</sup> Windows<sup>®</sup> 98, and Win32<sup>®</sup> are registered trademarks of Microsoft Corporation.

Motorola<sup>®</sup> is a registered trademark of Motorola, Inc.

Sun<sup>®</sup>, SPARC<sup>®</sup>, SunOS<sup>™</sup>, Solaris<sup>™</sup>, and Java<sup>™</sup>, are trademarks of Sun Microsystems, Inc..

UNIX<sup>®</sup> is a registered trademark of The Open Group.

All other brand and product names, services names, trademarks and copyrights are the property of their respective owners.

Permission is granted to make and distribute verbatim copies of this documentation, provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions.

While every precaution has been taken in the preparation of this documentation, the publisher assumes no responsibility for errors, for omissions, or for damages resulting from the use of the information within the documentation. For licenses and use information, see “General Licenses and Terms for Using GNUPro Toolkit” in this GNUPro Toolkit *Getting Started Guide*.

## How to Contact Red Hat

Use the following means to contact Red Hat.

### *Red Hat Corporate Headquarters*

1801 Varsity Drive

Raleigh, NC 27606

Postal Mail: P.O. Box 13588, RTP, NC 27709

Telephone (toll free): +1 888 REDHAT 1 (+1 888 733 4281)

Telephone (main line): +1 919 754 3700

Telephone (FAX line): +1 919 754 3701

Website: <http://www.redhat.com/>

# Contents

---

<b>Introduction .....</b>	<b>1</b>
<b>Tutorials .....</b>	<b>21</b>
Create Source Code.....	22
Compile, Assemble, and Link from Source Code .....	22
Run the Executable on the Simulator.....	23
Debug with the Built-in Simulator.....	23
Debug with Insight.....	26
Produce an Assembler Listing from Source Code .....	35
A Guide to Writing Linker Scripts.....	36
Rebuild Tools for Windows Systems .....	39
Rebuild Tools for HP/UX, Linux, or Solaris Systems .....	41
<b>Reference.....</b>	<b>45</b>
Compiler Features .....	46
ABI Summary of Features .....	52
Assembler Features .....	57
Linker Features .....	61
Binary Utility Features.....	61
Debugger Features .....	63
Simulator Features .....	63
Cygwin Features .....	63
<b>Index .....</b>	<b>67</b>



# Introduction

---

GNUPro<sup>®</sup> Toolkit from Red Hat is a complete solution for C and C++ development for the ARM and ARM/Thumb processors, including a compiler, a debugger, binary utilities, libraries, and other tools. This documentation provides tutorials and references for the ARM and ARM/Thumb features of the main GNUPro tools. For initial information, see your release's top-level directory for a `README` file for installation and general configuration assistance. For general documentation, see <http://www.redhat.com/support/manuals/gnupro.html>.

Table 1 shows the supported host operating systems for ARM and ARM/Thumb processors. For each operating system, there is a standard naming convention, a *toolchain triplet*.

**Table 1:** Supported hosts for ARM and ARM/Thumb processors

<i>Processor</i>	<i>Operating system</i>	<i>Naming convention</i>
HPPA	HP/UX 10.20/11.0	hppa1.1-hp-hpux10.20/-hpux11.00
x86	Red Hat Linux 7.0, 7.1	i686-pc-linux-gnulibc2.1
x86	Windows 98/NT/2000	i686-pc-cygwin
SPARC	Solaris 2.6, 2.7, 2.8	sparc-sun-solaris2.5

GNUPro Toolkit uses names that reflect the processor and the object file format (see Table 2 for tool names for ARM processors; see Table 3 for tool names for ARM/Thumb processors). For example, with the ARM and ARM/Thumb processors, the object file formats are ELF (Executable and Linker Format; for more information

on ELF, see Chapter 4 of *System V Application Binary Interface* from Prentice Hall, 1990). When using a tool, its complete name uses a *triplet*, a hyphenated string, with its first part indicating the family (for ARM processors, `arm`; for ARM/Thumb processors, `thumb`), its second part indicating the object file format output (`elf`), and the last part indicating the tool name (for instance, for the compiler, use `gcc`); to call the GNU Compiler Collection (GCC) for the ARM processors, for example, use `arm-elf-gcc`.

The ARM and ARM/Thumb processors can use the tool names in Table 3.

**Table 2:** Tools for ARM processors and their names

<i>Tool description</i>	<i>Tool name (with ELF)</i>
GAS assembler (GAS)	<code>arm-elf-as</code>
GNU binary utilities	<code>arm-elf-ar</code>
	<code>arm-elf-nm</code>
	<code>arm-elf-objcopy</code>
	<code>arm-elf-objdump</code>
	<code>arm-elf-ranlib</code>
	<code>arm-elf-readelf</code>
	<code>arm-elf-size</code>
	<code>arm-elf-strings</code>
<code>arm-elf-strip</code>	
GNU compiler collection (GCC)	<code>arm-elf-gcc</code>
GNU debugger (GDB)	<code>arm-elf-gdb</code>
GNU linker (LD)	<code>arm-elf-ld</code>
Stand alone simulator	<code>arm-elf-run</code>

The ARM/Thumb processors can use the tool names in Table 3.

**Table 3:** Tools for ARM/Thumb processors and their names

<i>Tool description</i>	<i>Tool name (with ELF)</i>
GAS assembler (GAS)	<code>thumb-elf-as</code>
GNU binary utilities	<code>thumb-elf-ar</code>
	<code>thumb-elf-nm</code>
	<code>thumb-elf-objcopy</code>
	<code>thumb-elf-objdump</code>
	<code>thumb-elf-ranlib</code>
	<code>thumb-elf-readelf</code>
	<code>thumb-elf-size</code>
	<code>thumb-elf-strings</code>
<code>thumb-elf-strip</code>	
GNU compiler collection (GCC)	<code>thumb-elf-gcc</code>
GNU debugger (GDB)	<code>thumb-elf-gdb</code>
GNU linker (LD)	<code>thumb-elf-ld</code>
Stand alone simulator	<code>thumb-elf-run</code>

# Get the Tools to Work Properly

For Windows systems, libraries are installed in different locations, so you must use environmental settings for the tools to function properly; in the following examples, replace *installdir* with the default installation directory (see the README file at the top-level of directories for your release for the location and path), and replace *yymmdd* with the release date for your release (see the same README for date).

## Example 1: Environment variables for ARM and ARM/Thumb processors

```
SET PROOT=C:\installdir\arm-yymmdd
SET PATH=%PROOT%\H-i686-pc-cygwin\BIN;%PATH%
SET INFOPATH=%PROOT%\info
REM Set TMPDIR to point to a ramdisk if you have one
SET TMPDIR=%PROOT%
```

## Example 2: Environment variables for ARM/Thumb processors

```
SET PROOT=C:\installdir\thumb-yymmdd
SET PATH=%PROOT%\H-i686-pc-cygwin\BIN;%PATH%
SET INFOPATH=%PROOT%\info
REM Set TMPDIR to point to a ramdisk if you have one
SET TMPDIR=%PROOT%
```

Environmental settings do not need to be set for Red Hat Linux or UNIX toolchains (HP/UX and Sun Solaris systems).

# Use This Information Appropriately

The following strings are case sensitive: command line options, assembler labels, linker script commands, and section names. The following strings are not case sensitive: GDB commands, assembler instructions, and register names. By default, file names are not case sensitive for Windows systems. File names are case sensitive with Red Hat Linux and UNIX systems (HP/UX and Sun Solaris). File names are case sensitive when passed to GCC, regardless of the operating system.

The documentation uses some general conventions (see Table 4).

**Table 4:** Documentation’s conventions

<i>Style convention</i>	<i>Meaning</i>
<b>Bold Font</b>	Indicates menus, window names, and tool buttons.
<b><i>Bold Italic Font</i></b>	Indicates book titles, both hardcopy and electronic.
Plain Typewriter Font	Indicates code fragments, command lines, file contents, and command names; also indicates directory, file, and project names where they appear in text.
<i>Italic Typewriter Font</i>	Indicates a variable to substitute.
<b>Typewriter Font</b>	Indicates command lines, options, and text output generated by the program.

# Basic Information About the Tools

GNUPro Toolkit provides productivity, flexibility, performance and portability with its collection of development tools. For a summary of the tools, see the following documentation:

- “Compiler and Development Tools” on page 5
- “Libraries” on page 5
- “Auxiliary Development Tools” on page 6.

For general information about the main tools, see the following documentation:

- “gcc, the GNU Compiler Collection” on page 7
- “cpp, the GNU Preprocessor” on page 8
- “as, the GNU Assembler” on page 9
- “ld, the GNU Linker” on page 10
- “make, the GNU Recompiling Tool” on page 12
- “gdb, the Debugging Tool” on page 14
- “Insight, a GUI Debugger” on page 15
- “newlib and libstdc++, the GNU Libraries” on page 16
- “binutils, the GNU Binary Utilities” on page 17
- “Cygwin, for Porting UNIX Applications for Working on Windows Systems” on page 18
- “info, the Documentation Tools” on page 19

To use the tools, in your system’s console terminal shell window, enter the tool’s name as a command (`gcc`, for instance, invokes the compiler); for working with the tools, see “Tutorials” on page 21.

See <http://www.redhat.com/docs/manuals/gnupro/> for more general information.

# Compiler and Development Tools

The following tools are the main tools for developing projects with GNUPro Toolkit.

<b><i>Tool name</i></b>	<b><i>Usage</i></b>
cpp	C preprocessor (see “cpp, the GNU Preprocessor” on page 8; see also <i>The C Preprocessor</i> in <b><i>GNUPro Compiler Tools</i></b> )
diff diff3 sdiff	Comparison tools for text files (see <i>Using diff &amp; patch</i> in <b><i>GNUPro Development Tools</i></b> )
gcc	ISO-conforming compiler (see “gcc, the GNU Compiler Collection” on page 7; see also <i>Using GCC</i> in <b><i>GNUPro Compiler Tools</i></b> )
gcov	Coverage analyzer, for testing code for efficiency and performance, and for profiling (see <i>Using GCC</i> in <b><i>GNUPro Compiler Tools</i></b> )
gdb -nw	Debugger for making applications work better (see “gdb, the Debugging Tool” on page 14; see also <i>Debugging with GDB</i> in <b><i>GNUPro Debugging Tools</i></b> )
gdb	Debugger using a graphical user interface, a visual debugger, known as Insight (also conceptually known as <code>gdbtk</code> ; see “Insight, a GUI Debugger” on page 15 and “Debug with Insight” on page 26)
ld	Linker (see “ld, the GNU Linker” on page 10; see also <i>Using ld</i> in <b><i>GNUPro Development Tools</i></b> )
make	Compilation control program (see “make, the GNU Recompiling Tool” on page 12; see also <i>Using make</i> in <b><i>GNUPro Development Tools</i></b> )
patch	Installation tool for source fixes (see <i>Using diff &amp; patch</i> in <b><i>GNUPro Development Tools</i></b> )

## Libraries

See “Cygwin, for Porting UNIX Applications for Working on Windows Systems” on page 18; see also ***GNUPro Libraries*** for documentation regarding the following libraries.

<b><i>Tool name</i></b>	<b><i>Usage</i></b>
libc	ANSI C runtime library ( <i>only available for cross-development</i> )
libio	C++ iostreams library
libm	C math subroutine library ( <i>only available for cross-development</i> )
libstdc++	C++ class library, implementing the ISO 14882 Standard C++ library (see <a href="http://gcc.gnu.org/onlinedocs/libstdc++/documentation.html">http://gcc.gnu.org/onlinedocs/libstdc++/documentation.html</a> )

# Auxiliary Development Tools

GNUPro Toolkit also provides the following components for general development.

<b><i>Tool name</i></b>	<b><i>Usage</i></b>
as	Assembler (see “as, the GNU Assembler” on page 9; see also <i>Using as</i> in <i>GNUPro Auxiliary Development Tools</i> )
cygwin	Porting layer for making UNIX applications work for Windows systems (see “Cygwin, for Porting UNIX Applications for Working on Windows Systems” on page 18, “Cygwin Features” on page 63, and see <a href="http://sources.redhat.com/cygwin/">http://sources.redhat.com/cygwin/</a> )
info	Online documentation tools (see “info, the Documentation Tools” on page 19 and <i>Using info</i> in <i>GNUPro Auxiliary Development Tools</i> )
man	man pages, the standard UNIX online documentation

The GNU binary utilities provide functionality beyond the main development tools (see “binutils, the GNU Binary Utilities” on page 17; see also *Using binutils* in *GNUPro Auxiliary Development Tools*).

<b><i>Tool name</i></b>	<b><i>Usage</i></b>
addr2line	Converts addresses into file names and line numbers
ar	Creates, modifies and extracts from object code archives
c++filt	Demangles and deciphers encoded C++ symbol names
dlltool	Creates files for builds, using dynamic link libraries (DLLs)
nm	Lists symbols from object files
nlmconv	Converts object code into a Netware Loadable Module (NLM)
objcopy	Copies and translates object files
objdump	Displays information from object files
ranlib	Generates index to archive contents
readelf	Displays information about ELF format object files
size	Lists file section sizes and total sizes
strings	Lists printable strings from files
strip	Discards symbols
windres	Manipulates resources to use GNU tools on Windows systems (see <a href="http://sources.redhat.com/cygwin/">http://sources.redhat.com/cygwin/</a> )

---

# gcc, the GNU Compiler Collection

`gcc`, the GNU compiler collection (also known as GCC), is a complete set of tools for compiling programs written in C, C++, Objective C, or languages for which you have installed front-ends, invoking the GNU compiler passes with the following utilities.

- `as`, the GNU assembler that produces binary code from assembly language code and puts it in an object file (see “`as`, the GNU Assembler” on page 9)
- `cpp`, the GNU preprocessor that processes all the header files and macros which your target requires (see “`cpp`, the GNU Preprocessor” on page 8)
- `ld`, the GNU linker that binds the code to addresses, linking the startup file and libraries to an object file, and then producing an executable binary image (see “`ld`, the GNU Linker” on page 10)

To invoke the compiler, type:

```
gcc options
```

Providing *options* allows you to stop the compile process at intermediate stages. Use commas to separate the options.

There are many options available for providing a specific type of compiled output, some for preprocessing, others controlling assembly, linking, optimization, debugging, and still others for target-specific functions. For instance, call the driver with a `-v` option to see precisely which options are in use for each compilation pass.

There are four implicit file extensions: `.c` (for C source code which must be preprocessed), `.C` (for C++ source code which must be preprocessed), `.s` (for assembler code), and `.S` (for assembler code which must be preprocessed).

When you compile C or C++ programs, the compiler inserts a call at the beginning of `main` to a `__main` support subroutine. To avoid linking to standard libraries, specify the `-nostdlib` option (including `-lgcc` at the end of your compiler command line input resolves this reference, linking only with the compiler support library `libgcc.a`; ending your command’s input with it ensures that you get a chance to link first with any of your own special libraries). `__main` is the initialization routine for C++ constructors. All programs must have this call; otherwise, object files linked with a call to `main` might fail.

Compilation can involve up to four stages, always in the following order.

- *preprocessing*
- *compiling*
- *assembling*
- *linking*

The first three stages apply to an individual source file: *preprocessing* establishes the

type of source code to process, *compiling* produces an object file, *assembling* establishes the syntax that the compiler expects for symbols, constants, expressions and the general directives; the last stage, *linking*, completes the compilation process, combining all object files (newly compiled, and those specified as input) into an executable file.

For working with the GNU compiler and using its options, see *Using GCC* in *GNUPro Compiler Tools*.

## cpp, the GNU Preprocessor

`cp`, a macro preprocessor, works with the compiler collection to direct the parsing of C preprocessor directives. Preprocessing directives are the lines in your program that start with a `#` directive name (a `#` sign followed by an identifier). `cpp` merges `#include` files, for instance, then expands macro definitions, and processes `#ifdef` sections; another example is `#define`, a directive that defines a macro (`#define` must be followed by a macro name and the macro's intended expansion).

To see the output of `cpp`, invoke `gcc` with the `-E` option, and the preprocessed file will print on `stdout`. The C preprocessor provides the following four separate facilities.

- *Inclusion of header files*, with declarations that can be substituted into your program
- *Macro expansion*, for use in defining *macros*, which are abbreviations for arbitrary fragments of C code, which the C preprocessor will replace with definitions throughout a program
- *Conditional compilation*, using special preprocessing directives that include or exclude parts of a program, according to various conditions
- *Line control*, using a program to combine or rearrange source files into an intermediate file, which is then compiled, using line control to provide a source line's origin

There are two convenient options to assemble handwritten files that require preprocessing; both options depend on using the compiler driver program instead of directly calling the assembler.

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

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

---

# as, the GNU Assembler

`as`, the GNU assembler, is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture (such as with Intel processors), you find a fairly similar environment when you use it on another architecture. Each version of assembler has much in common with the others, including object file formats, most assembler directives, assembler syntax (symbols, constants, and expressions), and instructions for libraries, all the components which developers expect. The GNU assembler's primary function is to assemble the output of a source for the GNU compiler to use by the GNU linker or the GNU debugger in order to create an executable as the result.

The GNU assembler is useful as a way to pass your source code through the compiler or to examine it at source-level; the compiler then emits the code as a relocatable object file from the assembly language source code. The object file contains the binary code and the debug symbols from the source.

If you are invoking the GNU assembler using the GNU compiler, use the `-Wa` option to pass arguments through to the assembler. Usually you do not need to use the `-Wa` mechanism, since many compiler command line options are automatically passed to the assembler by the compiler.

The following example's input emits standard output with high-level and assembly source on a `file.c` file.

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

With the output, examine the components of the source code in the file. Every time you run the assembler, it produces an output file, which is your assembly language program translated into numbers. Conventionally, object file names end with `.o`.

## Use the Object File to Link Source Files

The object file is meant for input to the GNU linker. It contains assembled program code, information to help the linker integrate the assembled program into a file that can run; optionally, the object file provides symbolic information for the GNU debugger.

The GNU assembler can be configured to produce several alternative object file formats. For the most part, this does not affect how you write assembly language programs; but directives for debugging symbols are typically different in different file formats.

Source describes the program input with one run of the compiler with the assembler directives. The program may be in one or more files; how the source is partitioned into files doesn't change the meaning of the source. The source program is a concatenation of the text or content in all the files, in the order that you have specified. Each time

you run the compiler with the assembler, you assemble exactly one source program. The source program is made up of one or more files (the standard input is also a file.)

When compiling, you give the assembler command line input that has zero or more input file names. The input files are read (from left file name to right). A command line argument (in any position) that has no special meaning is taken to be an input file name. If you give the compiler no file names, it attempts to read one input file from the assembler's standard input. If the source is empty, the compiler produces a small, empty object file. There are two ways of locating a line in the input file (or files). One way refers to a line number in a *physical* file; the other refers to a line number in a *logical* file. Physical files are those files named in the command line given to the assembler. Logical files are simply names declared explicitly by assembler directives; they bear no relation to physical files. Logical file names help error messages reflect the original source file, when the assembler source is itself synthesized from other files.

If the assembler source is coming from the standard input (for instance, because it is being created by GCC using the `-pipe` command line option), then the listing will not contain any comments or preprocessor directives, since the listing code buffers input source lines from standard input only after they have been preprocessed by the assembler. This reduces memory usage and makes the code more efficient.

## Use Directives to Make the Source Assemble

Directives tell a compiler what to generate from a source; they have names that begin with a period (.). The rest of the name is letters, usually in lower case. Also commonly called a *pseudo-op*, a pseudo-operation, a directive is an instruction to the assembler that does not generate any machine code. The assembler resolves pseudo-ops during assembly, unlike machine instructions, which are resolved only at runtime.

Pseudo-ops are sometimes called assembler instructions, assembler operators, or assembler directives. In general, pseudo-ops give the assembler information about data alignment, block and segment definition, and base register assignment. The assembler also supports pseudo-ops that give the assembler information about floating point constants and symbolic debugger information (such as with `dbx`). While they do not generate machine code, the pseudo-ops can change the contents of the assembler's location counter.

For more information, see *Using as* in *GNUPro Auxiliary Development Tools*.

## ld, the GNU Linker

`ld`, the GNU linker, resolves the code addresses, object and archive files, relocates their data, links the startup code and additional libraries to the binary code, combines

symbol references, and, usually as the last step in compiling a program, produces an executable binary image. This means producing a *linker script* to control every link; such a script derives from the linker command language. The main purpose of a linker script is to describe how *sections* in the input files should map into the output file and to control memory layout of the output file. When necessary, the linker script also directs the linker to perform other operations. For an example of a linker script, see “A Guide to Writing Linker Scripts” on page 36.

The linker combines an output file and each input file in a special data format known as an *object file format*, with each file being an object file. The output file is often called an executable, but for simplicity, refer to it as an object file. Each object file has, among other things, a list of *sections*. `ld` reads many object files (partial programs) and combines their contents to form a program that will run. When the GNU assembler, `as`, emits an object file, the partial program is assumed to start at address, 0. Then, `ld` assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification of relocation, but it suffices to explain how `as` uses sections. `ld` moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called *relocation*. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses. A section is in an input file as an input section; similarly, a section in an output file is an output section. Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the *section contents*. A section may be marked as *loadable*, meaning to load the contents into memory when running the output file. A section with no contents may be *allocatable*, meaning to set aside an area in memory, but without loading anything there (in some cases this memory must be *zeroed out*).

A section, which is neither loadable nor allocatable, typically contains some sort of debugging information. Every loadable or allocatable output section has two addresses. The first is the *virtual memory address* (VMA), the address the section will have when the running the output file. The second is the *load memory address* (LMA), the address at which the section will load. In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts (this technique is often used to initialize global variables in a ROM-based system); in this case, the ROM address would be the LMA, and the RAM address would be the VMA. To see the sections in an object file, use the `objdump` binary utility with the `-h` option.

Every object file also has a list of *symbols*, known as the *symbol table*. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information. If you compile a C or C++ program into an object

file, you will get a defined symbol for every defined function and global or static variable. Every undefined function or global variable, which is referenced in the input file, will become an undefined symbol. You can see the symbols in an object file by using the `nm` binary utility, or by using the `objdump` binary utility with the `-t` option. The linker will use a default script that compiles into the linker executable, if you do not supply one. Use the `--verbose` option to display the default linker script. Certain options (such as `-r` or `-N`) will affect the default linker script. Supply your own linker script with the `-T` option. Use linker scripts implicitly by naming them as input files to the linker, as though they were files to be linked.

`ld` accepts linker command language files written in a superset of AT&T's *Link Editor Command Language* syntax, to provide explicit and total control over the linking process. `ld` uses the general purpose BFD libraries to operate on object files (libraries whose name derives from *binary file descriptors*); `ld` can then read, combine, and write object files in many different formats, such as COFF or `a.out` formats, for instance. You can link different formats to produce any available kind of object file. Aside from its flexibility, the GNU linker is more helpful than other linker in providing diagnostic information. Many linkers stop executing upon encountering an error, for example, whereas `ld` continues executing, whenever possible, allowing you to identify other errors (or, in some cases, to get an output file in despite the error).

For more information, see *Using ld* in *GNUPro Development Tools*.

## make, the GNU Recompiling Tool

`make`, the GNU recompiling tool, helps to determine automatically which pieces of a large program that you need to recompile. `make` then issues commands to recompile them. Originally implemented by Richard Stallman and Roland McGrath, `make` conforms to *IEEE Standard 1003.2-1992 (POSIX.2)*. `make` is compatible with any programming language whose compiler can run with command line input from a shell. `make` is not limited only to programs; it is also for any task where some files must update automatically whenever other files change with which those files associate.

To use `make`, you must write a file (a *makefile*) that describes the relationships among files in your program and provides commands for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files. When using `make` to recompile an executable, the result may change source files in a directory; if you changed a header file, to be safe, you must recompile each source file that includes that header file. Each compilation produces an object file corresponding to the source file. If any source file has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable.

`make` uses the makefile database and the last modified files to decide which of the other files needs updating. For each of those files, `make` implements the commands recorded in the data base of the makefile. The makefile has *rules*, which explain how and when to remake certain files that are the targets of a particular rule. A simple makefile has the following form for rules:

```
target ... : dependency ...
           command
```

*target* is usually the name of a file that a program generates; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as with the `clean` command (a command that, to simplify, deletes all files from a build directory before building). *dependency* is a file that is used as input to create the target. A target often depends on several files. *command* is for `make` to activate. A rule may have more than one command, with each command on its own line.

**IMPORTANT!** Provide a tabulation at the beginning of every command line.

Usually a command is in a rule with dependencies and serves to create a target file if any dependencies change. However, the rule that specifies commands for the target does not require dependencies; for instance, the rule containing the `delete` command that associates with the target, `clean`, does not have dependencies. `make` activates commands on the dependencies to create or to update the target. A rule can also explain how and when to activate a command. A makefile may contain other text besides rules; a simple makefile requires only rules. Rules generally follow the same pattern.

Example 3 shows a simplified makefile that describes the way an `edit` executable file depends on eight object files which, in turn, depend on eight C source and three header files. In Example 3, all of the C files include `defs.h`, but only those defining editing commands include `command.h`, and only low level files that change the editor buffer include `buffer.h` files.

**Example 3:** Makefile

```
edit : main.o kbd.o command.o display.o insert.o search.o files.o      \
                                           utils.o
      cc -o edit main.o kbd.o command.o display.o insert.o search.o    \
                                           files.o utils.o

main.o : main.c defs.h
      cc -c main.c
kbd.o : kbd.c defs.h command.h
      cc -c kbd.c
command.o : command.c defs.h command.h
      cc -c command.c
```

**Example 3: Makefile** (*cont'd*)

```
display.o : display.c defs.h buffer.h
    cc -c display.c
insert.o : insert.c defs.h buffer.h
    cc -c insert.c
search.o : search.c defs.h buffer.h
    cc -c search.c
files.o : files.c defs.h buffer.h command.h
    cc -c files.c
utils.o : utils.c defs.h
    cc -c utils.c
clean :
    rm edit main.o kbd.o command.o display.o insert.o search.o \
        files.o utils.o
```

The example makefile's targets include the executable file, `edit`, and the `main.o` and `kbd.o` object files. `main.c` and `defs.h` are the dependency files. Each `.o` file is both a target and a dependency. When a target is a file, it needs to be recompiled or relinked if any of its dependencies change. Update any dependencies that automatically generate first. In Example 3, `edit` depends on eight object files; the object file, `main.o`, depends on the source file, `main.c`, and on the `defs.h` header file. A shell command follows each line that contains a target and dependencies, saying how to update the target file; a tab character must come at the beginning of every command line to distinguish command lines from other lines in the makefile. `make` does not know anything about how the commands work; it is up to you to supply commands that will update the target file properly. All `make` does is execute the commands in the rule you have specified when the target file needs updating.

For more details, see *Using make* in *GNUPro Development Tools*.

## gdb, the Debugging Tool

`gdb`, the GNU debugger, allows you to stop your program before it terminates. When your program stops, you must determine where it stopped and how it got there. With a command line approach when compiling on a `file.c` file, use `gcc -g -o directory file.c` as a command; `-g` produces the debugging information. Then, run the debugger, using the `arm-elf-gdb file.c` command, to debug on the `directory's file.c` file. See “Debug with the Built-in Simulator” on page 23 to debug, see “Debug with Insight” on page 26 for an introduction to using the graphical user interface for the GNU debugger. See also RedBoot's own documentation<sup>1</sup>.

Set breakpoints with the `breakpoint` command.

Navigate through the program with the `step` command or the `next` command.

The debugger debugs threads, signals, trace information, and other data in a program. Each time your program performs a function call, information generates about the call, a block of data (the *stack frame*), which shows the location of the call, the arguments, and the local variables of a function. This debugger examines the stack frame to get your program to work.

To create more efficient, faster running code, before debugging, use profiling (with the `gprof` tool) and test coverage (with the `gcov` tool) to analyze your programs.

Use a `gdb backend` with its standard remote protocol. The backend's standard remote protocol is for transmitting packets of data when communicating with a target and finding errors when debugging a program. Similar protocols will suffice, as long as they provide for the debugger to have reading and writing of registers and memory, being able to start execution at an address, single stepping, being able to read a last signal, and, often, resetting the hardware. The following two types of backend are the most common:

- A *stub* (a subroutine) that serves as an exception handler for breakpoints; it must link to your application. Stubs use the standard remote protocol.
- An existing ROM monitor used as a backend; the most common approach means using the following processes:
  - With a similar protocol to the standard remote protocol
  - With an interface that directly uses the ROM monitor; with such an interface, the debugger only formats and parses commands.

All the ROM monitor interfaces share a common set of routines.

For more information on the GNU debugger, see *Debugging with GDB* in **GNUPro Debugger Tools**. See also “Debug with Insight” on page 26 and *Insight, the GNUPro Debugger GUI Interface* in **GNUPro Debugger Tools**.

## Insight, a GUI Debugger

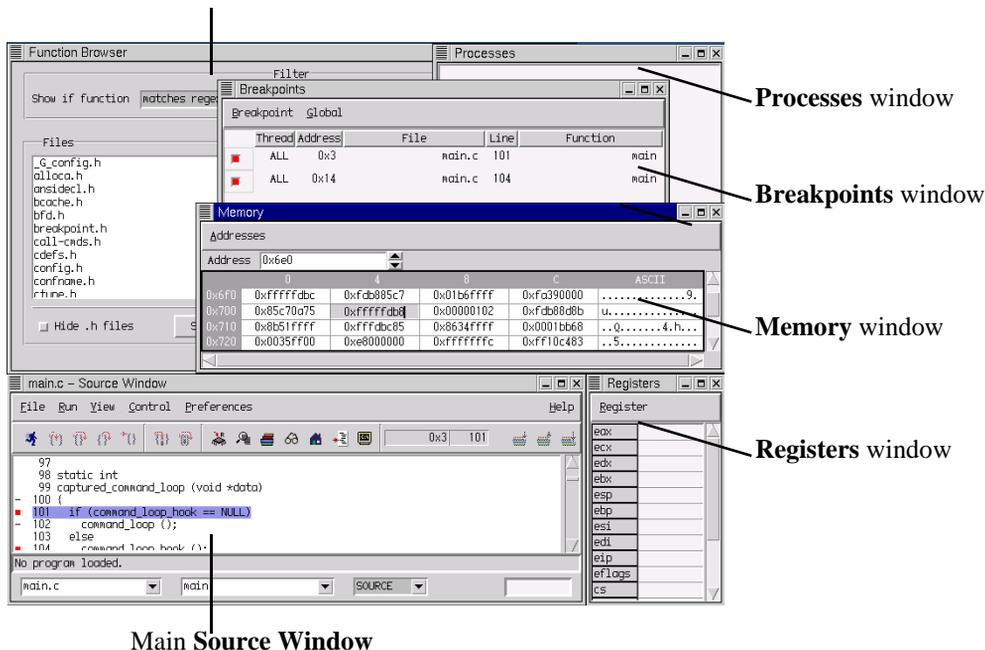
Besides the standard command line based debugger, GNUPro Toolkit includes Insight, a graphical user interface. Insight works on a range of host systems and target microprocessors, allowing development with complete access to the program state, for source and assembly level, with the ability to manage breakpoints, variables, registers,

---

<sup>1</sup> The RedBoot documentation is in `redboot-altera-vR1_34-2.tar.gz`. When you open this archive, it will create a `RedBoot_vR1_34-2` directory. Under that directory is, among other things, a `readme.txt` file and a `docs` subdirectory. In the `docs` directory you'll see `redboot/html` and `redboot/pdf`. They're the same documents in the named formats. You should be able to find everything you need in these documents.

memory, threads and other functionality. Adding a series of intuitive views into the debugging process, Insight provides you with a wide range of system information. See Figure 1 for an example of the windows that Insight provides for analyzing and debugging programs.

**Figure 1:** A composite view of working with Insight  
**Function Browser** window



For developing with Insight, see “Debug with Insight” on page 26.

## newlib and libstdc++, the GNU Libraries

newlib and libstdc++, the standard GNU libraries, serve as a collection of subroutines and functions, in compiled form, which link with a program to form a complete executable, linking either statically or, with some systems, dynamically.

newlib includes the GNU C library, libc, and the GNU C math library, libm.

See *GNUPro C Library* and *GNUPro C Math Library* for newlib functions in *GNUPro Libraries*.

libstdc++ is based on the ISO 14882 Standard C++ Library, with all its compliant classes and functions.

See <http://sources.redhat.com/libstdc++/links.html> for the libstdc++ library documentation.

---

# binutils, the GNU Binary Utilities

binutils, the GNU binary utilities, are available on all hosts. They include `ar`, `nm`, `objcopy`, `objdump`, `ranlib`, `readelf`, `size`, `strings`, and `strip`. There are three binary utilities, `addr2line`, `windres`, and `dlltool`, which are for use with Cygwin, the porting layer application for Win32 development. The most important of the binary utilities are `objcopy` and `objdump`.

- `objcopy`  
For a few ROM monitors (such as `a.out`), `objcopy` allows for loading executable binary images and, consequently, loading an S-record. An S-record is a printable ASCII representation of an executable binary image. S-records are suitable for both building ROM images for standalone boards and downloading images to embedded systems. Use the following example's input for this process.

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

`infile` 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 `objdump -i` as input to get a list of supported object file types for your architecture. For making an executable binary image, see “`objcopy` Utility” in *Using binutils in GNUPro Auxiliary Development Tools*.

- `objdump`  
`objdump` lets you display information about one or more object files, with options controlling particular information to display when working on the compilation tools. When specifying archives, `objdump` shows information on each of the member object files. `objfile...` designates the object files to be examined; for more information, see “`objcopy` Utility” in *Using binutils in GNUPro Auxiliary Development Tools*.

A few of the more useful options for commands are: `-d`, `--disassemble`, and `--prefix-addresses`. `-d` and `--disassemble` display assembler mnemonics for the machine instructions from `objfile`; they only disassemble those sections that are expected to contain instructions. `--prefix-addresses`, for disassembling, prints a complete address on each line, starting each output line with the address that it disassembles; it is an older disassembly format (otherwise, you only get raw opcodes).

For more information on binutils, see *Using binutils in GNUPro Auxiliary Development Tools*.

# Cygwin, for Porting UNIX Applications for Working on Windows Systems

Cygwin, a full-featured Win32 porting layer for UNIX applications, is compatible with all Win32 hosts (currently, these are Microsoft Windows NT/95/98 systems). With Cygwin, you can make all directories have similar behavior, with all the UNIX default tools in their familiar place. Shells include `bash`, `ash`, and `tcsh`. Tools such as Perl, Tcl/Tk, `sed`, `awk`, `vim`, Emacs, `xemacs`, `telnetd` and `ftpd`.

In order to emulate a UNIX kernel to access all processes that can run with it, use the Cygwin DLL (dynamically linked library). The Cygwin DLL will create shared memory areas so that other processes using separate instances of the DLL can access the kernel. Using DLLs with Cygwin, link into your program at run time instead of build time. The following documentation describes the three parts of a DLL and their usage.

- *exports*, a list of functions and variables that the `.dll` file makes available to other programs as a list of *global* symbols, with the rest of the contents hidden. Create this list with a text editor; it's also possible to do it automatically from the list of functions in your code. The `dlltool` utility creates the exports section of the `.dll` file from your text file of exported symbols.
- *code and data*, the parts you write, along with the functions, variables, and so forth, merged together, building one object file and linking it to a `.dll` file; they are not put into a `.exe` file.
- *import library*, a regular UNIX-like `.a` library, containing the vital information for the operating system and the program to interact as it or *imports* the `.dll` as data, linking the data into an `.exe` file, all generated by the `dlltool` utility.

The following example shows a the use of the `compile` command with `gcc`, demonstrating how to build a `.dll` file, using a single `myprog.c` file, for a `myprog.exe` program, with a single `mydll.c` file, for the contents of a `.dll` file, with the resulting `mydll.dll` file then compiling everything as objects.

```
gcc -c myprog.c
gcc -c mydll.c
```

See “Cygwin Features” on page 63 for more basic information, and see “Building and Using DLLs with Cygwin” on page 64 for more explanation of linking with the `dlltool` tool. Find the `~/cygwin/doc` directory to locate documentation discussing use of the GNU development tools with a Win32 host and exploring the architecture of the Cygwin library. See <http://sources.redhat.com/cygwin/> for more general documentation.

# info, the Documentation Tools

`info` provides the sources for documentation for the GNU tools; it requires the following tools, which include the  $\text{T}\text{E}\text{X}$  tools.

- `Texinfo`<sup>2</sup>, `texindex`, `texi2dvi`, the standard GNU documentation formatting tools.
- `makeinfo`, `info`, the GNU online documentation tools.
- `man pages`, the GNU documentation on all the tools and programs in this release.
- *FLEX: A Fast Lexical Analyzer Generator*<sup>3</sup>, which generates lexical analyzers suitable for GCC and other compilers.
- *Using and Porting GCC*, information about requirements for putting GCC on different platforms, or for modifying GCC; includes documentation from *Using GCC* (in *GNUPro Compiler Tools*).
- *BYacc*<sup>4</sup>, the discussion of the Berkeley Yacc parser generator.
- *Texinfo: The GNU Documentation Format*, the documentation that details  $\text{T}\text{E}\text{X}$  and the printing and generating of documentation, as well as how to write manuals in the  $\text{T}\text{E}\text{X}$  style.
- *Configuration program*, descriptions of the configuration program that GNUPro Toolkit uses.
- *GNU Coding Standards*, the more elaborate details on the coding standards with which the GNU projects develop.
- *GNU gprof*, details of the GNU performance analyzer (only for some systems).

You have the freedom to copy the documentation using its accompanying copyright statements, which include necessary permissions. To get the documentation in HTML or printable form, see <http://www.fsf.org/doc/doc.html> and <http://www.fsf.org/doc/other-free-books.html>.

See *Using info* in *GNUPro Auxiliary Development Tools* for documentation regarding these tools.

## Reading info Documentation

Browse through the documentation using either Emacs or the `info` documentation

<sup>2</sup> Requires  $\text{T}\text{E}\text{X}$ , the free technical documentation formatting tool written by Donald Knuth. See *Texinfo: The GNU Documentation Format* (ISBN: 1-882114 67 1).

<sup>3</sup> See *Flex: The Lexical Scanner Generator* (ISBN: 1-882114 21 3).

<sup>4</sup> See *Bison Manual: Using the YACC-compatible Parser Generator* (ISBN: 1-882114 44 2).

browser program. The information is in *nodes*, corresponding to the sections of a printed book. Follow them in sequence, as in books, or, using the hyperlinks, find the node that has the information you need. `info` has *hot* references (if one section refers to another section, `info` takes you directly to that other section with the capability to return easily to reading where you had been). You can also search for particular words or phrases. After installing GNUPro Toolkit, use `info` by typing its name at a shell prompt; no options or arguments are necessary. Check that `info` is in your shell path after you install GNUPro Toolkit. If you have problems running `info`, contact your system administrator.

To get help with using `info`, type `h` for a programmed instruction sequence, or **Ctrl+h** for a short summary of commands. To stop using `info`, type `q`.

See “Reading `info` Files” in *Using info in GNUPro Auxiliary Development Tools* for detailed references of the `info` tools.

# 1

## Tutorials

---

The following documentation gives examples of how to use the tools using command line mode from a shell window for UNIX or Windows operating systems. When text discusses UNIX systems, the referent systems include HPUX, Red Hat Linux, and Sun Solaris systems. For specific operating system information naming conventions, see Table 1 on page 1. See <http://www.redhat.com/docs/manuals/gnupro/> for more general information about the tools.

**NOTE:** Please be advised that your screen output may vary from that shown, depending on your environment.

- “Create Source Code” on page 22
- “Compile, Assemble, and Link from Source Code” on page 22
- “Run the Executable on the Simulator” on page 23
- “Debug with the Built-in Simulator” on page 23
- “Debug with Insight” on page 26
- “Produce an Assembler Listing from Source Code” on page 35
- “A Guide to Writing Linker Scripts” on page 36
- “Rebuild Tools for Windows Systems” on page 39
- “Rebuild Tools for HP/UX, Linux, or Solaris Systems” on page 41

**IMPORTANT!** Remember that GNUPro Toolkit tools are case sensitive, so enter all

commands and options as indicated. The examples show the ELF toolchain being used, similar output would be generated if the COFF format were used.

## Create Source Code

Create the following sample source code and save it as `hello.c` to verify correct installation and use of the tools.

```
#include <stdio.h>

int a, c;

void foo(int b)
{
    c = a + b;
    printf("%d + %d = %d\n", a, b, c);
}

int main()
{
    int b;

    a = 3;
    b = 4;
    printf("Hello, world!\n");
    foo(b);
    return 0;
}
```

## Compile, Assemble, and Link from Source Code

To compile code to run on the simulator, use the following example's input

On Windows, for ARM processors, type:

```
arm-elf-gcc -no-target-default-spec -g hello.c -o hello.exe
```

On UNIX, for ARM processors, type:

```
arm-elf-gcc -no-target-default-spec -g hello.c -o hello.x
```

On Windows, for ARM/Thumb processors, type:

```
thumb-elf-gcc -no-target-default-spec -g hello.c -o hello.exe
```

On UNIX, for ARM/Thumb processors, type:

```
thumb-elf-gcc -no-target-default-spec -g hello.c -o hello.x
```

For the previous examples, the `-no-target-default-spec` option is required to build

an executable that will run on the simulator. The `-g` option generates debugging information and the `-o` option specifies the name of the executable to be produced. Other useful options include `-O` for standard optimization, and `-O2` for extensive optimization. When no optimization option is specified, GCC will not optimize. To build an executable that will run on the excalibur board you do not need to use the option `-no-target-default-spec` as the default spec file is for the excalibur board. To compile for another board you should use the options `-no-target-default-spec` along with `--specs=board.spec` where `board.spec` is the spec file for the board you are compiling to. See “GNU CC Command Options” in *Using GNU CC* in *GNUPro Compiler Tools* for a complete list of available options.

## Run the Executable on the Simulator

To debug the program on the stand-alone simulator, use the following example’s input:

On Windows, for ARM and ARM/Thumb processors, type:

```
arm-elf-run hello.exe
```

On UNIX, for ARM and ARM/Thumb processors, type:

```
arm-elf-run hello.x
```

On Windows, for ARM/Thumb processors, type:

```
thumb-elf-run hello.exe
```

On UNIX, for ARM/Thumb processors, type:

```
thumb-elf-run hello.x
```

The program generates:

```
hello world!  
3 + 4 = 7
```

The simulator executes the program, and returns when the program exits.

## Debug with the Built-in Simulator

GDB can be used to debug executables using the simulator. To start GDB, use the following commands:

On Windows, for ARM processors, type:

```
arm-elf-gdb -nw hello.exe
```

On UNIX, for ARM processors, type:

```
arm-elf-gdb -nw hello.x
```

On Windows, for ARM/Thumb processors, type:

```
thumb-elf-gdb -nw hello.exe
```

On UNIX, for ARM/Thumb processors, type:

```
thumb-elf-gdb -nw hello.x
```

For the previous examples, `-nw` is for selecting the command line interface to GDB (the Insight interface is the default; for more information, see “Debug with Insight” on page 26); the command line shell is useful when you wish to report a problem you have with GDB, since a sequence of commands is simpler to reproduce. After the initial copyright and configuration information, GDB returns its own prompt, (`gdb`). The following is a sample debugging session using the `target sim` command to specify the simulator as the target.

1. To specify the target to debug on, in this case the `sim` simulator, type:

```
target sim
```

The following output displays:

```
Connected to the simulator.
```

2. To load the program into memory, type:

```
load
```

Output similar to the following will be displayed:

```
Loading section .init, size 0x10 lma 0x0
Loading section .text, size 0xad6e lma 0x10
Loading section .fini, size 0x8 lma 0xad7e
Loading section .rodata, size 0x372 lma 0xad88
Loading section .data, size 0x3d6 lma 0xb0fc
Loading section .ctors, size 0x4 lma 0xb4d2
Loading section .dtors, size 0x4 lma 0xb4d6
Loading section .eh_frame, size 0x1054 lma 0xff04
Start address 0x10
Transfer rate: 403792 bits in <1 sec.
```

To set a breakpoint, type:

```
break main
```

The following output displays:

```
Breakpoint 1 at 0x132: file hello.c, line 15.
```

3. To run the program, type:

```
run
```

For Windows, the following output displays:

```
Starting program: C:\hello.exe
Breakpoint 1, main () at hello.c:15
15      a = 3;
```

Similar output displays for UNIX systems with `hello.x` as the executable name.

The program stops at the breakpoint.

4. To print the value of variable `a`, type:

```
print a
```

The following output displays:

```
$1 = 0
```

- To execute the next command, type:

```
step
```

The following output displays:

```
16      b = 4;
```

- To display the value of a again, type:

```
print a
```

The following output displays:

```
$2 = 3
```

- To display the program being debugged, type:

```
list
```

The following output displays:

```
11      int main()
12      {
13          int b;
14
15          a = 3;
16          b = 4;
17          printf("Hello, world!\n");
18          foo(b);
19          return 0;
20      }
```

- To list a specific function code, use the list command with the name of the function to be display. For example, type:

```
list foo
```

The following output displays:

```
1      #include <stdio.h>
2
3      int a, c;
4
5      void foo(int b)
6      {
7          c = a + b;
8          printf("%d + %d = %d\n", a, b, c);
9      }
10
```

- To set a breakpoint at line seven, enter:

```
break 7
```

You can set a breakpoint at any line by entering `break linenumber`, where

*linenumber* is the specific line number to break.

The following output displays:

```
Breakpoint 2 at 0xf4: file hello.c, line 7.
```

10. To resume normal execution of the program until the next breakpoint, type:

```
continue
```

The following output displays:

```
Continuing.  
Hello, world!  
Breakpoint 2, foo (b=4) at hello.c:7  
7          c = a + b;
```

11. To step to the next instruction and execute it, type:

```
step
```

The following output displays:

```
8          printf("%d + %d = %d\n", a, b, c);
```

12. To display the value of *c*, type:

```
print c
```

The following output displays:

```
$3 = 7
```

13. To see how you got to where you are, type:

```
backtrace
```

The following output displays:

```
#0 foo (b=4) at hello.c:9  
#1 0x15c in main () at hello.c:18
```

14. To exit the program and quit the debugger, type:

```
quit
```

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

## Debug with Insight

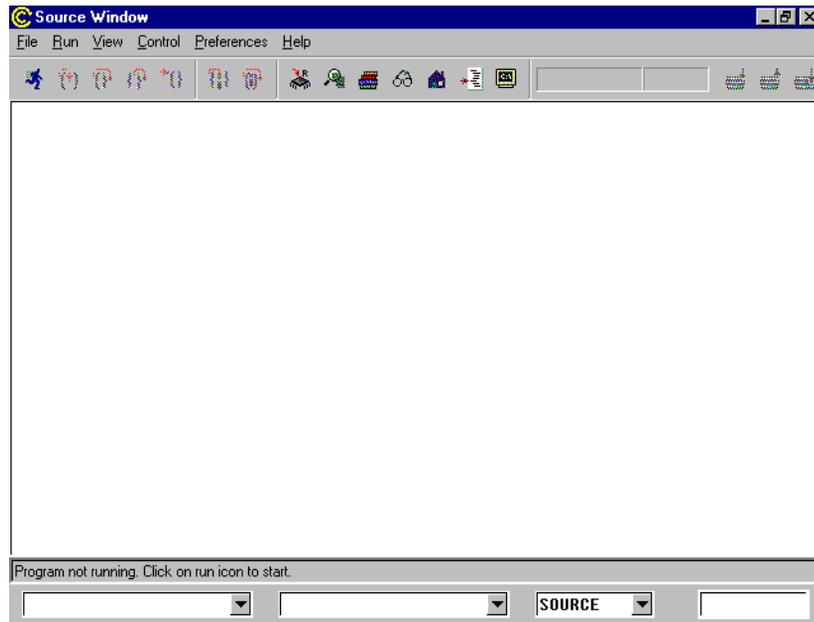
The following documentation serves as a general reference for debugging with GNUPro Toolkit's graphical user interface, Insight; for more information, see Insight's **Help** menu for discussion of general functionality and use of menus, buttons or other features; see also "Insight, GDB's Alternative Interface" and the "Examples of Debugging with Insight" documentation in *GNUPro Debugger Tools* (see <http://www.redhat.com/docs/manuals/gnupro/>).

**IMPORTANT!** Insight works as the default means for debugging; to disable the GUI, use the `gdb -nw` command for *non-windowing* command line work.

1. From a shell window, enter the following input:  
gdb

Insight launches, displaying the **Source Window**.

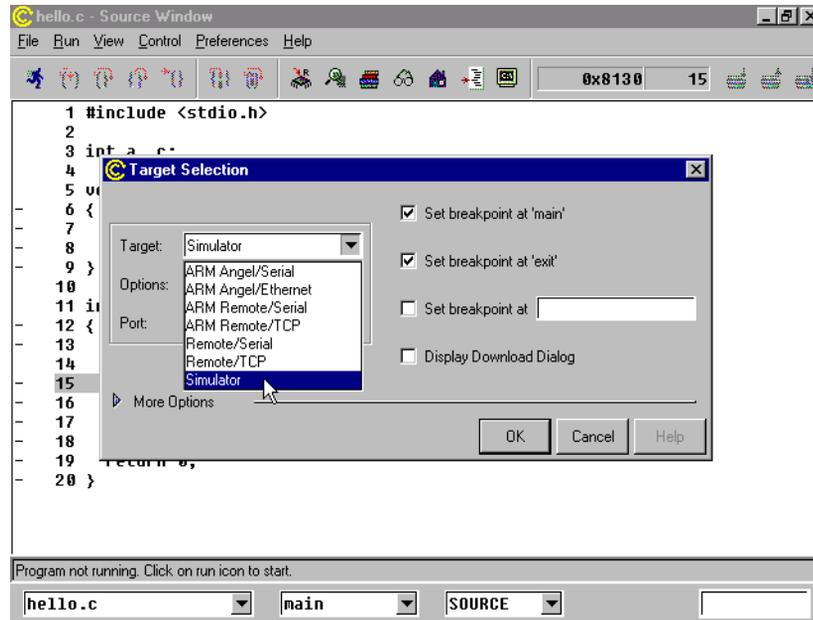
**Figure 2:** Source Window, the main window interface for Insight.



The menu selections in the **Source Window** are **File**, **Run**, **View**, **Control**, **Preferences**, and **Help**. To work with the other windows for debugging purposes specific to your project, use the **View** menu or the buttons in the toolbar.

2. To open a specific file as a project for debugging, select **File** → **Open** in the **Source Window**. Select hello.exe in the selection window. The file's contents will then pass to the GDB interpreter.
3. To connect to the target **Run** → **Connect to target** in the **Source Window**. Then in the Target drop down selection choose Simulator (as demonstrated in Figure 3)

**Figure 3:** Target Selection window



- To start debugging, click the **Run** button (Figure 4) from the **Source Window**.

**Figure 4: Run button**



When the debugger runs, the button turns into the **Stop** button (Figure 5).

**Figure 5: Stop button**



The **Stop** button interrupts the debugging process for a project, provided that the underlying hardware and protocols support such interruptions. Generally, machines that are connected to boards cannot interrupt programs on those boards. In such cases, a dialog box appears as a prompt asking if you want to abandon the session and if the debugger should detach from the target.

For an embedded project, click **Run**; then click the **Continue** button (Figure 6); this ensures configuration between the target and the host is clear so that the debugging tools will work effectively.

**Figure 6: Continue button**



**WARNING!** When debugging a target, do not click on the **Run** button during an active

debugging process, since using the **Run** button will effectively restart the session with all work unrecoverable.

For more information on Insight, see its **Help** menu. For examples of debugging session procedures for using Insight, see the following documentation (the content assumes familiarity with debugging procedures).

- “Selecting and Examining a Source File” (below)
- “Setting Breakpoints and Viewing Local Variables” on page 32

To specify how source code appears and to change debugging settings, from the **Preferences** menu, select **Source**.

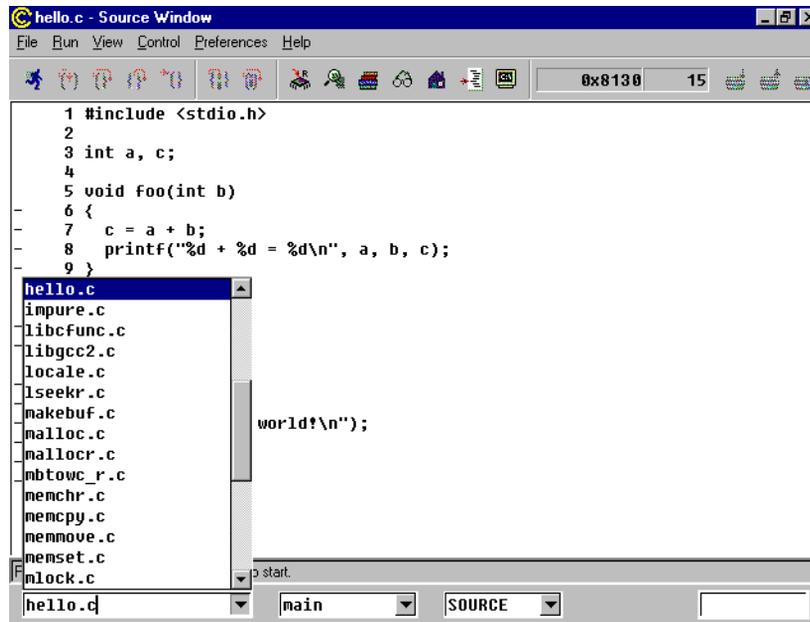
**IMPORTANT!** When debugging remote targets with RedBoot, the processor name and identification codes display when connecting to the target. To obtain the same information, from the **Source Window**, see the **Plugins** menu. To add identification codes to the debugger’s table of processors, see the ***GDB Internals*** documentation, distributed with the source code.

## Selecting and Examining a Source File

To select a source file, or to specify what to display when examining a source file when debugging, use the following processes.

1. Select a source file from the file drop-down list with the **Source Window** (`main.c` in Figure 7).

**Figure 7:** Source file selection



2. Select a function from the function drop-down list to the right of the file drop-down list, or type its name in the text field above the list to locate the function (in Figure 8, see the executable line 11, where the `main` function displays).

**Figure 8:** Search for functions

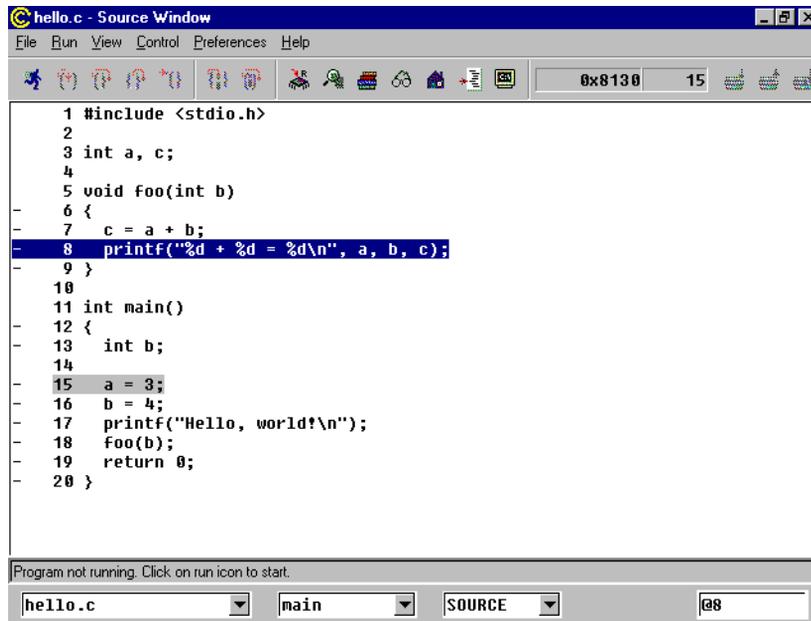
The screenshot shows the GDB Source Window for the file 'hello.c'. The window title is 'hello.c - Source Window'. The menu bar includes 'File', 'Run', 'View', 'Control', 'Preferences', and 'Help'. The toolbar contains various icons for navigation and execution. The source code is displayed in a text area with line numbers 1 through 20. Line 11, 'int main()', is highlighted in blue. The status bar at the bottom shows 'Program not running. Click on run icon to start.' and a search bar with 'hello.c' in the file dropdown, 'main' in the function dropdown, 'SOURCE' in the view dropdown, and 'main' in the search text box.

```

1 #include <stdio.h>
2
3 int a, c;
4
5 void foo(int b)
6 {
7     c = a + b;
8     printf("%d + %d = %d\n", a, b, c);
9 }
10
11 int main()
12 {
13     int b;
14
15     a = 3;
16     b = 4;
17     printf("Hello, world!\n");
18     foo(b);
19     return 0;
20 }

```

3. Use the **Enter** key to repeat a previous search. Use the **Shift** and **Enter** keys simultaneously to search backwards.
4. Type @ with a number in the search text box in the top right of the **Source Window**. Press **Enter**. Figure 9 shows a jump to line 86 in the `main.c` source file.

**Figure 9:** Searching for a specific line in source code

## Setting Breakpoints and Viewing Local Variables

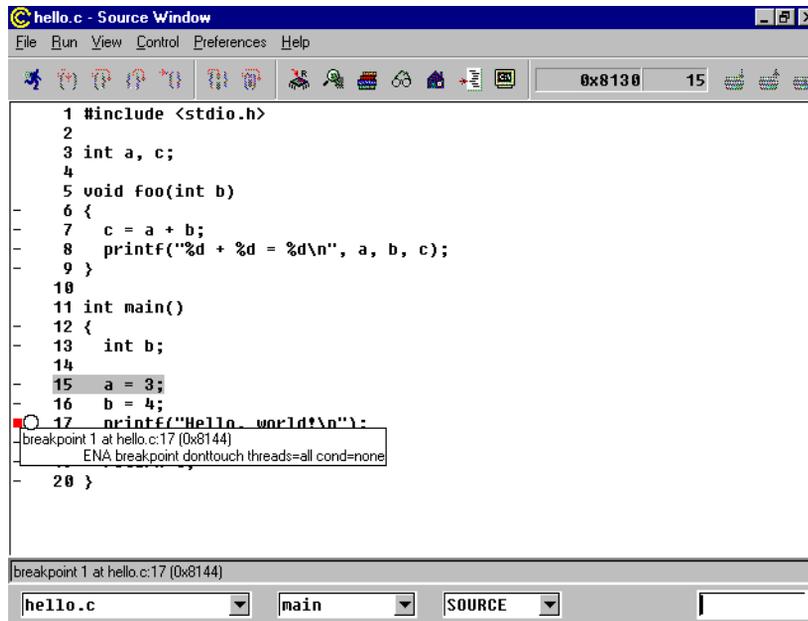
A breakpoint can be set at any executable line in a source file.

Executable lines are marked by a minus sign in the left margin of the **Source Window**. When the cursor is over a minus sign for an executable line, the cursor changes to a circle. When the cursor is in this state, a breakpoint can be set. The **Breakpoints** window is for managing the breakpoints: disabling them, enabling them, or erasing them; an enabled breakpoint is one for which the debugging session will stop, a disabled breakpoint is one which the debugging session ignores.

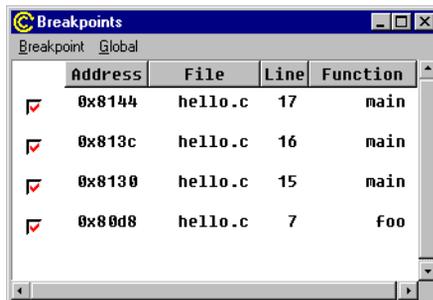
The following exercise steps you through setting four breakpoints in a function, as well as running the program and viewing changed values in local variables.

1. To set a breakpoint, have an active the `main.c` source file open in the **Source Window**, and, with the cursor over a minus sign on a line, click the left mouse button. When you click on the minus sign, a red square appears for the line, signifying a set breakpoint (see the highlighted line 105 in Figure 10 for a set breakpoint).

Clicking the line again will remove the breakpoint.

**Figure 10:** Results of setting breakpoint for line 17

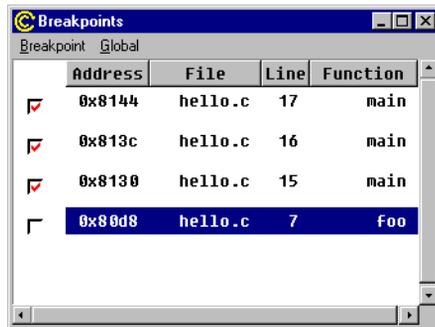
2. Open the **Breakpoints** window (Figure 11) using the **Breakpoints** button from the **Source Window**. See a line with a check box in the window appears showing that you set a breakpoint for a corresponding line in the **Source Window** frame. With the cursor over a breakpoint, a *breakpoint information balloon* displays in the **Source Window** (the information details the breakpoint, its address, its associated source file and line, its state, whether enabled, temporary, or erased, and the association to all threads for which the breakpoint will cause a stop).

**Figure 11:** Breakpoints window

3. The debugger ignores disabled breakpoints, lines indicated having a black square over them in the **Source Window** frame (see line 17 in Figure 10). Click on a breakpoint to disable the breakpoint. Figure 12 shows the results in the

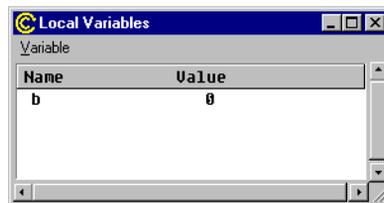
**Breakpoints** window of disabling a breakpoint. Re-enable a breakpoint at a line by clicking on the check box in the **Breakpoints** window. Once a breakpoint is enabled for a line, it will again have a red square in the **Source Window** frame.

**Figure 12:** Results of disabling a breakpoint at line 17

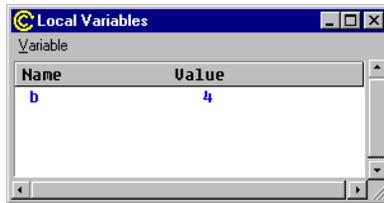


4. Repeat the process to set breakpoints at specific lines.
5. Click **Run** in the **Source Window** to start the executable. The debugger runs until it finds a breakpoint. When the target stops at a breakpoint, the debugger highlights a line, where the debugging stopped. For more information about breakpoints, see the standard documentation for Insight: “Insight, GDB’s Alternative Interface” and the “Examples of Debugging with Insight” documentation in *GNUPro Debugger Tools*; see <http://www.redhat.com/docs/manuals/gnupro/>.
6. Open the **Local Variables** window by clicking its button in the tool bar for the **Source Window**; the **Local Variables** window displays the values of the variables (see Figure 13 for the `b` variable in `main.c`).

**Figure 13:** Local Variables window



7. Click the **Continue** button in the **Source Window** tool bar to move to the next breakpoint. The variables that changed value turn color in the **Local Variables** window (see results in Figure 14 for the `b` variable in `hello.c`).

**Figure 14: Local Variables** window after setting breakpoints

8. Click the **Continue** button two more times to step through the next two breakpoints (until execution stops at line 17) and see the values of the local variables change (in Figure 14).

## Produce an Assembler Listing from Source Code

The following command produces an assembler listing:

```
arm-elf-gcc -g -O2 -Wa,-al -c hello.c
```

`-g` gives the assembler the necessary debugging information, `-O2` produces optimized code output, `-Wa` tells the compiler to pass the text immediately following the comma as a command line to the assembler, `-al` requests an assembler listing, and `-c` tells GCC to compile or assemble the source files without linking. The following example shows an excerpt similar to the output you will see.

**Example 5: Assembler listing**

```

66             .text
67             .align    2
68             .global   main
69             .type     main,function
70             main:
71             .LFB2:
72             .LM7:
73
74             @ args = 0, pretend = 0, frame = 0
75             @ frame_needed = 1,
                    current_function_anonymous_args = 0
76             .LBB2:
77 003c 0DC0A0E1    movip, sp
78             .LCFI3:
79 0040 00D82DE9    stmfd  sp!, {fp, ip, lr, pc}
80             .LCFI4:
81 0044 04B04CE2    sub   fp, ip, #4
82             .LCFI5:

```

```
83 0050 FFFFFFFEB      bl      __gccmain
84                      .LM8:
85
86 0054 0330A0E3      mov r3, #3
87 0058 18209FE5      ldr r2, .L6
88                      .LM9:
89
90 005c 18009FE5      ldr r0, .L6+4
91                      .LM10:
92
93 0060 003082E5      str r3, [r2, #0]
86                      .LM11:
87
89 0064 FFFFFFFEB      bl printf
```

## A Guide to Writing Linker Scripts

In the `/usr/releasename/host/target/lib/ldscripts/` path, find the example linker scripts (*host* signifies your host configuration and *target* signifies the embedded configuration to which you target). In that directory, there will be files with the `.x`, `.xnb`, `.xn`, `.xr`, `.xs`, and `.xu` extensions serving as examples of linker scripts.

The linker script accomplishes the following processes to result.

- Sets up the memory map for the application.  
When your application loads 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.
- Sets up the constructor and destructor tables for GNU C++ compiling.  
Actual section names vary depending on your object file format. For `a.out` and COFF formats, `.text`, `.data`, and `.bss` are the three main sections.
- Sets the default values for variables in use by `sbrk()` and the `crt0` file, which, typically, `_bss_start` and `_end` call.

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

- Having the linker create the memory map by using the `-Map` option.
- After linking, using the `nm` utility to check `start`, `bss_end` and `_etext` and other critical addresses.

The following discussion provides an example of setting up a linker script for a standard target.

1. Load the file so that it executes first with the `STARTUP(crt0.o)` command.

A configuration may, for instance, require using the COFF object file format, which, by default, does not link in `crt0.o` because it assumes that you have a `crt0` file. A `config` file controls this behavior in each architecture in a `STARTFILE_SPEC` macro. If you have `STARTFILE_SPEC` set to `NULL`, the GNU compiler formats its command line and does not add `crt0.o`. You can specify any filename 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 omit `STARTUP` and link in `crt0.o` in a makefile or by using different linker scripts (this option is useful with initializing floating point values or with adding device support).

- Using `GROUP`, load a file.

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

In this case, the file is a relocated library containing the definitions for the low-level functions that the `libc.a` file requires.

- Using `SEARCH_DIR`, specify the path in which to look for files.

```
SEARCH_DIR(.)
```

- Using `_DYNAMIC`, specify whether or not there are shared dynamic libraries. In the following example's case, a value of zero provides no shared libraries.

```
__DYNAMIC = 0;
```

- Set `_stack`, the variable for specifying RAM for the ROM monitor.

- Specify a name for a section to which you can refer 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 2MB. If the output file exceeds the upper limit, `MEMORY` produces an error message.

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

In this example's case, you set up the memory map of the board's stack for high memory.

- Next, set up constructor and destructor tables. 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 of the actual instructions reside in `.text` for setting up the constructor and destructor tables for the GNU C++ compiler. The section description redirects itself to the RAM variable that you previously set (see Step 5) with the `_stack` variable.

**8.** Set up the `.data` section.

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

A COFF file is where all of the initialized data goes. `CONSTRUCTORS` is a special command that the GNU linker, `ld`, uses.

**9.** Set up default values for `_bss_start` and `_end` variables by setting up the `.bss` section.

The default values for `_bss_start` and `_end` are for use by the `crt0` file when it zeros 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 ]
}
}
```

For more information on linking, see *Using ld* in *GNUPro Development Tools*.

## Rebuild Tools for Windows Systems

Red Hat may provide updates to this release in the form of source code patches. Once these patches are applied, you will need to rebuild the binaries before the update can be used. The following documentation details the instructions for rebuilding this release on a Windows system using the Red Hat Cygwin native compiler.

Use the following process for rebuilding:

1. Install the source code for this release and apply relevant patches. To install the source code, see the README file provided with your release. To apply the patches, follow the instructions provided with each patch that you receive.

**IMPORTANT!** In the following instructions, substitute the actual name that the following examples show (`arm-020110`) for the release name with which you rebuild (shown in the README file).

The amount of disk space required for rebuilding varies depending on the filesystem used. Red Hat recommends at least 1GB of free disk space for the source code, the build directory, and the new installation directory.

2. Install and set up the Cygwin native toolchain environment supported for this release. With the tools installed, setting up a Cygwin environment is part of the process of setting environment variables; see “Get the Tools to Work Properly” on page 3, if you have not already done those tasks.

**WARNING!** Do not use a Cygwin environment from another Cygwin release; doing so will cause problems rebuilding and subsequently using the tools.

3. Use the following steps for configuring, building and installing.

Do not continue to the next step unless the previous steps are successful.

If a step fails for any reason, please save a copy of the exact error message as a file (cut and paste, screen dump, etc.) along with any relevant log files and submit them in a bug report when reporting problems.

After using the steps, ensure the binaries are installed properly; see “Ensure

Completion of Rebuilding (Windows)” on page 41.

4. Start a bash shell with the `bash` command from a MS-DOS shell window). The following example shows what you will see in the window (typing `bash` after the `C:\>` prompt, you get a `bash-2.04$` prompt).

```
C:\>bash
bash-2.04$
```

5. From the `bash-2.04$` prompt, create a directory for building the tools and navigate to it with the following input (`build_dir` is the complete path and build directory name that you create in the following example).

```
mkdir build_dir
cd build_dir
```

6. Run the `configure` command from the build directory that you just created; Example 4 shows what you will see after completing the input in the build directory. `/cygdrive/c/redhat/arm-020110/` is an example for the source directory path and directory from which to execute the `configure` command.

**WARNING!** Never run the `configure` command in your source directory!

**WARNING!** Never rerun the `configure` command in your `build_dir` directory!

**Example 4:** Running the `configure` command from the build directory

```
bash-2.04$ /cygdrive/c/redhat/arm-020110/src/configure -v \
--prefix=/cygdrive/c/myredhat/arm-020110 \
--exec-prefix=/cygdrive/c/myredhat/arm-020110/H-i686-pc-cygwin \
--host=i686-pc-cygwin \
--target=arm-elf \
> configure.log 2>&1 &
```

The previous input is all one line before using the **Enter** key (the backslashes, `\`, signify line breaks for this text’s display requirements).

Watch the `configure` output using a `tail -f configure.log` command as the previous example shows; use the **Ctrl-C** key sequence to exit the `tail` process.

To save disk space, create three log files of the process in the `c:\build_dir` directory (`configure.log`, `build.log`, and `install.log`), so that you can have log files of the process when you later delete your build directory.

Use the `tail -f` command as you did with `configure.log` to view the content of files.

7. Use the `make` tool to build the binaries and `info` files.

```
make -w all info > make.log 2>&1 &
```
8. Use the `make` tool to install the binaries and `info` files.

```
make -w install install-info > install.log 2>&1 &
```
9. With the `cp` command, copy `cygwin1.dll` from the native tools directory,

`/cygdrive/c/redhat/arm-020110/H-i686-pc-cygwin/bin/`, to  
`/cygdrive/c/myredhat/arm-020110/H-i686-pc-cygwin/bin`, the new  
 installation). See Example 5.

**Example 5: Copying `cygwin1.dll` to a new installation**

```
cp
/cygdrive/c/redhat/arm-020110/H-i686-pc-cygwin/bin/cygwin1.dll \
/cygdrive/c/myredhat/arm-020110/H-i686-pc-cygwin/bin
```

The previous input is all one line before using the **Enter** key (the backslashes, `\`, signify line breaks for this text's display requirements).

## Ensure Completion of Rebuilding (Windows)

Test that the newly rebuilt tools work with the following instructions (which by no means show a comprehensive test).

1. Start a bash shell with the `bash` command from an MS-DOS shell window; typing `bash` after the `C:\>` prompt, you get a `bash-2.04$` prompt.
2. Add the new installation binaries to the `PATH` environment variable information as the following example shows.

```
export PATH=/cygdrive/c/myredhat/arm-020110/H-i686-pc-cygwin/bin:$PATH
```

3. Create a "**Hello world**" program with the following input:

```
cat > hello.c
extern int printf(__const char *format, ...);
int main () { printf("Hello World!\n"); }
```

Use the **Ctrl-D** key sequence to exit the process.

4. Compile the "**Hello world**" program.

```
arm-elf-gcc -Wall hello.c -o hello.exe
```

5. Execute the "**Hello world**" program.

```
arm-elf-run hello.exe
```

At the `bash-2.04$` prompt, see the following output.

```
Hello World!
```

Rebuilding is complete.

# Rebuild Tools for HP/UX, Linux, or Solaris Systems

Red Hat may provide updates to this release in the form of source code patches. Once these patches are applied, you will need to rebuild the binaries before the update can be used.

The following documentation details the instructions for rebuilding this release on a HP/UX, Linux, or Solaris system, using the Red Hat native compiler. Use the following process for rebuilding.

1. Install the source code for this release and apply relevant patches. To install the source code, see the README file provided with your release. To apply the patches, follow the instructions provided with each patch that you receive.

**IMPORTANT!** In the following instructions, substitute the actual name that the following examples show (`arm-020110`) for the release name with which you rebuild (shown in the README file).

The amount of disk space required for rebuilding varies depending on the filesystem used. Red Hat recommends at least 1GB of free disk space for the source code, the build directory, and the new installation directory.

2. Install and set up the native toolchain environment supported for this release. With the tools installed, setting up an environment is part of the process of setting environment variables; see “Get the Tools to Work Properly” on page 3, if you have not already done those tasks.

3. Use the following steps for configuring, building and installing. Do not continue to the next step unless the previous steps are successful. If a step fails for any reason, please save a copy of the exact error message as a file (cut and paste, screen dump, etc.) along with any relevant log files and submit them in a bug report when reporting problems.

After using the steps, ensure the binaries are installed properly; see “Ensure Completion of Rebuilding (Windows)” on page 41.

4. Start a bash shell with the `bash` command from a shell window). The following example shows what you will see in the window (typing `bash` after your standard prompt, you get a `bash-2.04$` prompt).

```
bash
bash-2.04$
```

5. From the `bash-2.04$` prompt, create a directory for building the tools and navigate to it with the following input (where `build_dir` is the build directory you create).

```
mkdir build_dir
cd build_dir
```

6. Run the `configure` command from the build directory that you just created; Example 4 shows what you will see after completing the input in the build directory. Replace `host` (where `host` signifies the toolchain’s triplet name) with `H-hppa1.1-hp-hpux10.20` for HP 10.20 or `H-hppa1.1-hp-hpux11.00` for 11.0 version, `H-i686-pc-linux-gnulibc2.1` for Red Hat Linux 7.0 or 7.1 versions,

and `H-sparc-sun-solaris2.5` for Sun Solaris versions.  
`/yourdrive/redhat/arm-020110/` is an example for the source directory from which to execute the `configure` command.

**WARNING!** Never run the `configure` command in your source directory!

**WARNING!** Never rerun the `configure` command in your `build_dir` directory!

**Example 6:** Running the `configure` command from the build directory

```
bash-2.04$ /yourdrive/redhat/arm-020110/src/configure -v \
--prefix=/yourdrive/myredhat/arm-020110 \
--exec-prefix=/yourdrive/myredhat/arm-020110/H-host \
--host=host \
--target=arm-elf \
> configure.log 2>&1 &
```

The previous input is all one line before using the **Enter** key (the backslashes, `\`, signify line breaks for this text's display requirements).

Watch the `configure` output using a `tail -f configure.log` command as the previous example shows; use the **Ctrl-C** key sequence to exit the `tail` process.

To save disk space, create three log files of the process in the `build_dir` directory (`configure.log`, `build.log`, and `install.log`), so that you can have log files of the process when you later delete your build directory. Use the `tail -f` command as you did with `configure.log` to view the content of files.

7. Use the `make` tool to build the binaries and `info` files.

```
make -w all info > make.log 2>&1 &
```

8. Use the `make` tool to install the binaries and `info` files.

```
make -w install install-info > install.log 2>&1 &
```

## Ensure Completion of Rebuilding (HP/UX, Linux, or Solaris)

Test that the newly rebuilt tools work with the following instructions (which by no means show a comprehensive test).

1. Start a bash shell with the `bash` command from a shell window; typing `bash` after the prompt, you get a `bash-2.04$` prompt.
2. Add the new installation binaries to the `PATH` environment variable information as the following example shows. Replace `host` (where `host` signifies the toolchain's triplet name) with `H-hppa1.1-hp-hpux10.20` for HP 10.20 or `H-hppa1.1-hp-hpux11.00` for 11.0 version, `H-i686-pc-linux-gnulibc2.1` for Red Hat Linux 7.0 or 7.1 versions, and `H-sparc-sun-solaris2.5` for Sun Solaris versions.

```
export PATH=/yourdrive/myredhat/arm-020110/H-host/bin:$PATH
```

3. Create a "**Hello world**" program with the following input:

```
cat > hello.c
extern int printf(__const char *format, ...);
int main () { printf("Hello World!\n"); }
```

Use the **Ctrl-D** key sequence to exit the process.

4. Compile the "**Hello world**" program.

```
arm-elf-gcc -Wall hello.c -o hello.x
```

5. Execute the "**Hello world**" program.

```
arm-elf-run hello.x
```

At the `bash-2.04$` prompt, see the following output.

```
Hello World!
```

Rebuilding is complete.

# 2

## Reference

---

The following documentation describes the specific features of the tools and the ABI requirements for ARM and ARM/Thumb processors.

- “Compiler Features” on page 46
- “ABI Summary of Features” on page 52
- “Assembler Features” on page 57
- “Linker Features” on page 61
- “Binary Utility Features” on page 61
- “Debugger Features” on page 63
- “Simulator Features” on page 63
- “Cygwin Features” on page 63

# Compiler Features

The following documentation describes ARM and ARM/Thumb features of the GNUPro compiler. For generic compiler information, see *Using GCC in GNUPro Compiler Tools*.

## Compiler Options

The following ARM and ARM/Thumb command line options are available.

`-mcpu=XXX`

Produces assembly code specifically for the indicated processor. For the `xxx` variable, substitute `arm` by default; other substitutions include: `arm2`, `arm250`, `arm3`, `arm6`, `arm60`, `arm600`, `arm610`, `arm620`, `arm7`, `arm7m`, `arm7d`, `arm7dm`, `arm7di`, `arm7dmi`, `arm70`, `arm700`, `arm700i`, `arm710`, `arm710c`, `arm7100`, `arm7500`, `arm7500fe`, `arm7tdmi`, `arm8`, `strongarm`, `strongarm110`, `strongarm1100`, `arm8`, `arm810`, `arm9`, `arm9e`, `arm920`, `arm920t`, `arm940t`, `arm9tdmi`.

**IMPORTANT!** If `-mcpu` is not specified, the default is to generate code for the `strongarm2`.

`-mtune=XXX`

Like `-mcpu`, except that, instead of specifying the actual target processor type, and hence restricting which instructions can be used, it specifies that GCC should tune the performance of the code as if the target were of the type specified, yet still choosing the instructions that it will generate based on the processor. For some ARM implementations, better performance can be obtained by using this option.

`-march=XXX`

Produce assembly code specifically for an ARM processor of the indicated architecture. The `xxx` variable can be one of the following architectures: `armv2`, `armv2a`, `armv3`, `armv3m`, `armv4`, `armv4t`, `armv5`, `armv5t`, and `armv5te`.

**IMPORTANT!** If `-march` is not specified, the default is to generate code for the `armv5`.

`-mapcs-frame`

`-mno-apcs-frame`

`-mapcs-frame` generates a stack frame upon entry to a function, as defined in the ARM® Procedure Calling Standard (APCS). `-mno-apcs-frame` does not generate a stack frame upon entry to a function; the APCS specifies generating stack frames, which produces slightly smaller and faster code. `-mno-apcs-frame` is the default setting. Specifying `-fomit-frame-pointer` with `-mapcs-frame` will cause stack frames not to be generated for leaf functions.

`-mapcs`

Synonymous with `-mapcs-frame`.

`-mapcs-26`

`-mapcs-32`

`-mapcs-26` produces code for a processor running with a 26-bit program counter or generates assembly code that conforms to the 26-bit version of the APCS, as used by earlier versions of the ARM processor (ARM2 and ARM3). `-mapcs-26` replaces `-m2` and `-m3` options from previous versions for the compiler.

`-mapcs-32` produces code for a processor running with a 32-bit program counter or generates assembly code that conforms to the 32-bit version of the APCS, as used by earlier versions of the ARM processor (ARM2 and ARM3). `-mapcs-26` replaces `-m6` option from previous versions for the compiler. `-mapcs-32` is the default setting.

`-mapcs-stack-check`

`-mno-apcs-stack-check`

`-mapcs-stack-check` produces assembly code that checks the amount of stack space available upon entry to a function, calling a suitable function if insufficient stack space is available. `-mno-apcs-stack-check` does not produce code to check for stack space upon entry to a function; this is the default setting.

`-mapcs-reentrant`

`-mno-apcs-reentrant`

`-mapcs-reentrant` produces assembly code that is position independent and reentrant. `-mno-apcs-reentrant` does not produce position independent, reentrant assembly code; this is the default setting.

`-mlittle-endian`

`-mbig-endian`

`-mlittle-endian` produces assembly code targeted for a little-endian processor; this is the default setting. `-mbig-endian` produces assembly code targeted for a big-endian processor.

`-mwords-little-endian`

Produces assembly code which is targeted for a big-endian processor, but which stores words in a little-endian word order (byte order of the 32107654 form); this is for backward compatibility with older versions of GCC. Use only if you require compatibility with code for big-endian ARM processors generated by the compilers prior to version 2.8.

`-mfpe=N`

`-mfp=N`

With `-mfpe=`, floating-point instructions should be emulated by the ARM Floating Point Emulator code version, *N*, valid version numbers being 2 and 2; 2 is the default setting for the `-mfpe=` option. `-mfp=` is synonymous with `-mfpe=`, for compatibility with earlier versions of GCC.

`-mlong-calls`

`-mno-long-calls`

`-mlong-calls` tells the compiler to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register; this switch is needed if the target function will lie outside of the 64 megabyte addressing range of the offset based version of subroutine call instruction. Even if this switch is enabled, not all function calls will be turned into long calls. The heuristic is that static functions, functions which have the `short-call` attribute, functions that are inside the scope of a `#pragma no_long_calls` directive, and functions whose definitions have already been compiled within the current compilation unit, will not be turned into long calls. The exception to this rule is that weak function definitions, functions with the `long-call` attribute or the `section` attribute, and functions that are within the scope of a `#pragma long_calls` directive, will always be turned into long calls. The `-mlong-calls` feature is not enabled by default. Specifying `-mno-long-calls` will restore the default behaviour, as will placing the function calls within the scope of a `#pragma long_calls_off` directive.

These switches have no effect on how the compiler generates code to handle function calls with function pointers.

`-malignment-traps`

`-mno-alignment-traps`

Use `-malignment-traps` to generate code that will not trap if the MMU (memory management unit) has alignment traps enabled. On ARM architectures prior to ARM version 4, there were no instructions to access half-word objects stored in memory; however, when reading from memory, a feature of the ARM architecture allows a word load to be used, even if the address is aligned, and the processor core will rotate the data as it is being loaded. `-malignment-traps` tells the compiler that such misaligned accesses will cause a MMU trap and that it should instead synthesize the access as a series of byte access; the compiler can still use word accesses to load half-word data if it knows the address is aligned to a word boundary. `-malignment-traps` is ignored when compiling for ARM version 4 or later, since these processors have instructions to access half-word objects directly in memory.

`-mno-alignment-traps` generates code that assumes that the MMU will not trap unaligned accesses; this produces better code when the target instruction set does not have half-word memory operations (for example, implementations prior to ARM version 4). You cannot use `-mno-alignment-traps` to access unaligned word objects, since the processor will only fetch one 32-bit aligned object from memory. The default setting is `-mno-alignment-traps`, since this produces better code when there are no half-word memory instructions available.

`-mshort-load-bytes`

`-mno-short-load-words`

Deprecated aliases for `-malignment-traps` option.

`-mshort-load-words`

`-mno-short-load-bytes`

Deprecated aliases for `-mno-alignment-traps` option.

`-mapcs-float`

`-mno-apcs-float`

With `-mapcs-float`, pass floating point arguments using the float point registers; this is one of the variants of the APCS. `-mapcs-float` is recommended if the target hardware has a floating point unit or if a lot of floating point arithmetic is going to be performed by the code. `-mno-apcs-float` is the default setting, since integer only code is slightly increased in size if you use `-mno-apcs-float`.

`-msched-prolog`

`-mno-sched-prolog`

`-mno-sched-prolog` prevents the reordering of instructions in the function prolog, or the merging of those instruction with the instructions in the function's body, meaning that all functions will start with a recognizable set of instructions (or one of a choice from a small set of different function prologues); this information can be used to locate the start if functions inside an executable piece of code. `-msched-prolog` is the default setting, which allows such reordering for instructions.

`-mhard-float`

`-msoft-float`

With `-mhard-float`, floating-point instructions are performed in hardware; this option does not apply to code generated for the ARM/Thumb microarchitecture. With `-msoft-float`, floating point instructions should be emulated by library calls; this is the default setting.

`-mnop-fun-dllimport`

Disable support for the `dllimport` attribute.

`-mpoke-function-name`

`-mpoke-function-name` causes the compiler to store the name of each function it compiles as an ASCII string in the assembler output, previous to the function, and follows it with a readily identifiable number, as the following example shows:

```
t0
    .ascii "arm_poke_function_name", 0
    .align
t1
    .word 0xff000000 + (t1 - t0)
arm_poke_function_name
    mov     ip, sp
    stmfd  sp!, {fp, ip, lr, pc}
```

```
sub    fp, ip, #4
```

When performing a stack backtrace, code can inspect the value of `pc` stored at `fp + 0`; if the trace function then looks at the `pc - 12` location and the top 8 bits are set, then you know that there is a function name embedded immediately preceding this location and has `((pc[-3]) & 0xff000000)` length.

`-mabort-on-noreturn`

`-mno-abort-on-noreturn`

`-mabort-on-noreturn` causes the compiler to generate a call to the function `abort()` at the end of a function which has the `noreturn` attribute;

`-mabort-on-noreturn` is disabled by default, since there is no guarantee that the host operating system will provide an `abort()` function call.

`-mno-abort-on-noreturn` is the setting for.

`-msched-prolog`

`-mno-sched-prolog`

`-msched-prolog` allows instructions in function prologues to be rearranged to improve performance; this is the default setting. `-mno-sched-prolog` does not allow the instructions in function prologues to be rearranged, guaranteeing that function prologues will have a well-defined form.

`-mthumb`

Generates Thumb instructions rather than ARM instructions.

`-mtpcs-frame`

`-mno-tpcs-frame`

`-mtpcs-frame` generates stack backtrace frames for non-leaf functions, if `-mthumb` has been specified. `-mno-tpcs-frame` is the default setting.

`-mtpcs-leaf-frame`

`-mtpcs-leaf-frame` generates stack backtrace frames for leaf functions, if `-mthumb` has been specified. `-mno-tpcs-leaf-frame` is the default setting.

`-mcallee-super-interworking`

`-mno-callee-super-interworking`

`-mcallee-super-interworking` assumes that non-static functions might be called in ARM mode, if `-mthumb` has been specified.

`-mcaller-super-interworking`

`-mno-caller-super-interworking`

`-mcaller-super-interworking` assumes that function pointers might point at non-interworking aware code, if `-mthumb` has been specified.

`-mthumb-interwork`

`-mno-thumb-interwork`

`-mthumb-interwork` produces assembly code which supports calls between the ARM instruction set and the Thumb instruction set. `-mno-thumb-interwork` does not produce code specifically intended to support calling between ARM and Thumb instruction sets; this is the default setting.

--target-default-spec  
 --no-target-default-spec  
 --target-default-spec changes gcc's default behavior to use the *excalibur* specs. --no-target-default-spec restores the old behavior. This is necessary in order to be able to compile programs that will run in the simulator.

## Preprocessor Symbols

Table 6 shows the compiler preprocessor symbols.

**Table 6:** Preprocessor symbols

<i>Symbol</i>	<i>Condition</i>
arm	Always defined
__semi__	Always defined
__APCS_32__	If -mapcs-26 has <i>not</i> been specified
__APCS_26__	If -mapcs-2 has been specified
__SOFTFP__	If -mhard-float has <i>not</i> been specified
__ARMWEL__	If -mwords-little-endian has been specified
__ARMEB__	If -mbig-endian has been specified
__ARMEL__	If -mbig-endian has <i>not</i> been specified
__ARM_ARCH_2__	If -mcpu=arm2 or -mcpu=arm205 or -mcpu=arm3 or -march=armv2 has been specified
__ARM_ARCH_3__	If -mcpu=arm6 or -mcpu=arm600 or -mcpu=arm610 or -mcpu=arm7 or -mcpu=arm700 or -mcpu=arm710 or -mcpu=arm7100 or -mcpu=arm7500 or -mcpu=arm7500fe or -march=armv3 has been specified
__ARM_ARCH_3M__	If -mcpu=arm7m or -mcpu=arm7dm or -mcpu=arm7dmi or -march=armv3m has been specified
__ARM_ARCH_4__	If -mcpu=arm8 or -mcpu=arm810 or -mcpu=arm920 or -mcpu=strongarm or -mcpu=strongarm110 or -mcpu=strongarm1100 or -march=armv4 has been specified
__ARM_ARCH_4T__	If -mcpu=arm7tdmi or -mcpu=arm9 or -mcpu=arm920t or -mcpu=arm9tdmi or -march=armv4t has been specified

## Attributes

For a complete description of attributes, see “Declaring Attributes of Functions” and “Specifying Attributes of Variables” in “Extensions to the C Language Family” in *Using GCC in GNUPro Compiler Tools*.

There is one specific attribute, `long_call`, which can only be applied to function prototypes; it specifies that calls to this function must be done indirectly, as it may lie

outside of the 26-bit addressing range of normal function calls.

## Pragmas

There are two pragmas, `#pragma long_calls` and `#pragma no_long_calls`.

`#pragma long_calls` indicates that all function call instructions generated by the compiler from this point on in the code should be indirect function calls (in other words, the address of the function to be called is first loaded into a register and then the call is made via that register), which allows function calls to functions that lie outside of the normal 26-bit addressable range of a function call instruction.

`#pragma no_long_calls` turns off the effect of the `#pragma long_calls`, after which the compiler generates the normal ARM function call instruction.

## ABI Summary of Features

For ARM, ARM/Thumb, and StrongARM processors, the tools adhere by default to the APCS. The following ABI summary for the GNUPro tools is consistent with this standard.

### Data Types

Table 7 shows the size and alignment for all data types.

**Table 7:** Data type sizes and alignment

<i>Type</i>	<i>Size (bytes)</i>	<i>Alignment (bytes)</i>
char	1 byte	1 byte
short	2 bytes	2 bytes
int	4 bytes	4 bytes
unsigned	4 bytes	4 bytes
long	4 bytes	4 bytes
long long	8 bytes	8 bytes
float	4 bytes	4 bytes
double	8 bytes	8 bytes
pointer	4 bytes	4 bytes

Alignment within aggregates (structures and unions) is as in Table 7, with padding added if necessary.

Aggregates have alignment equal to that of their most aligned member.

Aggregates have sizes which are a multiple of their alignment.

### Subroutine Calls and Registers

The following documentation describes the calling conventions for subroutine calls.

The general purpose registers,  $r0$  through  $r3$ , are for passing parameters.

Table 8 outlines other register usage.

**Table 8:** Register usage

<i>Register usage</i>	<i>Register</i>
Volatile	$r0$ through $r3$ , $r12$
Non-volatile	$r4$ through $r10$
Frame pointer	$r11$
Stack pointer	$r13$
Return address	$r14$
Program counter	$r15$

Structures that are less than or equal to 32 bits in length are passed as values.

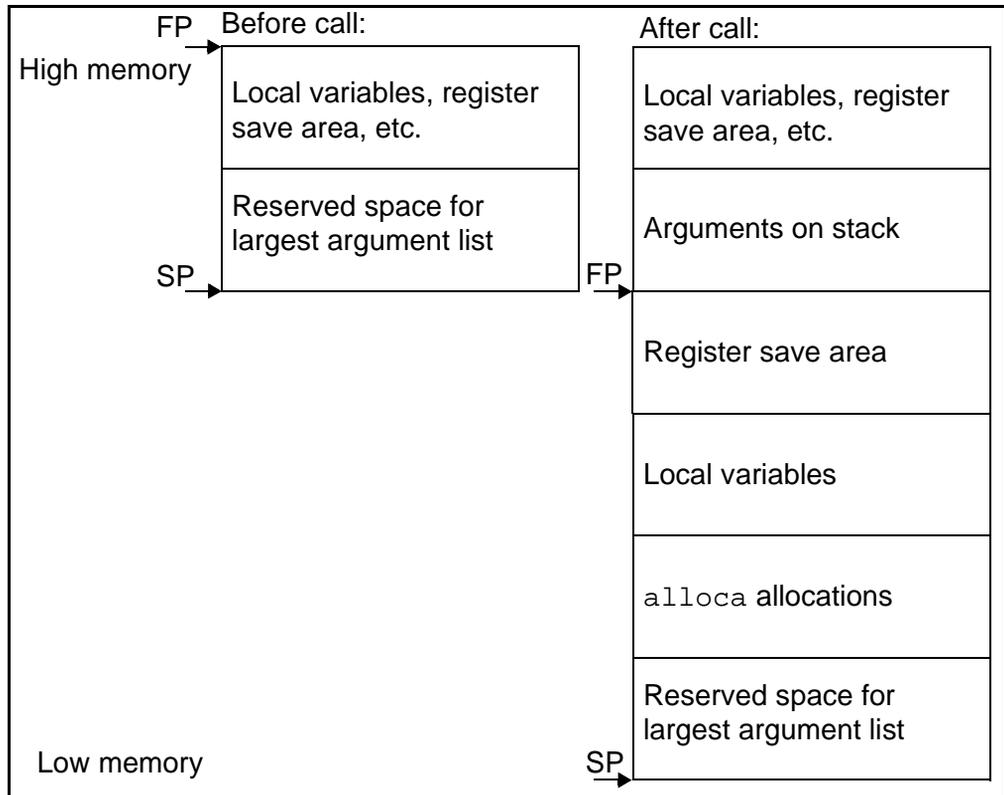
Structures that are greater than 32 bits in length are passed as pointers.

## Stack Frames

The following documentation describes the structure of stack frames for ARM, ARM/Thumb, and StrongARM processors:

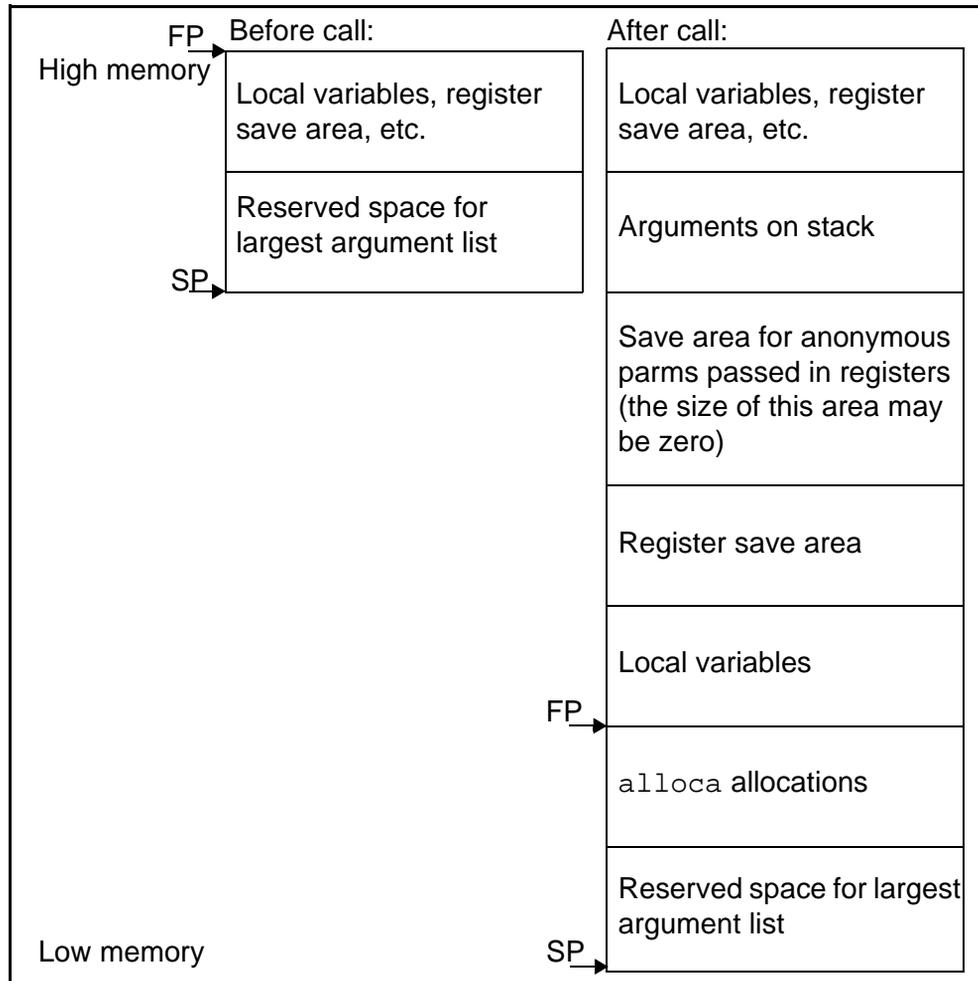
- The stack grows downwards from high addresses to low addresses.
- A leaf function does not need to allocate a stack frame if one is not needed.
- A frame pointer (FP) need not be allocated.
- The stack pointer (SP) is always aligned to four-byte boundaries.
- The stack pointer always points to the lowest addressed word currently stored on the stack.

See Figure 15 for stack frames for functions that take a fixed number of arguments.

**Figure 15:** Stack frames for functions that take a fixed number of arguments

See Figure 16 for stack frames for functions that take a variable number of arguments.

**Figure 16:** Stack frames for functions that take a variable number of arguments



## Compliances

A floating point value occupies one or two words as appropriate to its type. Floating point values are encoded in IEEE 754 format, with the most significant word of a double having the lowest address.

**IMPORTANT!** When targeting little-endian ARM processors, the words that make up a double will be stored in big-endian order, while the bytes inside each word will be stored in little-endian order.

The C compiler extends arguments of type `float` to type `double` to support working between ANSI C and classic C.

`char`, `short`, pointer, and other integral values occupy one word in an argument list. `char` and `short` values are extended to 32 bits by the C compiler when put into the argument list.

A structure always occupies an integral number of words (unless this is overridden by the `-mstructure-size-boundary` command line option). Argument values are collated in the order written in the source program.

The first four words of the argument values are loaded into registers `r0` through `r3`, and the remainder are pushed on to the stack in reverse order (so that arguments later in the argument list have higher addresses than those earlier in the argument list). As a consequence, a larger than word sized value can be passed in integer registers, or even split between an integer register and the stack.

The selection of register(s) to hold a function's result is slightly more complicated. A float or integer-like value is returned in register `r0`. Doubles and long longs are returned in registers `r0` and `r1`. For doubles the most significant word is always held in `r0`. For long longs this only happens if the `-mbig-endian` switch has been used.

All other results are returned by placing them into a suitably sized area of memory provided for this purpose by the function's caller. A pointer to this area of memory is passed to the function as a hidden first argument, generated at compile time, as Example 9 shows.

**Example 9:** Values returned by placing them into a sized area of memory

```
LargeType t;  
t = func (arg);
```

is implemented by the compiler as:

```
LargeType t;  
(void) func (& t, arg);
```

A type is integer-like if its size is less than or equal to one word. If the type is a struct, union, or array, then all of its addressable sub-fields must have an offset of zero (see the following examples).

**Example 10:** Types that are integer-like (struct)

```
struct {int a:8, b:8, c:8, d:8;}
```

**Example 11:** Types that are integer-like (union)

```
union {int i; char * p;}
```

**Example 12:** Types that are not integer-like (struct)

```
struct {char A; char B; char c; char D;}
```

Unlike Example 10 or Example 11, Example 12 shows a type that is not integer-like,

because it is possible to take the address of fields B, C or D, and their offsets from the start of the structure are not 0.

## Assembler Features

The following documentation describes features of the GNUPro assembler for the ARM, ARM/Thumb, and ARM/Thumb processors. For generic assembler information, see “Command Line Options” and other content in *Using as* in *GNUPro Auxiliary Development Tools*. Syntax is based on the syntax in the *ARM Architecture Reference Manual* documentation.

### Assembler Options for the Compiler

The following options are for assembler functionality when invoking the compiler.

`-EB`

Assembles code for a big-endian processor.

`-EL`

Assembles code for a little-endian processor; this is the default option.

`--gdwarf2`

Selects DWARF2 debugging output.

`--gstabs`

Selects STABS debugging output; to debug assembler source code, you must specify one of the previous options when assembling the code or, by default, the assembler will not generate any debugging output.

`-mall`

Allows any instruction.

`-mapcs-26`

Marks the code as supporting the 26-bit variant of the APCS.

`-mapcs-32`

Marks the code as supporting the 32-bit variant of the APCS. This is the default.

`-mapcs-reentrant`

Ensures code is position independent, in other words, reentrant.

`-m[arm][1|2|250|3|6|7|8|9][t][d][m][i]`

`-mstrongarm[110][0]`

Selects processor variant.

`-m[arm]v[2|2a|3|3m|4|4t]`

Selects architecture variant.

`-mfpa10`

Selects the v1.0 floating point architecture.

- mfpall  
Selects the v1.1 floating point architecture.
- mfpe-old  
Does not allow floating point multiple instructions.
- mno-fpu  
Does not allow any floating point instructions.
- mthumb  
Allows only ARM/Thumb instructions.
- mthumb-interwork  
Marks the assembled code as supporting interworking.

## Assembler Syntax

Assembler comments start with the at symbol (@) and extend to the end of the line. Multiple assembler statements can appear on the same line providing that they are separated by the semicolon symbol (;).

## Registers

Example 13 shows the format that registers use for ARM, ARM/Thumb, and ARM/Thumb processors.

**Example 13:** Register usage format

```
{register_name, register_number}
```

The following general registers for ARM, ARM/Thumb, and ARM/Thumb processors are available.

- {r0, 0}
- {r1, 1}
- {r2, 2}
- {r3, 3}
- {r4, 4}
- {r5, 5}
- {r6, 6}
- {r7, 7}
- {r8, 8}
- {r9, 9}
- {r10, 10}
- {r11, 11}
- {r12, 12}
- {r13, 13}
- {r14, 14}
- {r15, 15}

The accumulator for the ARM, ARM/Thumb, and ARM/Thumb processors has the following specification: {acc0, 0}.

For the APCS specification, the general registers have the following names.

- {a1, 0}
- {a2, 1}
- {a3, 2}
- {a4, 3},
- {v1, 4}
- {v2, 5}
- {v3, 6}
- {v4, 7},

- {v5, 8}
- {sb, 9}
- {s1, 10}
- {ip, 12}
- {lr, 14}
- {v6, 9}
- {v7, 10},
- {fp, 11}
- {sp, 13},
- {pc, 15}

The floating point registers for the ARM and ARM/Thumb processors have the following specification.

- {f0,16}
- {f4,20}
- {f1, 17}
- {f5, 21}
- {f2, 18}
- {f6, 22}
- {f3, 19}
- {f7, 23}

Both the assembler and the compiler support hardware floating point.

For detailed information on the ARM/Thumb machine instruction set, see the *ARM Architecture Reference Manual* and Intel's *ARM/Thumb Reference Manual*. The GNU assembler implements all the opcodes, including the standard ARM and ARM/Thumb opcodes and ARM/Thumb extensions.

## Synthetic Instructions

The assembler supports the following synthetic or *synthesized* instructions (*pseudo* instructions, which correspond to two or more actual machine instructions).

`.arm`

Subsequent code to this directive uses the ARM instruction set.

`.thumb`

Subsequent code to this directive uses the ARM/Thumb instruction set.

`.thumb_func`

Subsequent code to this directive labels the name of a function, which has been encoded using ARM/Thumb instructions, rather than ARM instructions.

`.code 16`

An alias directive for `.thumb`.

`.code 32`

An alias directive for `.arm`.

`.force_thumb`

Subsequent code to this directive uses the ARM/Thumb instruction set, and should be assembled even if the target processor does not support ARM/Thumb instructions.

`ldr register, = expression`

Loads the value of *expression* into *register* (if the value is one that can be constructed by a `mov` or `mvn` instruction, then this directive will be used; otherwise, the value will be placed into the nearest literal pool, if it is not there already, and a PC relative `ldr` instruction will be used to load the value into the register).

`.ltorg`

Dumps the current accumulated literal pool entries into the current section; this directive does *not* generate any jump instructions around the pool.

`.pool`

Synonymous directive for `.ltorg`.

`.req`

Creates an alias directive for a register name, as Example 14 and Example 15 show.

**Example 14:** `.req` usage

```
alias .req register_name
```

**Example 15:** Specific `.req` usage

```
overflow .req r1
```

Once the alias has been created, it can be used in the assembler sources at any place where a register name would be expected.

`.align`

`.align` pads the location counter to a particular storage boundary (in a current subsection, *ABS-EXPR* in the following example); with Example 16, see `.align`'s usage.

**Example 16:** `.align` usage

```
.align ABS-EXPR, ABS-EXPR, ABS-EXPR
```

The first expression, *ABS-EXPR* (which must be absolute), is the alignment required, expressed as the number of low-order zero bits the location counter must have after advancement (for example, `.align 3` advances the location counter until it is a multiple of eight; if the location counter is already a multiple of eight, no change is needed); the second expression, *ABS-EXPR* (also absolute), gives the fill value to be stored in the padding bytes (it and the comma may be omitted; if it is omitted, the padding bytes are zero); the third expression, *ABS-EXPR* (also absolute and also optional), if present, is the maximum number of bytes that should be skipped by this alignment directive.

If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment. There is one special case. If the first expression evaluates to zero it is treated as if it were two. This is for compatibility with ARM's own assembler, which uses `.align 0` to mean align to a word boundary.

## Assembler Error Messages

The following error messages can display when using the GNU assembler.

- **Error: Unrecognized opcode**  
For a misspelled instruction or for where there is a syntax error somewhere.
- **Warning: operand out of range**  
For when an immediate value was specified that is too large for the instruction

## Linker Features

The following documentation describes features of the GNUPro linker. There are no specific linker options; for generic linker information, see “Linker scripts” in *Using ld* in *GNUPro Development Tools*. The GNU linker uses a linker script to determine how to process each section in an object file, and how to lay out the executable. The linker script is a declarative program consisting of a number of directives. For instance, the `ENTRY()` directive specifies the symbol in the executable that will be the executable’s entry point. To see a linker script, use the `arm-elf-ld --verbose` command. For a complete description of the linker script, see “Linker scripts” in *Using ld* in *GNUPro Development Tools*.

## Binary Utility Features

The following documentation describes the specific features of the GNU binary utilities, specifically the GNUPro binary utility, `objdump`, for ARM, ARM/Thumb, and ARM/Thumb processors, for which a command line call has been added, using `--disassembler-options` (long version) or `-M` (short version), each of which takes an argument that can be any arbitrary piece of text, with this text passed on to the code specific to the target object file being dumped for when there is a requirement for fine tuning the dumping for that target. In the case of the ARM/Thumb, the target specific code will look to see if one of the register sets in Table 17 is provided, the corresponding register names will be used when displaying a disassembly.

**Table 17:** Register settings and their names

<i>Register set</i>	<i>Register names</i>
raw	r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15
std	r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, sp, lr, pc
apcs	a1, a2, a3, a4, v1, v2, v3, v4, v5, v6, s1, fp, ip, sp, lr, pc
atpcs	a1, a2, a3, a4, v1, v2, v3, v4, v5, v6, v7, v8, IP, SP, LR, PC
special-atpcs	a1, a2, a3, a4, v1, v2, v3, WR, v5, SB, SL, FP, IP, SP, LR, PC

The `std` set is the default register name set. Consider the assembler source code in Example 18.

**Example 18:** Using a register name set

```
add r1, r2, r3
```

The same code in Example 18 could be disassembled with the `apcs` register set

specified, as Example 19 shows.

**Example 19:** Using the APCS register name set (`apcs`)

```
objdump -d --disassembler-options=reg-names-apcs
```

Example 19 could produce output in Example 20.

**Example 20:** Output when using `--disassembler-options=reg-names-apcs`

```
00000000 <.text>:
0:   e0821003      add    a2, a3, a4
```

If the same assembler object file is disassembled without specifying a register set like Example 21, the output in Example 22 will be produced.

**Example 21:** Call to `objdump` without using a register name set

```
objdump -d
```

**Example 22:** Output from `objdump` without using a register name set

```
00000000 <.text>:
0:   e0821003      add    r1, r2, r3
```

If the code is disassembled with the `atpcs` register set specified, as with the call in Example 23, the output in Example 24 will be produced.

**Example 23:** Using the APCS register name set (`atpcs`)

```
objdump -d --disassembler-options=reg-names-atpcs
```

**Example 24:** Output when using `--disassembler-options=reg-names-atpcs`

```
00000000 <.text>:
0:   e0821003      add    a2, a3, a4
```

**IMPORTANT!** When using the `-M` command, there are some similarities.

When using the `-M` (short version) of the `objdump` command, the syntax for Example 25 is appropriate.

**Example 25:** Using `-M` with `objdump`

```
objdump -d -M reg-names-atpcs
```

There is an argument to the `--disassembler-options` or `-M` command line switches for the `objdump` command for ARM/Thumb, `force-thumb` (see Example 26 for usage).

**Example 26:** ARM/Thumb `objdump` call with

```
--disassembler-options=force-thumb
objdump -d --disassembler-options=force-thumb
```

The `force-thumb` switch tells the disassembler to treat the contents of the file it is disassembling as if they were Thumb instructions, even if it thinks that they are ARM instructions. Normally, the disassembler will rely upon detecting special flags in the file it is disassembling in order to tell whether it is an ARM binary or an ARM/Thumb binary. However, some compilers do not put these flags into their output files.

---

# Debugger Features

The following documentation describes ARM and ARM/Thumb features of the GNUPro debugger. GDB's built-in software simulation of the ARM or ARM/Thumb processors allows the debugging of programs compiled for them without requiring any access to actual hardware. To activate the simulator mode in GDB, use the `target sim` command, and then load the code; see “Debug with the Built-in Simulator” on page 23 for instructions. For information on Insight, the debugger graphical user interface, see “Debug with Insight” on page 26. There are no specific debugger command line options for ARM or ARM/Thumb processors. For generic debugger information, see *Debugging with GDB* in *GNUPro Debugger Tools*.

**IMPORTANT!** If the Arm chip switches modes during a debugging session the results from GDB will be indeterminate.

## Simulator Features

The simulator can simulate any ARM or ARM/Thumb processor, and any ARM or ARM/Thumb instructions. It is not a cycle accurate simulator, nor is it a board level simulator. It does not simulate any hardware outside of the CPU; for example, it does not simulate an MMU or any co-processor. It does have limited pass through capability to the host operating system; for example, it is able to simulate basic file operations (including writing to `stdout`) and memory allocation. The simulator is theoretically capable of simulating any address space, providing that memory is available on the host operating system.

There are no specific simulator command line options for ARM or ARM/Thumb processors.

## Cygwin Features

Cygwin, a full-featured Win32 porting layer for UNIX applications, is compatible with all Win32 hosts (currently, these are Microsoft Windows NT/95/98 systems). With Cygwin, you can make all directories have similar behavior, with all the UNIX default tools in their familiar place. Shells include `bash`, `ash`, and `tcsh`. Tools such as Perl, Tcl/Tk, `sed`, `awk`, `vim`, Emacs, `xemacs`, `telnetd` and `ftpd` are also available.

In order to emulate a UNIX kernel to access all processes that can run with it, use the Cygwin DLL (dynamically linked library). The Cygwin DLL will create shared memory areas so that other processes using separate instances of the DLL can access

the kernel. Using DLLs with Cygwin, link into your program at run time instead of build time.

## Defining Windows Resources for Cygwin

`windres` reads a Windows resource file (`*.rc`) and converts it to a `.res` COFF file. The syntax and semantics of the input file are the same as for any other resource compiler; see any publication describing the Windows resource format for details.

`windres` compiles a `.res` file to include all the bitmaps, icons, and other resources you need, into one object file. Omitting the `-O coff` declaration would create a Windows `.res` format file without linkable COFF objects. Instead, `windres` produces a COFF object, for compatibility with how a linker can handle Windows resource files directly, maintaining the `.res` naming convention.

For more information on `windres`, see *Using binutils* in *GNUPro Auxiliary Development Tools*.

## Building and Using DLLs with Cygwin

The following documentation provides an example of how to build a `.dll` file, using a single file, `myprog.c`, for the program, `myprog.exe`, and a single file, `mydll.c`, for the contents of the `.dll` file, `mydll.dll`, then compiling everything as objects.

```
gcc -shared myprog.c -o mydll.dll -e __mydll_init@12
```

Now, when you build your program, you link against the import library, with declaration's like the following example's commands.

```
gcc myprog.o mydll.dll -o myprog.exe
```

## Using GCC with Cygwin

The following documentation discusses using the GNUPro compiler with Cygwin to compile like with UNIX. Use the following command in a shell console.

```
gcc hello.c -o hello.exe
hello.exe
Hello, World
```

Cygwin allows you to build programs with full access to the standard Windows 32-bit API, including the GUI functions (as defined in Microsoft publications); however, the process of building those applications is slightly different using the GNU tools instead of the Microsoft tools. Your sources won't need to change; just remove all `__export` attributes from functions and replace them, as the following example shows.

```
int foo (int) __attribute__ ((__dllexport__));

int
foo (int i)
```

For most cases, remove the `__export` attributes. For convenience, include the

following code fragment; otherwise, you'll have to add a `-e _mainCRTStartup` declaration to your link line in your Makefile.

```
#ifdef __CYGWIN__
WinMainCRTStartup() { mainCRTStartup(); }
#endif
```

The Makefile is similar to any other UNIX-like or Cygwin Makefile. The only difference is that you use a `gcc -mwindows` declaration to link your program (*myapp.exe* in the following example's script) into a GUI application instead of into a command line application.

```
myapp.exe : myapp.o myapp.res
            gcc -mwindows myapp.o myapp.res -o $@

myapp.res : myapp.rc resource.h
            windres $< -O coff -o $@
```

## Debugging Cygwin Programs

Before you can debug your program, you need to prepare your program for debugging. Add a `-g` declaration to all the other flags you use when compiling your sources to objects, in order to add extra information to the objects (making them much bigger), and to provide critical information to the debugger regarding line numbers, variable names, and other useful things; these extra symbols and debugging data give your program enough information about the original sources so that the debugger can resolve the problems. Use declarations like the following example's commands.

```
gcc -g -O2 -c myapp.c
gcc -g myapp.c -o myapp
```

To debug, use the `gdb myapp.exe` declaration (substituting the executable file's name for *myapp*). The copyright text displays followed by the (`gdb`) prompt, waiting for you to enter commands like `run` or `help`.

If your program stops and you want to determine where it crashed, type `run` and let your program run. After it crashes, use the `where` command to determine where it crashed, or a `info locals` call to see the values of all the local variables. There is also the `print` declaration that lets you examine individual variables or lines to which pointers point. If your program is doing something unexpected, use the `break` command to stop your program when it gets to a specific function or line number.

Using the `run` command, debugging continues until stopping your program at a breakpoint; use other commands to look at the state of your program at that point, to modify variables, or to step through your program's statements one at a time. Use the `help` command to get a list of all the commands to use, or see *Debugging with GDB* in *GNUPro Debugger Tools*.



# Index

---

`#define` 8  
`#include` files 8  
`#pragma long_call` 52  
`%PATH`, for environment variables 3  
`.align` 60  
`.bss` 36, 38  
`.data` 38  
`.text` 36  
`.text` section 37  
`;` (semicolon), for assembler comments 58

## Symbols

@, searching for line numbers, when debugging 31  
@, semicolon, assembler comments 58  
\_\_\_, for preprocessor symbols 51  
`_bss_start` 36, 38  
`_DYNAMIC`, for shared dynamic libraries 37  
`_end` 36, 38

## Numerics

26-bit address 51, 52  
26-bit version 47, 57  
32-bit address 56, 57  
754, IEEE format, for floating point values 55

## A

`a.out` 17  
address 11  
    function calls beyond 48  
    memory (virtual, load) 11  
aggregates 52  
alignment of data types 52  
`alloca` allocations 54  
allocatable sections 11  
ANSI C runtime library 5  
APCS (ARM Procedure Calling Standard) 46, 47, 52,

57, 58, 62

Application Binary Interface (ABI) summary 52

`ar` 6

architecture variant, floating point 57

archive index 6

arguments, on stack 54, 55

assembler 2, 3, 6, 7, 22, 57–61, 62

`.align` 60

    code, generating 47

    code-specific processor 46

    comments 58

    debugging 35

    error messages 60

    listing 35

    little-endian 47, 57

    opcodes 59

    registers 58

    synthetic instructions *see also* synthesized instructions

ATPCS (ARM/Thumb Procedure Calling Standard) 62

attributes 51

## B

big-endian 47, 57

binary 11

binary utilities 2, 6, 17, 39, 41, 61

blocks 11

breakpoint 15, 32–33

`buffer.h` 13

build process 40, 43

## C

C

    compiler 5

    library 16

    math subroutine library 5

    preprocessor 5, 8

C++  
   class library 5  
   constructors 7  
   iostreams library 5  
 c++filt 6  
 case sensitivity 3  
 char values 56  
 COFF 2, 22  
 command line input, overriding structure limitations 56  
 command.h 13  
 compatibility 55  
 compiler 2, 5, 7, 22, 46–52, 56  
   assembler 35, 57  
   case sensitivity 3  
   command line options 46–51  
   conditional use 8  
   Cygwin development 18, 64  
   debugger development 14  
   leaf functions 50  
   linking 35  
   optimization 23  
   preprocessor 5  
   rebuilding tools (HP/UX, Linux, or Solaris) 41  
   rebuilding tools (Windows) 39  
   Windows, working with  
 compliancy 55  
 conditional compilation 8  
 configuring 1, 12–14, 40, 42, 43  
 constructor and destructor tables 36, 37  
 CONSTRUCTORS 38  
 contacting Red Hat ii  
 cpp 7  
 CREATE\_OBJECT\_SYMBOLS 37  
 crt0 (C RunTime 0) file 37  
 Cygwin 18, 63–65  
   .dll files, building example 64  
   compiler, working with 18, 64  
   debugger development 65  
   DLLs 18, 64  
   dlltool 18  
   GCC *see* compiler:Cygwin development  
   global symbols 18  
   Makefile 65  
   Windows resource file 64  
   windres 64

## D

-d, for assembler 17  
 data section 11  
 data type 52  
 debugger 2, 5, 23–??, 63  
   assembler 35  
   breakpoints 33, 35  
   Cygwin development 65  
   DWARF2 output 57  
   embedded projects, working with 28  
   GUI 5  
   information, getting 14

  Insight 26–??  
   jumps 31  
   lines, searching code for 31  
   local variables 32  
   STABS 57  
 deciphering utility 6  
 defs.h 13  
 destructor tables 36  
 diff, diff3, sdiff 5  
 directives 59  
   -disassemble 17  
   --disassembler-options 61  
 dlltool 18  
 documentation 1, 4, 29  
 DWARF2 57  
 dynamic libraries 37

## E

edit 13  
 ELF 1, 2, 22  
 embedded development, defined 28  
 endian processor (little and big) 47  
 environment variables, setting 3, 39, 41, 42, 44  
 exception handling 15  
 executable 11, 17

## F

file names 3  
 floating-point architecture selection 57  
 FPE (Floating-Point Emulator) 47  
 frame pointer (FP) 53

## G

GAS *see also* assembler  
 GCC *see also* compiler  
   gcov, for testing performance 5  
 GDB *see also* debugger  
   generating conforming code 62  
 GLD *see also* linker  
 global variables 12  
 GROUP, for loading 37

## H

hardware floating-point 59  
 header files 8  
 hosts supported 1  
 HP/UX 1, 21  
   environment variables, setting 3  
   rebuilding 41–44

## I

identifier 8  
 including files 8  
 info files 40, 43  
 input section 11  
 Insight 26–??  
 installation 1, 3, 22, 39, 42  
 instructions, synthesized, pseudo, or machine 3, 59

**L**

labels 3  
 LD *see also* linker  
 ld, the GNU linker *see also* linker  
 leaf function 50, 53  
 libc 5, 16  
 libg++ 5  
 libgcc.a 7  
 libio 5  
 libm 5, 16  
 libraries 3, 5, 7  
 line control 8  
 linker 2, 3, 5, 22, 36, 61  
 linker scripts 11, 61  
 Linux 1, 21
 

- environment variables, setting 3
- rebuilding 41–44

 little-endian assembly code, generating 47, 57  
 LMA (load memory address) 11  
 loadable 11  
 local variables 35, 54, 55  
 log files of the build process 40, 43  
 long\_call 51

**M**

-M 61  
 m68k-coff configuration 37  
 macro expansion 8  
 main() 7  
 main.c 14  
 make, for reconfiguring 5, 13  
 Makefile 12  
 math library 16  
 MEMORY 37  
 memory 36  
 memory management 48  
 MMU (memory management unit) 48

**N**

newlib 16  
 nm 6, 36  
 -nostlib 7

**O**

objcopy 6, 17  
 objdump 6, 12, 17, 61–62  
 object code archives 6  
 object file 6, 10, 12
 

- C library, linking 7
- format 1, 11
- information 6
- symbol tables 6

 opcodes 59  
 operating systems 1  
 optimization 23, 35  
 output section 11

**P**

patch 5  
 patches 39, 41  
 pointers 53, 54, 55  
 porting layer for UNIX applications 18, 63  
 pragmas 52
 

- prefix-addresses 17

 preprocessor 8, 51  
 problems ii  
 problems, reporting 39, 42  
 processor variant selection 57  
 PROM (Programmable Read-Only Memory) 17  
 pseudo instructions 59

**R**

RAM 38  
 RAM space 37  
 ranlib 6  
 rebuilding
 

- HP/UX, Linux, or Solaris 41–44
- Windows 39–41

 recompiling 12  
 Red Hat, contacting ii  
 RedBoot 15  
 register 3, 53, 54, 61
 

- floating point 59
- format usage 58
- general 58

 relinking 14  
 relocation 11  
 ROM monitor 15, 17  
 rule 13

**S**

sbrk() 36  
 SEARCH\_DIR, for specifying paths 37  
 sections 11
 

- .data 38
- .text 37

 main 36  
 names 3, 36  
 sizes 6  
 semicolon symbol (;) 58  
 short values 56  
 simulator 2, 22, 23, 24, 63
 

- compiling with 22
- debugging with 23

 size 6  
 sizes and alignment of data types 52  
 Solaris 1, 21
 

- environment variables, setting 3
- rebuilding 41–44

 source code patches 39, 41  
 source line control 8  
 STABS debugging output 57  
 stack frame 46, 53, 54, 55  
 stack pointer (SP) 53  
 stack space 47

- STARTFILE\_SPEC 37
- static variable 12
- stdout 8
- strip 6
- structures 52, 56
- sub-routine calls (stubs) 15, 52
- support 1
- symbol 3, 6, 11, 51
  - names 6
  - table 11
- synthesized instructions 59
- system settings 3
- systems 1

## T

- testing 5
- Thumb
  - ARM 50
  - endian targets (little and big) 47
  - instruction generation 50
  - MMU (memory management unit) 48
  - non-static functions 50
  - restricting instructions to 58
  - stack backtrace frames 50
  - synthetic instructions *see* synthesized instructions
- tool names 2
- total sizes 6
- triplet 1, 2
- tutorials 21–44

## U

- unions 52
- UNIX applications, porting to Windows 18, 63
- UNIX toolchains 3, 21
- utilities 6

## V

- variables, default values 36
- variables, environment, setting 3, 39, 41, 42, 44
- variables, local 35, 54, 55
- VMA (virtual memory address) 11

## W

- warnings 26, 28, 39, 40, 43, 60
- Web support site ii
- Windows 1, 21
  - Cygwin 18, 63
  - environment variables, setting 3, 41, 44
  - rebuilding 39–41