Sourcery G++ Lite

ARM EABI

Sourcery G++ Lite 2008q3-66

Getting Started

CodeSourcery

Sourcery G++ Lite: ARM EABI: Sourcery G++ Lite 2008q3-66: Getting Started

CodeSourcery, Inc.
Copyright © 2005, 2006, 2007, 2008 CodeSourcery, Inc.

Abstract

This guide explains how to install and build applications with Sourcery G++ Lite, CodeSourcery's customized, validated, and supported version of the GNU Toolchain. Sourcery G++ Lite includes everything you need for application development, including C and C++ compilers, assemblers, linkers, and libraries.

When you have finished reading this guide, you will know how to use Sourcery G++ from the command line.

Table of Contents

Preface

This preface introduces *Getting Started With Sourcery G++ Lite*. It explains the structure of this guide and lists other sources of information that relate to Sourcery G++ Lite.

1. Intended Audience

This guide is written for people who will install and/or use Sourcery G++ Lite. This guide provides a step-by-step guide to installing Sourcery G++ Lite and to building simple applications. Parts of this document assume that you have some familiarity with using the command-line interface.

2. Organization

This document is organized into the following chapters and appendices:

Chapter 1, *Sourcery G++ Lite Licenses*	This chapter provides information about the software licenses that apply to Sourcery G++ Lite. Read this chapter to understand your legal rights and obligations as a user of Sourcery G++ Lite.
Chapter 2, *Sourcery G++ Subscriptions*	This chapter provides information about Sourcery G++ subscriptions. CodeSourcery customers with Sourcery G++ subscriptions receive comprehensive support for Sourcery G++. Read this chapter to find out how to obtain and use a Sourcery G++ subscription.
Chapter 3, *Sourcery G++ Lite for ARM EABI*	This chapter provides information about this release of Sourcery G++ Lite including any special installation instructions, recent improvements, or other similar information. You should read this chapter before building applications with Sourcery G++ Lite.
Chapter 4, *Installation and Configuration*	This chapter describes how to download, install and configure Sourcery G++ Lite. This section describes the available installation options and explains how to set up your environment so that you can build applications.
Chapter 5, *Using Sourcery G++ from the Command Line*	This chapter explains how to build applications with Sourcery G++ Lite using the command line. In the process of reading this chapter, you will build a simple application that you can use as a model for your own programs.
Chapter 6, *CS3™: The CodeSourcery Common Startup Code Sequence*	CS3 is CodeSourcery's low-level board support library. This chapter describes the organization of the system startup code and tells you how you can customize it, such as by defining your own interrupt handlers. This chapter also documents the boards supported by Sourcery G++ Lite and the compiler and linker options you need to use with them.
Chapter 7, *Sourcery G++ Debug Sprite*	This chapter describes the use of the Sourcery G++ Debug Sprite for remote debugging. The Sprite allows you to debug programs running on a bare board without an operating system. This chapter includes information about the debugging devices and boards supported by the Sprite for ARM EABI.
Chapter 8, *Next Steps with Sourcery G++*	This chapter describes where you can find additional documentation and information about using Sourcery G++ Lite and its components.

3. Typographical Conventions

The following typographical conventions are used in this guide:

> `> command arg ...`   A command, typed by the user, and its output. The ">" character is the command prompt.

command   The name of a program, when used in a sentence, rather than in literal input or output.

`literal`   Text provided to or received from a computer program.

placeholder   Text that should be replaced with an appropriate value when typing a command.

`\`   At the end of a line in command or program examples, indicates that a long line of literal input or output continues onto the next line in the document.

Chapter 1
Sourcery G++ Lite Licenses

Sourcery G++ Lite contains software provided under a variety of licenses. Some components are "free" or "open source" software, while other components are proprietary. This chapter explains what licenses apply to your use of Sourcery G++ Lite. You should read this chapter to understand your legal rights and obligations as a user of Sourcery G++ Lite.

1.1. Licenses for Sourcery G++ Lite Components

The table below lists the major components of Sourcery G++ Lite for ARM EABI and the license terms which apply to each of these components.

Some free or open-source components provide documentation or other files under terms different from those shown below. For definitive information about the license that applies to each component, consult the source package corresponding to this release of Sourcery G++ Lite. Sourcery G++ Lite may contain free or open-source components not included in the list below; for a definitive list, consult the source package corresponding to this release of Sourcery G++ Lite.

Component	License
GNU Binary Utilities	GNU General Public License 3.0 [1]
GNU Compiler Collection	GNU General Public License 3.0 [2]
GNU Debugger	GNU General Public License 3.0 [3]
Sourcery G++ Debug Sprite for ARM	CodeSourcery License
Newlib C Library	Newlib License [4]
GNU Make	GNU General Public License 2.0 [5]
GNU Core Utilities	GNU General Public License 2.0 [6]

The CodeSourcery License is available in Section 1.2, "Sourcery G++™ Software License Agreement".

Important

Although some of the licenses that apply to Sourcery G++ Lite are "free software" or "open source software" licenses, none of these licenses impose any obligation on you to reveal the source code of applications you build with Sourcery G++ Lite. You can develop proprietary applications and libraries with Sourcery G++ Lite.

1.2. Sourcery G++™ Software License Agreement

1. **Parties.** The parties to this Agreement are you, the licensee ("You" or "Licensee") and CodeSourcery. If You are not acting on behalf of Yourself as an individual, then "You" means Your company or organization.

2. **The Software.** The Software licensed under this Agreement consists of computer programs and documentation referred to as Sourcery G++™ Lite Edition (the "Software").

3. **Definitions.**

---

[1] http://www.gnu.org/licenses/gpl.html
[2] http://www.gnu.org/licenses/gpl.html
[3] http://www.gnu.org/licenses/gpl.html
[4] http://sources.redhat.com/newlib/COPYING.NEWLIB
[5] http://www.gnu.org/licenses/old-licenses/gpl-2.0.html
[6] http://www.gnu.org/licenses/old-licenses/gpl-2.0.html

3.1. **CodeSourcery Proprietary Components.**   The components of the Software that are owned and/or licensed by CodeSourcery and are not subject to a "free software" or "open source" license, such as the GNU Public License. The CodeSourcery Proprietary Components of the Software include, without limitation, the Sourcery G++ Installer, any Sourcery G++ Eclipse plug-ins, and any Sourcery G++ Debug Sprite. For a complete list, refer to the *Getting Started Guide* included with the distribution.

3.2. **Open Source Software Components.**   The components of the Software that are subject to a "free software" or "open source" license, such as the GNU Public License.

3.3. **Proprietary Rights.**   All rights in and to copyrights, rights to register copyrights, trade secrets, inventions, patents, patent rights, trademarks, trademark rights, confidential and proprietary information protected under contract or otherwise under law, and other similar rights or interests in intellectual or industrial property.

4. **License Grant to Proprietary Components of the Software.**   You are granted a non-exclusive, royalty-free license to install and use the CodeSourcery Proprietary Components of the Software, transmit the CodeSourcery Proprietary Components over an internal computer network, and/or copy the CodeSourcery Proprietary Components for Your internal use only.

5. **Restrictions.**   You may not: (i) copy or permit others to use the CodeSourcery Proprietary Components of the Software, except as expressly provided above; (ii) distribute the CodeSourcery Proprietary Components of the Software to any third party; or (iii) reverse engineer, decompile, or disassemble the CodeSourcery Proprietary Components of the Software, except to the extent this restriction is expressly prohibited by applicable law.

6. **"Free Software" or "Open Source" License to Certain Components of the Software.**
This Agreement does not limit Your rights under, or grant You rights that supersede, the license terms of any Open Source Software Component delivered to You by CodeSourcery. Sourcery G++ includes components provided under various different licenses. The *Getting Started Guide* provides an overview of which license applies to different components. Definitive licensing information for each "free software" or "open source" component is available in the relevant source file.

7. **CodeSourcery Trademarks.**   Notwithstanding any provision in a "free software" or "open source" license agreement applicable to a component of the Software that permits You to distribute such component to a third party in source or binary form, You may not use any CodeSourcery trademark, whether registered or unregistered, including without limitation, CodeSourcery™, Sourcery G++™, the CodeSourcery crystal ball logo, or the Sourcery G++ splash screen, or any confusingly similar mark, in connection with such distribution, and You may not recompile the Open Source Software Components with the `--with-pkgversion` or `--with-bugurl` configuration options that embed CodeSourcery trademarks in the resulting binary.

8. **Term and Termination.**   This Agreement shall remain in effect unless terminated pursuant to this provision. CodeSourcery may terminate this Agreement upon seven (7) days written notice of a material breach of this Agreement if such breach is not cured; provided that the unauthorized use, copying, or distribution of the CodeSourcery Proprietary Components of the Software will be deemed a material breach that cannot be cured.

9. **Transfers.**   You may not transfer any rights under this Agreement without the prior written consent of CodeSourcery, which consent shall not be unreasonably withheld. A condition to any transfer or assignment shall be that the recipient agrees to the terms of this Agreement. Any attempted transfer or assignment in violation of this provision shall be null and void.

10. **Ownership.** CodeSourcery owns and/or has licensed the CodeSourcery Proprietary Components of the Software and all intellectual property rights embodied therein, including copyrights and valuable trade secrets embodied in its design and coding methodology. The CodeSourcery Proprietary Components of the Software are protected by United States copyright laws and international treaty provisions. CodeSourcery also owns all rights, title and interest in and with respect to its trade names, domain names, trade dress, logos, trademarks, service marks, and other similar rights or interests in intellectual property. This Agreement provides You only a limited use license, and no ownership of any intellectual property.

11. **Warranty Disclaimer; Limitation of Liability.** CODESOURCERY AND ITS LICENSORS PROVIDE THE SOFTWARE "AS-IS" AND PROVIDED WITH ALL FAULTS. CODESOURCERY DOES NOT MAKE ANY WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. CODESOURCERY SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, SYSTEM INTEGRATION, AND DATA ACCURACY. THERE IS NO WARRANTY OR GUARANTEE THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED, ERROR-FREE, OR VIRUS-FREE, OR THAT THE SOFTWARE WILL MEET ANY PARTICULAR CRITERIA OF PERFORMANCE, QUALITY, ACCURACY, PURPOSE, OR NEED. YOU ASSUME THE ENTIRE RISK OF SELECTION, INSTALLATION, AND USE OF THE SOFTWARE. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS AGREEMENT. NO USE OF THE SOFTWARE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

12. **Local Law.** If implied warranties may not be disclaimed under applicable law, then ANY IMPLIED WARRANTIES ARE LIMITED IN DURATION TO THE PERIOD REQUIRED BY APPLICABLE LAW.

13. **Limitation of Liability.** INDEPENDENT OF THE FORGOING PROVISIONS, IN NO EVENT AND UNDER NO LEGAL THEORY, INCLUDING WITHOUT LIMITATION, TORT, CONTRACT, OR STRICT PRODUCTS LIABILITY, SHALL CODESOURCERY BE LIABLE TO YOU OR ANY OTHER PERSON FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND, INCLUDING WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER MALFUNCTION, OR ANY OTHER KIND OF COMMERCIAL DAMAGE, EVEN IF CODESOURCERY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY TO THE EXTENT PROHIBITED BY APPLICABLE LAW. IN NO EVENT SHALL CODESOURCERY'S LIABILITY FOR ACTUAL DAMAGES FOR ANY CAUSE WHATSOEVER, AND REGARDLESS OF THE FORM OF ACTION, EXCEED THE AMOUNT PAID BY YOU IN FEES UNDER THIS AGREEMENT DURING THE PREVIOUS ONE YEAR PERIOD.

14. **Export Controls.** You agree to comply with all export laws and restrictions and regulations of the United States or foreign agencies or authorities, and not to export or re-export the Software or any direct product thereof in violation of any such restrictions, laws or regulations, or without all necessary approvals. As applicable, each party shall obtain and bear all expenses relating to any necessary licenses and/or exemptions with respect to its own export of the Software from the U.S. Neither the Software nor the underlying information or technology may be electronically transmitted or otherwise exported or re-exported (i) into Cuba, Iran, Iraq, Libya, North Korea, Sudan, Syria or any other country subject to U.S. trade sanctions covering the Software, to individuals or entities controlled by such countries, or to nationals or residents of such countries other than nationals who are lawfully admitted permanent residents of countries not subject to such sanctions; or (ii) to anyone on the U.S. Treasury Department's list of Specially Designated Nationals and Blocked Persons or the U.S. Commerce Department's Table of Denial Orders.

By downloading or using the Software, Licensee agrees to the foregoing and represents and warrants that it complies with these conditions.

15. **U.S. Government End-Users.** The Software is a "commercial item," as that term is defined in 48 C.F.R. 2.101 (Oct. 1995), consisting of "commercial computer software" and "commercial computer software documentation," as such terms are used in 48 C.F.R. 12.212 (Sept. 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (June 1995), all U.S. Government End Users acquire the Software with only those rights set forth herein.

16. **Licensee Outside The U.S.** If You are located outside the U.S., then the following provisions shall apply: (i) Les parties aux presentes confirment leur volonte que cette convention de meme que tous les documents y compris tout avis qui siy rattache, soient rediges en langue anglaise (translation: "The parties confirm that this Agreement and all related documentation is and will be in the English language."); and (ii) You are responsible for complying with any local laws in your jurisdiction which might impact your right to import, export or use the Software, and You represent that You have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

17. **Severability.** If any provision of this Agreement is declared invalid or unenforceable, such provision shall be deemed modified to the extent necessary and possible to render it valid and enforceable. In any event, the unenforceability or invalidity of any provision shall not affect any other provision of this Agreement, and this Agreement shall continue in full force and effect, and be construed and enforced, as if such provision had not been included, or had been modified as above provided, as the case may be.

18. **Arbitration.** Except for actions to protect intellectual property rights and to enforce an arbitrator's decision hereunder, all disputes, controversies, or claims arising out of or relating to this Agreement or a breach thereof shall be submitted to and finally resolved by arbitration under the rules of the American Arbitration Association ("AAA") then in effect. There shall be one arbitrator, and such arbitrator shall be chosen by mutual agreement of the parties in accordance with AAA rules. The arbitration shall take place in Granite Bay, California, and may be conducted by telephone or online. The arbitrator shall apply the laws of the State of California, USA to all issues in dispute. The controversy or claim shall be arbitrated on an individual basis, and shall not be consolidated in any arbitration with any claim or controversy of any other party. The findings of the arbitrator shall be final and binding on the parties, and may be entered in any court of competent jurisdiction for enforcement. Enforcements of any award or judgment shall be governed by the United Nations Convention on the Recognition and Enforcement of Foreign Arbitral Awards. Should either party file an action contrary to this provision, the other party may recover attorney's fees and costs up to $1000.00.

19. **Jurisdiction And Venue.** The courts of Placer County in the State of California, USA and the nearest U.S. District Court shall be the exclusive jurisdiction and venue for all legal proceedings that are not arbitrated under this Agreement.

20. **Independent Contractors.** The relationship of the parties is that of independent contractor, and nothing herein shall be construed to create a partnership, joint venture, franchise, employment, or agency relationship between the parties. Licensee shall have no authority to enter into agreements of any kind on behalf of CodeSourcery and shall not have the power or authority to bind or obligate CodeSourcery in any manner to any third party.

21. **Force Majeure.** Neither CodeSourcery nor Licensee shall be liable for damages for any delay or failure of delivery arising out of causes beyond their reasonable control and without their fault or negligence, including, but not limited to, Acts of God, acts of civil or military authority, fires, riots, wars, embargoes, or communications failure.

22. **Miscellaneous.** This Agreement constitutes the entire understanding of the parties with respect to the subject matter of this Agreement and merges all prior communications, representations, and agreements. This Agreement may be modified only by a written agreement signed by the parties. If any provision of this Agreement is held to be unenforceable for any reason, such provision shall be reformed only to the extent necessary to make it enforceable. This Agreement shall be construed under the laws of the State of California, USA, excluding rules regarding conflicts of law. The application of the United Nations Convention of Contracts for the International Sale of Goods is expressly excluded. This license is written in English, and English is its controlling language.

Chapter 2
Sourcery G++ Subscriptions

CodeSourcery provides support contracts for Sourcery G++. This chapter describes these contracts and explains how CodeSourcery customers can access their support accounts.

2.1. About Sourcery G++ Subscriptions

CodeSourcery offers Sourcery G++ subscriptions. Professional Edition subscriptions provide unlimited support, with no per-incident fees. CodeSourcery's support covers questions about installing and using Sourcery G++, the C and C++ programming languages, and all other topics relating to Sourcery G++. CodeSourcery provides updated versions of Sourcery G++ to resolve critical problems. Personal Edition subscriptions do not include support, but do include free upgrades as long as the subscription remains active.

CodeSourcery's support is provided by the same engineers who build Sourcery G++. A Sourcery G++ subscription is like having a team of compiler engineers and programming language experts available as consultants!

Subscription editions of Sourcery G++ also include many additional features not included in the free Lite editions:

- **Sourcery G++ IDE.** The Sourcery G++ IDE, based on Eclipse, provides a fully visual environment for developing applications, including an automated project builder, syntax-highlighting editor, and a graphical debugging interface. The debugger provides features especially useful to embedded systems programmers, including the ability to step through code at both the source and assembly level, view registers, and examine stack traces. CodeSourcery's enhancements to Eclipse include improved support for hardware debugging via JTAG or ICE units and complete integration with the rest of Sourcery G++.

- **Debug Sprites.** Sourcery G++ Debug Sprites provide hardware debugging support using JTAG and ICE devices. On some systems, Sourcery G++ Sprites can automatically program flash memory and display control registers. And the board initialization performed by each Sprite can be customized with simple XML-based configuration files to insert delays and write to particular memory addresses. Debug Sprites included in Lite editions of Sourcery G++ include only a subset of the functionality of the Sprites in the subscription editions.

- **QEMU Instruction Set Simulator.** The QEMU instruction set simulator can be used to run — and debug — programs even without target hardware. Most bare-metal configurations of Sourcery G++ include QEMU and linker scripts targeting the simulator. Configurations of Sourcery G++ for GNU/Linux targets include a user-space QEMU emulator that runs on Linux hosts.

- **Sysroot Utilities.** Subscription editions of Sourcery G++ include a set of sysroot utilities for GNU/Linux targets. These utilities simplify use of the Sourcery G++ dynamic linker and shared libraries on the target and also support remote debugging with **gdbserver**.

- **CS3.** CS3 provides a uniform, cross-platform approach to board initialization and interrupt handling on ARM EABI, ColdFire ELF, fido ELF, and Stellaris EABI platforms.

- **GNU/Linux Prelinker.** For select GNU/Linux target systems, Sourcery G++ includes the GNU/Linux prelinker. The prelinker is a postprocessor for GNU/Linux applications which can dramatically reduce application launch time. CodeSourcery has modified the prelinker to operate on non-GNU/Linux host systems, including Microsoft Windows.

- **Library Reduction Utility.** Sourcery G++ also includes a Library Reduction Utility for GNU/Linux targets. This utility allows the GNU C Library to be relinked to include only those functions used by a given collection of binaries.

- **Additional Libraries.** For some platforms, additional run-time libraries optimized for particular CPUs are available. Pre-built binary versions of the libraries with debug information are also available to subscribers.

If you would like more information about Sourcery G++ subscriptions, including a price quote or information about evaluating Sourcery G++, please send email to <sales@codesourcery.com>.

2.2. Accessing your Sourcery G++ Subscription Account

If you have a Sourcery G++ subscription, you may access your account by visiting the Sourcery G++ Portal[1]. If you have a support account, but are unable to log in, send email to <support@codesourcery.com>.

---

[1] https://support.codesourcery.com/GNUToolchain/

Chapter 3
Sourcery G++ Lite for ARM EABI

This chapter contains information about using Sourcery G++ Lite on your target system. This chapter also contains information about changes in this release of Sourcery G++ Lite. You should read this chapter to learn how to best use Sourcery G++ Lite on your target system.

3.1. Library Configurations

Sourcery G++ includes copies of run-time libraries that have been built with optimizations for different target architecture variants or other sets of build options. Each such set of libraries is referred to as a *multilib*. When you build a target application, Sourcery G++ automatically selects the multilib matching the build options you have selected.

Note that a given multilib may be compatible with additional processors and configurations beyond those explicitly named here.

The following library configurations are available in Sourcery G++ Lite for ARM EABI.

ARMv4 - Little-Endian, Soft-Float	
Command-line option(s):	default

ARMv4 Thumb - Little-Endian, Soft-Float	
Command-line option(s):	`-mthumb`

ARMv7 Thumb-2 - Little-Endian, Soft-Float	
Command-line option(s):	`-mthumb -march=armv7 -mfix-cortex-m3-ldrd`

ARMv6-M Thumb - Little-Endian, Soft-Float	
Command-line option(s):	`-mthumb -march=armv6-m`

3.2. Using Sourcery G++ Lite for ARM EABI

3.2.1. ARMv7M Interrupt handlers

Because of a discrepancy between the ARMv7M Architecture and the ARM EABI, it is not safe to use normal C functions directly as interrupt handlers. The EABI requires the stack be 8-byte aligned, whereas ARMv7M only guarantees 4-byte alignment when calling an interrupt vector. This can cause subtle runtime failures, usually when 8-byte types are used.

Functions that are used directly as interrupt handlers should be annotated with `__attribute__((__interrupt__))`. This tells the compiler to add special stack alignment code to the function prologue.

3.2.2. ARM Interrupt Vectors

The ARM CS3 implementation treats all systems as address vector processors. On traditional ARM systems (everything except M profile devices) an exception causes the processor to jump to a fixed address. However the arrangement of these addresses is such that the CS3 code vector model is not feasible.

The default CS3 vector table emulates an address vector system by placing an indirect branch at the real exception vector. If you override the entire vector table (rather than individual vectors) the indirect stubs are not included, and your replacement table is placed at address zero.

3.2.3. NEON SIMD Code

Sourcery G++ includes support for automatic generation of NEON SIMD vector code. Autovectorization is a compiler optimization in which loops involving normal integer or floating-point code are transformed to use NEON SIMD instructions to process several data elements at once.

To enable generation of NEON vector code, use the command-line options `-ftree-vectorize -mfpu=neon -mfloat-abi=softfp`. The `-mfpu=neon` option also enables generation of VFPv3 scalar floating-point code.

Sourcery G++ also includes support for manual generation of NEON SIMD code using C intrinsic functions. These intrinsics, the same as those supported by the ARM RealView® compiler, are defined in the `arm_neon.h` header and are documented in the 'ARM NEON Intrinsics' section of the GCC manual. The command-line options `-mfpu=neon -mfloat-abi=softfp` must be specified to use these intrinsics; `-ftree-vectorize` is not required.

3.2.4. Half-Precision Floating Point

Sourcery G++ for ARM EABI includes support for half-precision (16-bit) floating point, including the new `__fp16` data type in C and C++, support for generating conversion instructions when compiling for processors that support them, and library functions for use in other cases.

3.2.4.1. Representations

ARM supports two incompatible representations for half-precision floating-point values. You must choose one of the representations and use it consistently in your program. The linker gives an error if objects compiled or assembled with different half-precision float attributes are combined in the same executable.

Compiling or assembling with `-mfp16-format=ieee` selects the representation defined in the *IEEE 754-2008* standard, with 1 sign bit, 5 exponent bits, and 10 significand bits (11 bits of significand precision, approximately 3 decimal digits). This format is capable of representing normalized values in the range of 2^{-14} to 65504. It includes support for infinities and NaNs, following the usual IEEE 754 rules.

ARM also supports an alternative half-precision representation, which you can select with `-mfp16-format=alternative`. This format does not include support for infinities and NaNs. Instead, the range of exponent values is extended, so that this format can represent normalized values in the range of 2^{-14} to 131008.

The default for this option is `-mfp16-format=none`, which disables support for half-precision floats.

3.2.4.2. C and C++ Usage

When you compile with `-mfp16-format=ieee` or `-mfp16-format=alternative`, GCC defines the `__fp16` data type to represent half-precision float values. Objects of this type have a size of 2 bytes and a natural alignment of 2 bytes.

The `__fp16` type is a storage format only. For purposes of arithmetic and other operations, `__fp16` values are automatically promoted to `float`. In addition, you cannot declare a function with a return value or parameters of type `__fp16`.

Note that conversions from `double` to `__fp16` involve an intermediate conversion to `float`. Because of rounding, this can sometimes produce a different result than a direct conversion.

3.2.4.3. Hardware and Library Support

ARM provides hardware support for conversions between `__fp16` and `float` values as an extension to VFP and NEON (Advanced SIMD). GCC generates code using the instructions provided by this extension if you compile with the options `-mfpu=neon-fp16 -mfloat-abi=softfp`, in addition to the `-mfp16-format` option to select a half-precision format.

In other cases, conversions between `__fp16` and `float` values are implemented as library calls.

3.2.5. ABI Compatibility

The Application Binary Interface (ABI) for the ARM Architecture is a collection of standards, published by ARM Ltd. and other organizations. The ABI makes it possible to combine tools from different vendors, including Sourcery G++ and ARM RealView®.

Sourcery G++ implements the ABI as described in these documents, which are available from the ARM Information Center[1]:

*   BSABI - ARM IHI 0036A (25 October 2007)

*   BPABI - ARM IHI 0037A (25 October 2007)

*   EHABI - ARM IHI 0038A (25 October 2007)

*   CLIBABI - ARM IHI 0039A (25 October 2007)

*   AADWARF - ARM IHI 0040A (25 October 2007)

*   CPPABI - ARM IHI 0041A (25 October 2007)

*   AAPCS - ARM IHI 0042B (2 April 2008)

*   RTABI - ARM IHI 0043A (25 October 2007)

*   AAELF - ARM IHI 0044B (2 April 2008)

*   ABI Addenda - ARM IHI 0045A (13 November 2007)

Sourcery G++ currently produces DWARF version 2, rather than DWARF version 3 as specified in AADWARF.

3.2.6. Object File Portability

It is possible to create object files using Sourcery G++ for ARM EABI that are link-compatible with the GNU C library provided with Sourcery G++ for ARM GNU/Linux as well as with the Newlib C library provided with the ARM EABI toolchain. These object files are additionally link-compatible with other ARM C Library ABI-compliant static linking environments and toolchains.

To use this feature, when compiling your files with the bare-metal ARM EABI toolchain define the preprocessor constant `_AEABI_PORTABILITY_LEVEL` to 1 before including any system header files. For example, pass the option `-D_AEABI_PORTABILITY_LEVEL=1` on your compilation command line. No special options are required when linking the resulting object files. When building applications for ARM EABI, files compiled with this definition may be linked freely with those compiled without it.

---

[1] http://infocenter.arm.com

Files compiled in this manner may not use the functions `fgetpos` or `fsetpos`, or reference the type `fpos_t`. This is because Newlib assumes a representation for `fpos_t` that is not AEABI-compliant.

Note that object files are only portable from EABI to GNU/Linux, and not vice versa; object files compiled for ARM GNU/Linux targets cannot be linked into ARM EABI executables.

3.2.7. ARM Profiling Implementation

Profiling is enabled by means of the `-pg` compiler option. In this mode, the compiler inserts a call to `__gnu_mcount_nc` into every function prologue. However, no implementation of `__gnu_mcount_nc` is provided (to do so would be impossible without knowledge of the execution environment).

You must provide your own implementation of `__gnu_mcount_nc` . Here are the requirements:

* On exit, pop the top value from the stack, and place it in the `lr` register. The `sp` register should be adjusted accordingly. For example, this is how to write it as a stub function:

```
        .globl __gnu_mcount_nc
        .type __gnu_mcount_nc, %function
__gnu_mcount_nc:
        mov    ip, lr
        pop    { lr }
        bx     ip
```

* Preserve all other register state except for `r12` and the CPSR condition code bits. In particular all coprocessor state and registers `r0-r3` must be preserved.

* Record and count all occurrences of the function calls in the program. The caller can be determined from the `lr` value stored on the top of the stack (on entry to `__gnu_mcount_nc`), and the callee can be determined from the current value of the `lr` register (i.e. the caller of this function).

* Arrange for the data to be saved to a file named `gmon.out` when the program exits (via `atexit`). Refer to the **gprof** profiler manual for more information.

3.3. Using Flash Memory

Sourcery G++ Lite supports development and debugging of applications loaded into flash memory on ARM EABI targets. There are three steps involved:

1. You must use an appropriate linker script that identifies the ROM memory region on your target board, and locates the program text within that region. Refer to Chapter 6, *CS3™: The CodeSourcery Common Startup Code Sequence* for information about the boards supported by Sourcery G++.

2. Next, load your program into the flash memory on your target board. You must use third-party tools to program the flash memory.

3. Finally, when debugging a program in flash memory, GDB must be told about the ROM region so that it knows where it must use hardware breakpoints to control program execution. If you are using the Sourcery G++ Debug Sprite to debug your program, the Sprite does this automatically, using the memory map provided in the board configuration file. Otherwise, you must provide this information explicitly.

When using GDB from the command line, you can mark the flash memory as read-only by using the command:

```
(gdb) mem start end ro
```

where *start* and *end* define the address range of the read-only memory region.

3.4. Sourcery G++ Lite Release Notes

This section documents Sourcery G++ Lite changes for each released revision.

3.4.1. Changes in Sourcery G++ Lite 2008q3-66

Bug fix for assembly listing.   A bug that caused the assembler to produce corrupted listings (via the `-a` option) on Windows hosts has been fixed.

Reduced RAM usage for semihosting.   The previous change to add support for command-line arguments in semihosted applications increased RAM requirements for all programs using semihosting. This feature has been reimplemented to substantially reduce the amount of additional memory required.

Optimizer bug fix.   A bug that caused an `unrecognizable insn` internal compiler error when compiling at optimization levels above `-O0` has been fixed.

VFP compiler fix.   A compiler bug that resulted in `internal compiler error: output_operand: invalid expression as operand` when generating VFP code has been fixed.

GDB display of source.   A bug has been fixed that prevented GDB from locating debug information in some cases. The debugger failed to display source code for or step into the affected functions.

Profiling support added.   The `-pg` option is now supported by the compiler. You are required to provide a function named `__gnu_mcount_nc`. For more details, see Section 3.2.7, "ARM Profiling Implementation".

Workaround for Cortex-M3 CPU errata.   Errata present in some Cortex-M3 cores can cause data corruption when overlapping registers are used in `LDRD` instructions. The compiler avoids generating these problematic instructions when the `-mfix-cortex-m3-ldrd` or `-mcpu=cortex-m3` command-line options are used. The Sourcery G++ runtime libraries have also been updated to include this workaround.

GDB segment warning.   Some compilers produce binaries including uninitialized data regions, such as the stack and heap. GDB incorrectly displayed the warning `Loadable segment "name" outside of ELF segments` for such binaries; the warning has now been fixed.

Misaligned NEON memory accesses.   A bug has been fixed that caused the compiler to use aligned NEON load/store instructions to access misaligned data when autovectorizing certain loops. The bug affected code compiled with `-mfpu=neon` and loop vectorization enabled with `-O3` or `-ftree-vectorize`.

Sprite crash on error.   A bug has been fixed which sometimes caused the Sourcery G++ Debug Sprite to crash when it attempted to send an error message to GDB.

Portable object file fixes.   Bugs in the `limits.h`, `stdlib.h`, `time.h`, and `setjmp.h` headers have been fixed. These bugs caused compilation errors when using `-D_AEABI_PORTABILITY_LEVEL=1`, as described in Section 3.2.6, "Object File Portability".

Persistent remote server connections.   A GDB bug has been fixed that caused the **target extended-remote** command to fail to tell the remote server to make the connection persistent across program invocations.

3.4.2. Changes in Sourcery G++ Lite 2008q3-39

Definition of `va_list`.   In order to conform to the ABI for the ARM Architecture, the definition of the type of va_list (defined in stdarg.h) has been changed. This change impacts only the mangled names of C++ entities. For example, the mangled name of a C++ function taking an argument of type va_list, or va_list *, or another type involving va_list has changed. Since this is an incompatible change, you must recompile and relink any modules defining or using affected va_list-typed entities.

Thumb-2 assembler fixes.   The Thumb-2 encodings of QADD, QDADD, QSUB, and QDSUB have been corrected. Previous versions of the assembler generated incorrect object files for these instructions. The assembler now accepts the ORN, QASX, QSAX, RRX, SHASX, SHSAX, SSAX, USAX, UHASX, UQSAX, and USAX mnemonics. The assembler now detects and issues errors for invalid uses of register 13 (the stack pointer) and register 15 (the program counter) in many instructions.

Printing casted values in GDB.   A GDB bug that caused incorrect output for expressions containing casts, such as in the print *(Type *)ptr command, has been fixed.

Bug fix for objcopy/strip.   An objcopy bug that corrupted COMDAT groups when creating new binaries has been fixed. This bug also affected **strip -g**.

Improved support for debugging RealView® objects .   GDB support for programs compiled by the ARM RealView® compiler has been improved.

Binutils support for DWARF Version 3.   The **addr2line** command now supports binaries containing DWARF 3 debugging information. The **ld** command can display error messages with source locations for input files containing DWARF 3 debugging information.

NEON improvements.   Several improvements and bug fixes have been made to the NEON Advanced SIMD Extension support in GCC. A problem that caused the autovectorizer to fail in some circumstances has been fixed. Also, many of the intrinsics available via the arm_neon.h header file now have improved error checking for out-of-bounds arguments, and the vget_lane intrinsics that return signed values now produce improved code.

NEON compiler fix.   A compiler bug that resulted in incorrect NEON code being generated has been fixed. Typically the incorrect code occurred when NEON intrinsics were used inside small if statements.

Connecting to the target using a pipe.   A bug in GDB's **target remote | _program_** command has been fixed. When launching the specified _program_ failed, the bug caused GDB to crash, hang, or give a message Error: No Error.

Mixed-case NEON register aliases.   An assembler bug that prevented NEON register aliases from being created with mixed-case names using the .dn and .qn directives has been fixed. Previously only aliases created with all-lowercase or all-uppercase names worked correctly.

Newlib manuals.   The documentation packaged with Sourcery G++ Lite now includes the Newlib C Library and Math Library manuals.

Keil MCBSTM32 Evaluation Board support.   Support for the Keil MCBSTM32 Evaluation Board, with an STMicroelectronics STM32F103RB processor, has been added.

Object file portability. Sourcery G++ for ARM EABI can now produce object files that are link-compatible with the GNU C Library included with Sourcery G++ for ARM GNU/Linux. These object files are additionally link-compatible with other ARM C Library ABI-compliant static linking environments and toolchains. For additional information, refer to Section 3.2.6, "Object File Portability".

Janus 2CC support. GCC now includes a work-around for a hardware bug in Avalent Janus 2CC cores. To compile and link for these cores, use the `-mfix-janus-2cc` compiler option. If you are using the linker directly use the `--fix-janus-2cc` linker option.

ARM exception handling bug fix. A bug in the runtime library has been fixed that formerly caused throwing an unexpected exception in C++ to crash instead of calling the unexpected exception handler. The bug only affected C++ code compiled by non-GNU compilers such as ARM RealView®.

Simulation of programs with command-line arguments. When using the simulator, command-line arguments are now correctly passed to the simulator program via `argc` and `argv`.

Mangling of NEON type names. A bug in the algorithm used by the C++ compiler for mangling the names of NEON types, such as `int8x16_t`, has been fixed. These mangled names are used internally in object files to encode type information in addition to the programmer-visible names of the C++ variables and functions. The new mangled name encoding is more compact and conforms to the ARM C++ ABI.

Errors after loading the debugged program. An intermittent GDB bug has been fixed. The bug could cause a GDB internal error after the **load** command.

Half-precision floating point. Sourcery G++ now includes support for half-precision floating point via the `__fp16` type in C and C++. The compiler can generate code using either hardware support or library routines. For more information, see Section 3.2.4, "Half-Precision Floating Point".

3.4.3. Changes in Sourcery G++ Lite 2008q3-12

Stellaris UDMAERR interrupt vector. The name of the UDMAERR (uDMA Error) interrupt vector provided by CS3 for Luminary Micro Stellaris configurations has been corrected. The correct name is `__cs3_isr_udmaerr`.

GDB update. The included version of GDB has been updated to 6.8.50.20080821. This update adds numerous bug fixes and new features, including support for decimal floating point, improved Thumb mode support, the new **find** command to search memory, the new /m (mixed source and assembly) option to the **disassemble** command, and the new **macro define** command to define C preprocessor macros interactively.

Uppercase operands to IT instructions. The assembler now accepts both uppercase and lowercase operands for the `IT` family of instructions.

NEON autovectorizer fix. A compiler bug that caused generation of bad `VLD1` instructions has been fixed. The bug affected code compiled with `-mfpu=neon -ftree-vectorize`.

Bug fix in simulator reset code. A bug which could cause programs to crash at startup has been fixed in the CS3 reset code used for simulators. The affected linker scripts are `generic.ld`, `generic-vfp.ld`, `generic-m.ld`, and hosted versions of the above.

Output files removed on error. When GCC encounters an error, it now consistently removes any incomplete output files that it may have created.

Atmel AT91SAM7S-EK support. Support for Atmel AT91SAM7S-EK boards has been added.

Placing bss-like regions in load regions. The linker no longer issues an incorrect error message when a bss-like section is placed at specific load region. The linker formerly incorrectly considered the section as taking up space in the load region.

ARMv7 offset out of range errors. An assembler bug that resulted in `offset out of range` errors when compiling for ARMv7 processors has been fixed.

Thumb-2 `MUL` encoding. In Thumb-2 mode, the assembler now encodes `MUL` as a 16-bit instruction (rather than as a 32-bit instruction) when possible. This fix results in smaller code, with no loss of performance.

ARM C++ ABI utility functions. Vector utility functions required by the ARM C++ ABI no longer crash when passed null pointers. The affected functions are `__aeabi_vec_dtor_cookie`, `__aeabi_vec_delete`, `__aeabi_vec_delete3`, and `__aeabi_vec_delete3_nodtor`. These functions are not intended for use by application programmers; they are only called by compiler-generated code. They are not presently used by the GNU C++ compiler, but are used by some other compilers, including ARM's RealView® compiler.

GCC version 4.3.2. Sourcery G++ Lite for ARM EABI is now based on GCC version 4.3.2. For more information about changes from GCC version 4.2 that was included in previous releases, see `http://gcc.gnu.org/gcc-4.3/changes.html`.

Smaller Thumb-2 code. When optimizing for size (i.e., when `-Os` is in use), GCC now generates the 16-bit `MULS` Thumb-2 multiply instruction instead of the 32-bit `MUL` instruction.

Thumb-2 `RBIT` encoding. An assembler bug that resulted in incorrect encoding of the Thumb-2 `RBIT` instruction has been fixed.

Additional Stellaris interrupt vectors. The USB0, PWM3, UDMA and UDMAERR interrupt vectors have been added to the CS3 Luminary Micro Stellaris configurations.

Additional Luminary Micro CPUs. A large number of additional Luminary Micro CPUs are now supported. Linker scripts for these CPUs include the internal ROM region.

Sprite communication improvements. The Sourcery G++ Debug Sprite now uses a more efficient protocol for communicating with GDB. This can result in less latency when debugging, especially when running the Sprite on a remote machine over a network connection.

Marvell Feroceon compiler bug fix. A bug that caused an internal compiler error when optimizing for Marvell Feroceon CPUs has been fixed.

Misaligned accesses to packed structures fix. A bug that caused GCC to generate misaligned accesses to packed structures has been fixed.

`SIZE_MIN` and `SIZE_MAX`. `SIZE_MAX` is now correctly defined in `stdint.h` to be an unsigned quantity equal to the maximum possible value of a `size_t`, as required by the C standard. In addition, the definition of `SIZE_MIN` has been removed, as this constant is not prescribed by the C standard.

Bug fix for objdump on Windows. An objdump bug that caused the `-S` option not to work on Windows in some cases has been fixed.

3.4.4. Changes in Sourcery G++ Lite 2008q1-126

VFP coprocessor initialization. CS3 now includes startup code to enable ARM VFP coprocessors in the QEMU Arm Simulator (`qemu-arm`) and Arm Simulator (VFP) (`generic-vfp`) board configuration files.

CS3 mapping symbols. The CS3 startup code now contains correct ARM EABI mapping symbols. Previously these were missing in some cases, which could confuse some third-party utilities.

Disassembler bug fix. A bug in the disassembler has been fixed that formerly caused **objdump** to crash when processing raw binary files, or other executables with an empty symbol table.

NEON assembler symbols. An assembler bug that caused spurious undefined symbols to be generated has been fixed. The `mov d0, d1` instruction would incorrectly cause an undefined symbol `d1` to be created.

GDB info registers crash fix. Executing **info registers** after executing **flushregs** no longer crashes GDB.

3.4.5. Changes in Sourcery G++ Lite 2008q1-102

GDB and Ctrl-C on Windows . GDB no longer crashes when you press **Ctrl-C** twice during remote debugging to give up waiting for the target.

GDB support for user-defined prefixed commands. The GDB **define** and **document** commands, which allow you to add new commands to the GDB command-line interface, now support creating commands within an existing prefix such as **target**. Hooks for prefixed commands are also supported. Refer to the Debugger manual for more information.

UNC pathname bug fix. A bug has been fixed that caused linker errors on Windows hosts when running a Sourcery G++ toolchain installed in a UNC path (`\\host\directory`).

3.4.6. Changes in Sourcery G++ Lite 2007q3-93

ARM Cortex-A9 processor support. The compiler can now generate code optimized for the ARM Cortex-A9 processor. This is enabled by the the `-mcpu=cortex-a9` command-line option.

MOVW and MOVT relocations. A linker error that resulted in incorrect offsets when processing relocations on `MOVW` and `MOVT` instructions referencing mergeable string sections has been fixed.

Improved argument-passing code. The compiler can now generate more efficient code for certain functions whose arguments must be sign-extended to conform with language or ABI conventions. The required conversion was formerly being performed both in the called function and at all call sites; now the redundant conversion has been eliminated for functions that can only be called within the compilation unit where they are defined.

GDB `qOffsets` crash fix. GDB no longer crashes when a remote stub provides load offsets for an unlinked object file.

Linker error allocating ELF segments. A bug where the linker produces an incorrect error message with segments at the top of the address space has been fixed.

GCC stack size limit increased. On Windows hosts, the maximum stack size for the GCC executable has been increased. This means that more complex programs can be compiled.

Invalid object file after strip. A bug in the assembler has been fixed that formerly caused `.set symbol expression` constructs to emit `symbol` in the wrong section. This in turn caused inconsistent behavior after stripping the symbol table.

GCC update. The GCC package has been updated to version 4.2.3. This version includes numerous bug fixes since GCC 4.2.

License checking on Linux. Sourcery G++'s license-checking logic now includes a workaround for a kernel bug present in some versions of Linux. This bug formerly caused failures with an error message from the `cs-license` component.

Cortex-R4F and VFPv3-D16. Sourcery G++ now supports the ARM Cortex-R4F CPU and the VFPv3-D16 floating-point coprocessor. These can be selected with `-mcpu=cortex-r4f` and `-mfpu=vfpv3-d16`, respectively.

Overlapping operands for long multiply instructions. An incorrect assembler warning has been removed in the case of overlapping source and destination operands for `UMULL`, `SMULL`, `UMLAL` and `SMLAL` instructions on ARMv6 processors.

Additional Luminary Micro CPUs. A large number of additional Luminary Micro CPUs are now supported.

ARMv4 processor support. Sourcery G++ now supports ARMv4 processors such as the ARM8 and StrongARM. To generate code for these processors use the `-march=armv4` command-line option.

Size optimization bug. A code generation bug that caused corruption of function arguments when compiling with `-Os` has been fixed. The corruption occurred as part of the sibling call optimization.

C++ library ABI fix. GCC 4.2.1's `std::type_info` was not fully compatible with earlier versions. The ordering of four virtual functions has been fixed in this update.

GDB update. The included version of GDB has been updated to 6.7.20080107. This update includes numerous bug fixes.

Linker crash on invalid input files. Some older versions of GCC generated object files with invalid mergeable string sections when compiling with `-fmerge-all-constants`. This bug was fixed in Sourcery G++ as of version 4.1-43. However, since system libraries included with some GNU/Linux distributions were affected by this bug, the linker has now been changed to accept object files with such invalid sections, rather than crash or produce an error message.

GDB search path bug fix. A bug in GDB has been fixed that formerly resulted in an internal error when setting `solib-search-path` or `solib-absolute-prefix` after establishing a connection to a remote target.

Binutils update. The binutils package has been updated to version 2.18.50.20080215 from the FSF trunk. This update includes numerous bug fixes.

Read-only variables. The C++ compiler now places variables whose types are instantiations of template classes in a read-only data section if they are declared `const` and initialized with a constant value. This changes reduces the RAM usage of affected applications.

3.4.7. Changes in Sourcery G++ Lite 2007q3-53

Volatile postincrement and postdecrement bug fix. A code generation bug that caused postincrement or postdecrement of a volatile object to reread the modified value from that object in some contexts has been fixed. The bug affected code performing a comparison of the postincrement or postdecrement expression with a constant, or that was optimized to comparison with a constant.

Support for debugging with FlashPro3. Support has been added for debugging with the Actel FlashPro3 JTAG device on Windows hosts. This works only with Actel Cortex-M1 FPGAs.

Static data initialization.   A bug in RAM profile linker scripts has been fixed that caused startup code to erroneously clear the beginning of initialized data, rather than uninitialized data.

C++ class debug information.   The flag `-femit-class-debug-always` is now disabled by default. The flag produces duplicate C++ class debug information as a work-around for older debuggers.

Improved breakpoints in constructors and template functions.   GDB now supports breakpoints on source code locations that have several code addresses associated with them. Setting a breakpoint on a constructor automatically associates the breakpoint with all constructor bodies generated by GCC. If you set a breakpoint on a line of a templated function, GDB breaks at the indicated line in all instantiations of the templated function.

GDB printf %p.   GDB's **printf** command now supports the `"%p"` format specifier.

Widening multiply instructions for ARMv6 and later.   GCC now makes use of the 32-to-64-bit widening multiply instructions (`umull`, `smull`, `umlal`, and `smlal`) when generating code for ARMv6 and later. A bug had caused these instructions to be used for ARMv3 to ARMv5 only.

GDB update.   The included version of GDB has been updated to 6.6.20070821. This update includes numerous bug fixes.

Assembler code file name suffixes.   GCC now recognizes `.sx` as well as `.S` as a file name suffix indicating assembler code which must be preprocessed. The alternate suffix may be useful in conjunction with other program development tools on Windows that do not distinguish case on filenames and treat `.S` the same as `.s`, which GCC uses to indicate assembler code without preprocessing.

3.4.8. Changes in Sourcery G++ Lite 2007q3-33

Preprocessing assembly code.   The compiler driver passes `-I` options to the assembler, so that `#include` directives (processed by the preprocessor) and `.include` directives (processed by the assembler) use the same search path.

Dynamically-initialized `const` variables.   Dynamically-initialized namespace-scope C++ variables are no longer placed in read-only data sections, even when marked `const`. These variables must be modified at startup, so they cannot be placed in ROM, even though their values cannot change once initialized.

Register allocation bug fix.   A register allocation bug has been fixed. Under rare circumstances, the bug caused incorrect code generation.

iWMMXt bug fix.   A GCC bug affecting code generation for iWMMXt processors has been fixed. The bug caused internal compiler errors when compiling some functions with large stack frames.

NEON coprocessor system registers.   The assembler now accepts the `MVFR0` and `MVFR1` coprocessor registers in `fmrx` and `fmxr` instructions.

Disabling diagnostics for use of system header and library directories.   The warnings for use of options such as `-I/usr/include` when cross compiling can be disabled with a new option `-Wno-poison-system-directories`. This option is intended for use in chroot environments when such directories contain the correct headers and libraries for the target system rather than the host.

Default linker script.   GCC no longer uses the simulator linker script by default. To avoid a link failure, you must specify a linker script explicitly with the `-T` command-line option, or via the `Properties` item on the `Project` menu in the Sourcery G++ IDE.

Thumb-2 doubleword writeback addressing modes.   An assembler bug that caused writeback addressing modes for `ldrd` and `strd` to be incorrectly encoded has been fixed.

Stricter check for anonymous unions.   G++ now issues an error about invalid code that uses the same name for a member of an anonymous union and an entity in the surrounding namespace. For example, you will now get an error about code like:

```
int i;
static union { int i; };
```

because both the global variable and the anonymous union member are named `i`. To make this code valid you must change one of the declarations to use a different name.

GCC update.   The GCC package has been updated to version 4.2.1. This version includes numerous bug fixes since GCC 4.2.

Smaller code for C++ destructors.   G++ now generates more compact code to handle the destruction of C++ objects declared at namespace scope or declared within a function scope using the `static` keyword.

Robustness on Microsoft Windows.   Defects that sometimes caused GDB to become non-responsive on Microsoft Windows have been eliminated.

Binutils update.   The binutils package has been updated to the 2007-08-19 version of the pre-2.18 FSF trunk. This contains many new improvements and bug fixes. For more information, refer to the manuals for the individual utilities, and to the binutils web site at `http://www.gnu.org/software/binutils/`.

Debugging information fix.   GCC no longer generates invalid debugging information for sections with no contents. The invalid debugging information caused the GNU/Linux prelinker to crash.

Calls to undefined weak symbols.   The linker now implements semantics that comply to the ARM EABI for `R_ARM_CALL` and `T_ARM_THM_CALL` relocations against undefined weak symbols. These now result in a jump to the next instruction.

Thumb-2 shift instruction aliases.   The assembler now accepts `mov` with shifted operands as an alias for Thumb-2 shift instructions. For example `mov r0, r1, lsl r2` is encoded as `lsl r0, r1, r2`.

Inlined function debugging fix.   GDB now backtraces correctly when stopped at the first instruction of an inlined function. Earlier versions would sometimes encounter internal errors in this situation.

Assembler skipping \ characters.   A bug is fixed where the assembler would skip \ characters when they appeared at certain positions in the input file. This bug primarily affected assembler macros.

Improved diagnostics for region overflow.   The linker will now give more helpful diagnostics when the object files being linked are too big for one of the memory regions defined in the linker script.

EABI object attribute merging.   The linker now properly merges EABI object attributes into its output file.

Thumb-2 exception return instructions. An assembler bug that caused `subs pc, lr, #const` and `movs pc, lr` to be incorrectly encoded has been fixed.

Tag_ABI_PCS_wchar_t object attributes. Objects generated with `-fshort-wchar` are now given the correct `Tag_ABI_PCS_wchar_t` EABI object attribute annotations.

Spurious compiler warnings eliminated. GCC no longer emits warnings when linker-specific command-line options are provided in combination with modes that do not perform linking, such as with the `-c` flag.

Debugging of inlined functions. GDB now supports inlined functions. GDB can include inlined functions in the stack trace; display inlined functions' arguments and local variables; and step into, over, and out of inlined functions.

Uppercase special register names. The assembler now accepts both uppercase and lowercase special register names when assembling `msr` and `mrs` instructions for the Microcontroller profile of the ARM Architecture.

Debugger access to out-of-bounds memory. GDB turns on `inaccessible-by-default` by default, disallowing access to memory outside the regions specified in a board configuration.

Call shortening bug fix. GCC no longer overrides `__attribute__((long_call))` on calls to locally-defined functions when the function is weak, or when it is in a different section from the caller.

CodeSourcery Common Startup Code Sequence. Support for CS3, a unified startup scheme is included.

Binutils update. The binutils package has been updated from version 2.17 to the pre-2.18 FSF trunk. This is a significant update with many improvements and bug fixes.

Changes to the assembler (**as**) include:

- On MIPS targets, support for additional processors and the SmartMIPS and DSP Release 2 extensions has been added.

New linker (**ld**) features include:

- A new command-line option `--default-script` has been added to give more precise control over linker script processing.

- There are new command-line options `-Bsymbolic-functions`, `--dynamic-list`, `--dynamic-list-cpp-new`, and `--dynamic-list-data` to control symbols that should be dynamically linked.

- The new `--print-gc-sections` option lists sections removed by garbage collection.

Other changes include:

- The **objcopy** utility has a new `--extract-symbol` option to extract only symbol table information from the input file.

- The **gprof** utility now allows input files to have histogram records for several memory ranges, provided those ranges are disjoint.

For more information, refer to the manuals for the individual utilities, and the binutils web site at `http://www.gnu.org/software/binutils/`.

GDB update.　　The included version of GDB has been updated to 6.6.50.20070620. This update includes numerous bug fixes.

Forced alignment of array variables.　　A new option `-falign-arrays` has been added to the compiler. Specifying this option sets the minimum alignment for array variables to be the largest power of two less than or equal to their total storage size, or the biggest alignment used on the machine, whichever is smaller. This option may be helpful when compiling legacy code that uses type punning on arrays that does not strictly conform to the C standard.

ARM EABI compliance.　　Objects produced by Sourcery G++ are now marked as ARM ELF version 5 rather than ARM ELF version 4. This reflects compliance with recent revisions of the ARM EABI. Sourcery G++ still accepts objects marked with version 4.

Smaller C++ applications.　　The C++ runtime library has been modified so that using namespace-scope objects with destructors does not pull in unnecessary support functions. Therefore, statically linked C++ applications compiled with `-fno-exceptions` are substantially smaller.

ARMv6-M floating-point bug fix.　　A bug affecting conversion of wider floating-point types to subnormal `float` values on ARMv6-M processors has been fixed.

3.4.9. Changes in Sourcery G++ Lite 2007q1-21

NEON coprocessor system registers.　　The assembler now accepts the `MVFR0` and `MVFR1` coprocessor registers in `fmrx` and `fmxr` instructions.

Disabling diagnostics for use of system header and library directories.　　The warnings for use of options such as `-I/usr/include` when cross compiling can be disabled with a new option `-Wno-poison-system-directories`. This option is intended for use in chroot environments when such directories contain the correct headers and libraries for the target system rather than the host.

Thumb-2 doubleword writeback addressing modes.　　An assembler bug that caused writeback addressing modes for `ldrd` and `strd` to be incorrectly encoded has been fixed.

Thumb-2 shift instruction aliases.　　The assembler now accepts `mov` with shifted operands as an alias for Thumb-2 shift instructions. For example `mov r0, r1, lsl r2` is encoded as `lsl r0, r1, r2`.

EABI object attribute merging.　　The linker now properly merges EABI object attributes into its output file.

Thumb-2 exception return instructions.　　An assembler bug that caused `subs pc, lr, #const` and `movs pc, lr` to be incorrectly encoded has been fixed.

Tag_ABI_PCS_wchar_t object attributes.　　Objects generated with `-fshort-wchar` are now given the correct `Tag_ABI_PCS_wchar_t` EABI object attribute annotations.

Uppercase special register names.　　The assembler now accepts both uppercase and lowercase special register names when assembling `msr` and `mrs` instructions for the Microcontroller profile of the ARM Architecture.

3.4.10. Changes in Sourcery G++ Lite 2007q1-10

Disassembly of overlapping sections.   A bug in the disassembler that caused code to be displayed as data (and vice-versa) in files with overlapping sections has been fixed. This mainly affects the **objdump** utility.

Installer hangs while refreshing environment.   The Sourcery G++ installer for Microsoft Windows now updates the PATH environment variable without waiting for open applications to acknowledge the update. This change prevents open applications from blocking the installer's progress.

Improved assembler diagnostics for 8-bit offsets.   The assembler now correctly diagnoses out-of-range offsets to instructions such as LDRD as 8-bit rather than half-word offsets.

Less disk space required for installation.   Sourcery G++ Lite packages are smaller because multiple copies of files have been replaced with hard and/or symbolic links when possible. Both the size of the installer images and the amount of disk space required for an installed package have been reduced.

Thumb register corruption fix.   A bug in the compiler that could cause register corruption in Thumb mode has been fixed. The compiler was formerly emitting code to restore registers on function return that was not interrupt safe.

__aeabi_lcmp.   An error in the libgcc implementation of __aeabi_lcmp that caused incorrect results to be returned has been fixed. This is a support routine defined by the ARM EABI. GCC does not normally use this routine directly, however it may be used by third-party code.

The \@ assembler pseudo-variable.   A bug in the assembler that caused uses of the \@ pseudo-variable to be mis-parsed as comments has been fixed.

Crash when generating vector code.   A bug that sometimes caused the compiler to crash when invoked with the -ftree-vectorize option has been fixed.

Propagation of Thumb symbol attributes.   Symbols referring to Thumb functions on ARM targets now have their Thumb attribute correctly propagated to any aliases defined with .set or .symver.

Linking of non-ELF images.   A linker bug that could cause a crash when linking non-ELF objects for ARM targets has been fixed.

Invalid load instructions.   A bug in the compiler which caused it to generate invalid assembly (e.g. ldrd r0, [#0, r2]) has been fixed.

VFPv3/NEON debug information.   A bug in the compiler which caused it to generate incorrect debug information for code using VFPv3/NEON registers has been fixed. The debugger is now able unable to locate and display values held in these registers.

ARMv6-M system instructions.   An assembler bug that caused some ARMv6-M system instructions to be incorrectly rejected has been fixed. The affected instructions are msr, mrs, yield, wfi, wfe and sev.

VFP initialization code.   Sourcery G++ now includes startup code to enable VFP coprocessors. This can be selected with -T rdimonvfp-ram.ld.

Assembly of Thumb-2 load/store multiple instructions.   The Thumb-2 ldm and stm assembly mnemonics are now assembled to ldr and str instructions when a single register is transferred, as specified in the Thumb-2 Architecture Supplement.

Conditional Thumb-2 branch instructions. A linker bug that could cause objects involving conditional Thumb-2 branch instructions to be incorrectly rejected has been fixed.

Alignment bug fix. A bug has been fixed that formerly caused incorrect code to be generated in some situations for copying structure arguments being passed by value. The incorrect code caused alignment errors on stack accesses on some targets.

3.4.11. Changes in Sourcery G++ Lite 2007q1-3

Marvell Feroceon support. Sourcery G++ Lite now generates code optimized for Marvell Feroceon CPUs when the `mcpu=marvell-f` option is specified. This option also selects runtime libraries optimized for this processor.

Handling of out-of-range values by `strtof` . The `strtof` function now sets `errno` to `ERANGE` when the input is not representable as a `float`, as required by the ISO C standard.

Fix --gc-sections and C++ exceptions. A bug in the `--gc-sections` linker option has been fixed. Previously this would incorrectly remove unwinding tables, breaking C++ applications that use exceptions.

Assembly of SRS instructions. An assembler bug that resulted in incorrect encoding of the Thumb-2 `SRS` instruction has been fixed. In addition the assembler supports explicit specification of the base register, as accepted by other ARM toolchains.

Symbols defined in linker scripts. A bug is fixed that caused the linker to crash in some circumstances when a linker script defined a symbol in an output section. Typically usage is where the script contained a `__DATA_LOAD = LOADADDR(.data)` statement in the `.data` section.

VFP disassembly crash. A bug that caused crashes when disassembling some forms of the VFP `fmrx` and `fmxr` instructions has been fixed.

ARM NEON store intrinsics bug fix. A compiler bug that incorrectly caused calls to ARM NEON store intrinsics (such as `vst1_u8`) to be optimized away has been fixed.

Improvements to ARM NEON support. The ARM NEON support in GCC has been enhanced to comply with new rules for containerized vector types specified in the ARM procedure call standard. Additionally, the compiler now rejects implicit conversions between NEON polynomial vector types and NEON integer vector types of the same layout.

Complex numbers bug fix. A bug that could lead to incorrect code generation for code using complex numbers has been fixed.

Simulation of `lseek`. The ARM simulator no longer sets `errno` to `EBADF` when `lseek` is applied to file descriptor zero. Instead, the simulator performs the simulated seek, and accurately reports the result.

Use of system header and library directories diagnosed. The compiler and linker now diagnose the incorrect use of native system header and library directories for cross-compilation. This typically arises from options such as `-I/usr/X11R6/include` hard-coded in build scripts written without a view to cross-compilation.

Improved debugging for optimized code. GDB's ability to print and change variables' values in optimized code is improved. GDB now tracks variable scopes more accurately, making better use of the detailed debugging information produced by Sourcery G++ compilers.

Improved handling of Windows paths in GDB. GDB now properly recognizes the names of source files that were passed to the compiler using an absolute path on Windows. You may refer to the file either by its base name (without any leading directory components), by the exact path passed to the compiler, or by its absolute path.

Initialization priorities. The `constructor` and `destructor` function attributes now accept an optional priority argument. Constructors with small priorities are run before those with larger priorities; the opposite is true for destructors. For example:

```
void f __attribute__((constructor(500)));
void f() {
  /* Perform initialization.  */
}
```

defines a function `f` with priority 500. This function will be run before constructors with larger priorities. Constructor and destructors with no explicit priority argument have priority 65535, the maximum permitted value.

Thumb-2 IT block code generation error fixed. A bug in Thumb-2 code generation has been fixed. This bug would result in missing IT instructions, causing the assembler to reject the code.

ARM Cortex-R4 performance improvements. Sourcery G++ Lite now generates faster code when compiling for the ARM Cortex-R4 processor by scheduling instructions for the processor's pipelines. To generate code for this processor, use the `-mcpu=cortex-r4` command-line option.

iWMMXt compiler errors. A compiler bug that caused invalid assembly when generating iWMMXt code has been fixed.

GDB update. The included version of GDB has been updated to 6.6.50.20070228. This update includes numerous bug fixes and improved support for C++ pointers to members.

Assembling Thumb store-multiple instructions. The assembler now issues an error message instead of crashing on load/store multiple instructions that incorrectly use Thumb-2 addressing modes (e.g., `ldmdb`) in legacy Thumb syntax mode. If you want to use these address modes, you should use unified syntax mode instead.

Thumb-2 stack decrement misassembly. An assembler bug that resulted in incorrect encoding of the 32-bit Thumb-2 form of the `sub sp, sp, #const` instruction has been fixed. Previously this was misassembled as `subs`.

Naked functions. Functions marked with `__attribute__((naked))` no longer contain prologue and epilogue code. Please refer to the GCC manual for the proper use of this attribute.

Destructor execution order. Prioritized destructors (whether specified with the `destructor` function attribute in C or the `init_priority` variable attribute in C++) are now executed in the correct order. Previous releases ran the destructors in an indeterminate order.

Execution order of `.fini_array` entries. For ELF targets using `.init_array` and `.fini_array` sections, the functions whose addresses are stored in `.fini_array` are now run in reverse order (i.e., the last function in the array is run first), as required by the ELF specification. Previous releases ran the entries in forward order (i.e., the first function in the array was run first).

Fix addr2line defect. The binary utility **addr2line** now operates correctly on 64-bit targets with DWARF2 debug information.

Thumb-2 assembler infinite loop.   An assembler bug that would cause it to enter an infinite loop when processing some Thumb-2 assembly has been fixed.

Assembler warnings about overlapping multiplication operands.   The assembler no longer warns about overlapping `Rd` and `Rm` operands when assembling `mul` and `mla` instructions for the ARM architecture version six or above.

ARMv6-M support.   Sourcery G++ Lite can now generate code for the ARMv6-M architecture profile, including Cortex-M1 CPUs. This can be selected with the `-march=armv6-m` or `-mcpu=cortex-m1` command-line options.

ARM Cortex-A8 performance improvements.   Sourcery G++ Lite now generates faster code when compiling for the ARM Cortex-A8 processor by scheduling instructions for the processor's dual-issue pipelines. To generate code for this processor, use the `-mcpu=cortex-a8` command-line option.

GCC version 4.2.   Sourcery G++ Lite for ARM EABI is now based on GCC version 4.2. For more information about changes from GCC version 4.1 that was included in previous releases, see `http://gcc.gnu.org/gcc-4.2/changes.html`.

Improve handling of corrupt debug information.   The binary utility **readelf** now copes more gracefully with corrupted DWARF 2 information.

Smaller C++ programs.   Rarely-used functions in the C++ runtime library have been isolated into separate object files so that they will not be included unless needed. As a result, most statically linked C++ programs are smaller.

Semihosting compatibility improvements.   The semihosting I/O implementation has been improved to work with a wider range of simulators and host implementations.

3.4.12. Changes in Sourcery G++ Lite 4.1-37

Preserve volatile accesses.   Reads from volatile memory are no longer incorrectly optimized away at higher optimization levels.

3.4.13. Changes in Sourcery G++ Lite 4.1-34

Implicit conversions between generic vector types.   Implicit conversions between generic vector types are now only permitted when the two vectors in question have the same number of elements and compatible element types. (Note that the restriction involves *compatible* element types, not implicitly-convertible element types: thus, a vector type with element type `int` may not be implicitly converted to a vector type with element type `unsigned int`.) This restriction, which is in line with specifications for SIMD architectures such as AltiVec, may be relaxed using the flag `-flax-vector-conversions`. This flag is intended only as a compatibility measure and should not be used for new code.

`type_info` comparison fix.   Comparison of `type_info` objects now uses pointer comparison where possible.

C++ forced unwinding fixes.   Some bugs relating to forced unwinding through C++ code have been fixed.

Support for additional Stellaris boards.   Linker scripts are provided for the 6xx and 8xx series Stellaris boards.

Linux support for USB Debug Sprite.   A new driver is included to allow the Sourcery G++ Lite USB Debug Sprite to run on Linux hosts. See Chapter 3, *Sourcery G++ Lite for ARM EABI* for additional information.

3.4.14. Changes in Sourcery G++ Lite 4.1-33

Linker scripts.   A bug is fixed where an erroneous linker script would cause a linker crash. An error message is now produced.

Newlib memory use improvements.   The memory overhead of linking with newlib is reduced. Applications that use only a minimal set of library features may now require significantly less memory.

3.4.15. Changes in Sourcery G++ Lite 4.1-31

Compiler alias analysis.   The type-based alias analysis performed by the compiler when compiling with `-O2` or with `-fstrict-aliasing` is now more conservative. The more aggressive analysis used in previous versions sometimes resulted in incorrect code generation.

Fully relocatable preprocessor.   When cross-compiling, the default preprocessor search path includes only the directories present in the installed toolchain. This speeds up the preprocessor and prevents the unintentional use of unrelated files and directories on the machine where it is installed.

3.4.16. Changes in Sourcery G++ Lite 4.1-29

Support for new-style symbol hashing.   Support has been added in binutils and the prelinker for new-style (also known as `DT_GNU_HASH`) symbol hashing. This can dramatically speed up symbol resolution time and is particularly applicable in environments where full prelinking is not possible (for example where shared libraries are dynamically opened at runtime). The new-style hashing may be enabled by passing `--hash-style=gnu` to the linker.

Prelinker update.   The prelinker has been updated to the current upstream sources and some bugs affecting operation have been fixed.

3.4.17. Changes in Sourcery G++ Lite 4.1-28

Improved support for ROM debugging.   GDB now determines ROM regions automatically from the memory map included in target configuration files. This information is used to determine when hardware breakpoints should automatically be used (for instance the **step**, **next** and **finish** commands). Separate ROM configurations have been removed from the Eclipse debugger menu. The Eclipse GUI has been extended to provide improved support for debugging programs in ROM, when a memory map is not automatically available.

3.4.18. Changes in Sourcery G++ Lite 4.1-27

Rename Windows executables.   The Windows host tools **make.exe** and **rm.exe** are now named **cs-make.exe** and **cs-rm.exe**. This change avoids conflicts with tools provided by other distributors.

iWMMXt bug fixes.   Some bugs involving incorrect code generation and internal compiler errors when generating iWMMXt code have been fixed.

Cortex-M3 startup code.   The ARMv7M startup code (`armv7m-crt0.o`) incorrectly contained ARM code. This has been replaced with Thumb-2 code.

ARM EABI coverage testing support. Coverage testing using GCOV is now supported for the ARM EABI target. Please refer to the *GNU C Compiler Manual (HTML)* for more information on coverage testing.

3.4.19. Changes in Sourcery G++ Lite 4.1-23

Windows debugging fix. In recent releases of Sourcery G++ Lite, the GDB **target remote |** command would hang on Windows. This affected both command line and Eclipse debugging when using the Sourcery G++ Lite Debug Sprite.

Stellaris USB Debug Sprite improvements. The former USB Debug Stub, **armswd**, is now known as the USB Debug Sprite, and has been renamed to **arm-stellaris-eabi-sprite**. In addition, its initialization sequence has been updated to recognize the r1p1 release of the Cortex-M3 processor.

Incompatible changes to Stellaris linker scripts. Sourcery G++ Lite now supports linking executables to run from RAM as well as ROM. As part of this change, there are now separate RAM and ROM versions of the linker scripts for each supported board, and the former ROM-based versions have been renamed. For example, if you were formerly linking with `-T lm3s10x.ld`, you should now use `-T lm3s10x-rom.ld` to get the same behavior.

3.4.20. Changes in Sourcery G++ Lite 4.1-21

Eclipse debuggers. Eclipse configurations for debugging arm-none-eabi applications using the GDB simulator and remote debug stubs have been added.

iWMMXt2 support. The assembler and disassembler now support iWMMXt2 instructions.

NEON intrinsics support. GCC now supports NEON intrinsics defined in the `arm_neon.h` header. These are the same intrinsics supported by the ARM RealView® compiler and are documented in the 'ARM NEON Intrinsics' section of the GCC manual.

3.4.21. Changes in Sourcery G++ Lite 4.1-19

ARMv4t linux multilib. Linux configurations now support ARMv4t CPUs.

Linker scripts. Several problems with the linker scripts for bare-metal targets have been fixed.

3.4.22. Changes in Sourcery G++ Lite 4.1-18

Binutils update. The binutils in this release is based on the final binutils 2.17 release.

GDB update. The included version of GDB has been upgraded to 6.5.50.20060822. This includes numerous bug fixes from the previous version.

GDB support for flash memory. The GDB **load** command can now write to flash memory, if the remote debugging stub contains appropriate support.

Compiler support for NEON. Initial GCC support for autovectorization and generation of NEON SIMD instructions has been added.

Bare metal Cortex-M3 configurations. Bare metal configurations now support generating images for use on ARMv7M devices (eg. Cortex-M3).

iWMMXt support in GLIBC. GLIBC's `setjmp` and `longjmp` now support saving and restoring iWMMXt registers on hardware with those registers. This requires a kernel reporting `iwmmxt` in the `Features` entry in `/proc/cpuinfo`.

iWMMXt exception handling support.   Exception handling now restores the values of iWMMXt registers correctly.

Corrected IPC functions.   A bug in GLIBC's `msgctl`, `semctl`, and `shmctl` functions has been corrected.

3.4.23. Changes in Sourcery G++ Lite 4.1-16

GCC update.   This release is based on GCC 4.1.1.

Fully relocatable compiler.   The compiler now searches for its components only in the directory where it has been installed, and no longer also searches pathnames matching the directory where it was configured. This speeds up the compiler and prevents problems with unintentionally finding unrelated files or directories on the machine where it has been installed.

Stack permission marking for ARM GNU/Linux.   Non-executable stacks can provide increased security against some forms of buffer overflow attacks. The tools involved must coordinate the annotation of required stack permissions, either executable, or non-executable. For ARM GNU/Linux targets the compiler now outputs annotations indicating the required stack permissions.

3.4.24. Changes in Sourcery G++ Lite 4.1-15

Stabs debugging information support.   Using the Stabs debugging format (available with `-gstabs` or `-gstabs+`) now works in conjunction with `-mthumb`. CodeSourcery recommends the default DWARF debugging format (available with `-g`) as DWARF is a more comprehensive debugging format.

3.4.25. Changes in Sourcery G++ Lite 4.1-13

Stellaris linker scripts in IDE.   Linker scripts may now be selected via a drop-down menu in Eclipse.

Stellaris linker scripts for 3xx series CPUs.   The linker scripts for 3xx Series CPUs now place the ISR vector at address zero, as required by all Cortex-M3 cores.

Stellaris USB Debug Sprite improvements.   Bug fixes and new features include:

• A bug that caused the stub not to correctly update the program counter and other register values was fixed. As a result of this fix, it is now possible to run programs residing in SRAM using the `continue` command from GDB.

• The stub no longer prints status messages via GDB console output when invoked with the `-q` command-line option.

• The stub's initialization sequence was updated to recognize revision C Cortex-M3 hardware.

3.4.26. Changes in Sourcery G++ Lite 4.1-9

Stellaris USB Debug Sprite improvements.   Program images exceeding 4K can now be uploaded to flash memory.

Additional Stellaris boards supported.   The Stellaris 301, 310, 315, and 316 CPUs are now supported. Linker scripts have been added for these boards.

3.4.27. Changes in Sourcery G++ Lite 4.1-8

Stellaris USB Debug Sprite improvements. Several bug fixes and enhancements were made to the USB Debug Stub. In particular:

- Bugs in the implementation of `open`, `read`, and `lseek` were fixed.

- Support was added for `isatty`, `rename`, `unlink`, and `system`.

- Memory reads that span 4K block boundaries now work correctly.

3.4.28. Changes in Sourcery G++ Lite 4.1-4

Runtime libraries. Support for ARMv7 including Cortex-M3 and pure Thumb-2.

Assembler. Support for NEON and VFPv3, including unified NEON/VFP syntax.

3.4.29. Changes in Sourcery G++ Lite 4.1-1

Initial release. This release is based on GCC 4.1.0.

Chapter 4
Installation and Configuration

This chapter explains how to install Sourcery G++ Lite. You will learn how to:

1. Verify that you can install Sourcery G++ Lite on your system.

2. Download the appropriate Sourcery G++ Lite installer.

3. Install Sourcery G++ Lite.

4. Configure your environment so that you can use Sourcery G++ Lite.

4.1. Terminology

Throughout this document, the term *host system* refers to the system on which you run Sourcery G++ while the term *target system* refers to the system on which the code produced by Sourcery G++ runs. The target system for this version of Sourcery G++ is `arm-none-eabi`.

If you are developing a workstation or server application to run on the same system that you are using to run Sourcery G++, then the host and target systems are the same. On the other hand, if you are developing an application for an embedded system, then the host and target systems are probably different.

4.2. System Requirements

4.2.1. Host Operating System Requirements

This version of Sourcery G++ supports the following host operating systems and architectures:

• Microsoft Windows NT 4, Windows 2000, Windows XP, and Windows Vista systems using IA32, AMD64, and EM64T processors.

• GNU/Linux systems using IA32, AMD64, or EM64T processors, including Debian 3.0 (and later), Red Hat Enterprise Linux 3 (and later), and SuSE Enterprise Linux 8 (and later).

Sourcery G++ is built as a 32-bit application. Therefore, even when running on a 64-bit host system, Sourcery G++ requires 32-bit host libraries. If these libraries are not already installed on your system, you must install them before installing and using Sourcery G++ Lite. Consult your operating system documentation for more information about obtaining these libraries.

4.2.2. Host Hardware Requirements

In order to install and use Sourcery G++ Lite, you must have at least 128MB of available memory.

The amount of disk space required for a complete Sourcery G++ Lite installation directory depends on the host operating system and the number of target libraries included. Typically, you should plan on at least 400MB. In addition, the graphical installer requires a similar amount of temporary space during the installation process.

4.2.3. Target System Requirements

See Chapter 3, *Sourcery G++ Lite for ARM EABI* for requirements that apply to the target system.

4.3. Downloading an Installer

If you have received Sourcery G++ Lite on a CD, or other physical media, then you do not need to download an installer. You may skip ahead to Section 4.4, "Installing Sourcery G++ Lite".

If you have a Sourcery G++ subscription (or evaluation), then you can log into the Sourcery G++ Portal[1] to download your Sourcery G++ toolchain(s). CodeSourcery also makes some toolchains available to the general public from the Sourcery G++ web site[2]. These publicly available toolchains do not include all the functionality of CodeSourcery's product releases.

---

[1] https://support.codesourcery.com/GNUToolchain/
[2] http://www.codesourcery.com/gnu_toolchains/

Once you have navigated to the appropriate web site, download the installer that corresponds to your host operating system. For Microsoft Windows systems, the Sourcery G++ installer is provided as an executable, with the `.exe` extension. For GNU/Linux systems Sourcery G++ Lite is provided as an executable installer package with the `.bin` extension, or as a compressed archive `.tar.bz2`.

On Microsoft Windows systems, save the installer to the desktop. On GNU/Linux systems, save the download package in your home directory.

4.4. Installing Sourcery G++ Lite

The method used to install Sourcery G++ Lite depends on your host system.

4.4.1. Installing Sourcery G++ Lite on Microsoft Windows

If you have received Sourcery G++ Lite on CD, insert the CD in your computer. On most computers, the installer then starts automatically. If your computer has been configured not to automatically run CDs, open `My Computer`, and double click on the CD. If you downloaded Sourcery G++ Lite, double-click on the installer.

After the installer starts, follow the on-screen dialogs to install Sourcery G++ Lite.

If you prefer, you can run the installer in console mode rather than using the graphical interface. To do this, invoke the installer with the `-i console` command-line option. For example:

```
> /path/to/package.exe -i console
```

4.4.2. Using the Sourcery G++ Lite Installer on GNU/Linux Hosts

Start the graphical installer by invoking the executable shell script:

```
> /bin/sh ./path/to/package.bin
```

After the installer starts, follow the on-screen dialogs to install Sourcery G++ Lite.

If you prefer, or if your host system does not run the X Window System, you can run the installer in console mode rather than using the graphical interface. To do this, invoke the installer with the `-i console` command-line option. For example:

```
> /bin/sh ./path/to/package.bin -i console
```

4.4.3. Installing Sourcery G++ Lite on Solaris or GNU/Linux Hosts from a Compressed Archive

You do not need to be a system administrator to install Sourcery G++ Lite on a GNU/Linux or Solaris system. You may install Sourcery G++ Lite using any user account and in any directory to which you have write access. This guide assumes that you have decided to install Sourcery G++ Lite in the `$HOME/CodeSourcery` subdirectory of your home directory and that the filename of the package you have downloaded is `/path/to/package.tar.bz2`. After installation the toolchain will be in `$HOME/CodeSourcery/sourceryg++-2008q3`.

First, uncompress the package file:

```
> bunzip2 /path/to/package.tar.bz2
```

Next, create the directory in which you wish to install the package:

```
> mkdir -p $HOME/CodeSourcery
```

Change to the installation directory:

```
> cd $HOME/CodeSourcery
```

Unpack the package:

```
> tar xf /path/to/package.tar
```

4.5. Installing Sourcery G++ Lite Updates

If you have already installed an earlier version of Sourcery G++ Lite for ARM EABI on your system, it is not necessary to uninstall it before using the installer to unpack a new version in the same location. The installer detects that it is performing an update in that case.

If you are installing an update from a compressed archive, it is recommended that you remove any previous installation in the same location, or install in a different directory.

Note that the names of the Sourcery G++ commands for the ARM EABI target all begin with **arm-none-eabi**. This means that you can install Sourcery G++ for multiple target systems in the same directory without conflicts.

4.6. Uninstalling Sourcery G++ Lite

The method used to uninstall Sourcery G++ Lite depends on your host system. If you have modified any files in the installation it is recommended that you back up these changes. The uninstall procedure may remove the files you have altered.

4.6.1. Uninstalling Sourcery G++ Lite on Microsoft Windows

Select `Start`, then `Control Panel`. Select `Add or Remove Programs`. Scroll down and click on `Sourcery G++ for ARM EABI`. Select `Change/Remove` and follow the on-screen dialogs to uninstall Sourcery G++ Lite.

You can run the uninstaller in console mode, rather than using the graphical interface, by invoking the Uninstall executable found in your Sourcery G++ Lite installation directory with the `-i console` command-line option.

To uninstall third-party drivers bundled with Sourcery G++ Lite, first disconnect the associated hardware device. Then use `Add or Remove Programs` to remove the drivers separately. Depending on the device, you may need to reboot your computer to complete the driver uninstall.

4.6.2. Uninstalling Sourcery G++ Lite on Microsoft Windows Vista

Select `Start`, then `Settings` and finally `Control Panel`. Select the `Uninstall a program` task. Scroll down and double click on `Sourcery G++ for ARM EABI`. Follow the on-screen dialogs to uninstall Sourcery G++ Lite.

You can run the uninstaller in console mode, rather than using the graphical interface, by invoking the Uninstall executable found in your Sourcery G++ Lite installation directory with the `-i console` command-line option.

To uninstall third-party drivers bundled with Sourcery G++ Lite, first disconnect the associated hardware device. Then use `Uninstall a program` to remove the drivers separately. Depending on the device, you may need to reboot your computer to complete the driver uninstall.

4.6.3. Using the Sourcery G++ Lite Uninstaller on GNU/Linux

If you installed Sourcery G++ Lite on GNU/Linux using the installer script, then you must use the corresponding uninstaller to remove Sourcery G++ Lite. The `arm-none-eabi` directory located in the install directory will be removed entirely by the uninstaller. Please back up any changes you have made to this directory, such as modified linker scripts.

Start the graphical uninstaller by invoking the executable Uninstall shell script located in your installation directory. After the uninstaller starts, follow the on-screen dialogs to uninstall Sourcery G++ Lite.

You can run the uninstaller in console mode, rather than using the graphical interface, by invoking the Uninstall script with the `-i console` command-line option.

4.6.4. Uninstalling Sourcery G++ Lite on GNU/Linux or Solaris

If you installed Sourcery G++ Lite from a `.tar.bz2` file, you can uninstall it by manually deleting the installation directory created in the install procedure.

4.7. Setting up the Environment

As with the installation process itself, the steps required to set up your environment depend on your host operating system.

4.7.1. Setting up the Environment on Microsoft Windows

On a non-Vista Microsoft Windows system, the installer automatically adds Sourcery G++ to your `PATH`. You can test that your `PATH` is set up correctly by using the following command:

```
> arm-none-eabi-g++ -v
```

and verifying that the last line of the output contains: `Sourcery G++ Lite 2008q3-66`.

On a Microsoft Windows Vista system, the installer does not automatically add Sourcery G++ to your `PATH`. To set up your `PATH` on Microsoft Windows Vista, use the following command in a `cmd.exe` shell:

```
> setx PATH "%PATH%;C:\Program Files\Sourcery G++\bin"
```

where `C:\Program Files\Sourcery G++` should be changed to the path of your Sourcery G++ Lite installation. You can verify that the command worked by starting a second `cmd.exe` shell and running:

```
> arm-none-eabi-g++ -v
```

Verify that the last line of the output contains: `Sourcery G++ Lite 2008q3-66`.

4.7.1.1. Working with Cygwin

Sourcery G++ Lite does not require Cygwin or any other UNIX emulation environment. You can use Sourcery G++ directly from the Windows command shell. You can also use Sourcery G++ from within the Cygwin environment, if you prefer.

The Cygwin emulation environment translates Windows path names into UNIX path names. For example, the Cygwin path `/home/user/hello.c` corresponds to the Windows path `c:\cygwin\home\user\hello.c`. Because Sourcery G++ is not a Cygwin application, it does not, by default, recognize Cygwin paths.

If you are using Sourcery G++ from Cygwin, you should set the `CYGPATH` environment variable. If this environment variable is set, Sourcery G++ Lite automatically translates Cygwin path names into Windows path names. To set this environment variable, type the following command in a Cygwin shell:

```
> export CYGPATH=cygpath
```

To resolve Cygwin path names, Sourcery G++ relies on the **cygpath** utility provided with Cygwin. You must provide Sourcery G++ with the full path to `cygpath` if **cygpath** is not in your `PATH`. For example:

```
> export CYGPATH=c:/cygwin/bin/cygpath
```

directs Sourcery G++ Lite to use `c:/cygwin/bin/cygpath` as the path conversion utility. The value of `CYGPATH` must be an ordinary Windows path, not a Cygwin path.

4.7.2. Setting up the Environment on GNU/Linux or Solaris

If you installed Sourcery G++ Lite using the `.bin` graphical installer then you may skip this step. The graphical installer does this setup for you.

Before using Sourcery G++ Lite you should add it to your `PATH`. The command you must use varies with the particular command shell that you are using. If you are using the C Shell (**csh** or **tcsh**), use the command:

```
> setenv PATH $HOME/CodeSourcery/sourceryg++-2008q3/bin:$PATH
```

If you are using Bourne Shell (**sh**), the Korn Shell (**ksh**), or another shell, use:

```
> PATH=$HOME/CodeSourcery/sourceryg++-2008q3/bin:$PATH
> export PATH
```

If you are not sure which shell you are using, try both commands. In both cases, if you have installed Sourcery G++ Lite in an alternate location, you must replace the directory above with `bin` subdirectory of the directory in which you installed Sourcery G++ Lite.

You may also wish to set the `MANPATH` environment variable so that you can access the Sourcery G++ manual pages, which provide additional information about using Sourcery G++. To set the `MANPATH` environment variable, follow the same steps shown above, replacing `PATH` with `MANPATH`, and `bin` with `share/doc/sourceryg++-arm-none-eabi/man`.

You can test that your `PATH` is set up correctly by using the following command:

```
> arm-none-eabi-g++
```

and verifying that you receive the message:

```
arm-none-eabi-g++: no input files
```

Chapter 5
Using Sourcery G++ from the Command Line

This chapter demonstrates the use of Sourcery G++ Lite from the command line. This chapter assumes you have installed Sourcery G++ Lite as described in Chapter 4, *Installation and Configuration*.

5.1. Building an Application

This chapter explains how to build an application with Sourcery G++ Lite using the command line. As elsewhere in this manual, this section assumes that your target system is arm-none-eabi, as indicated by the **arm-none-eabi** command prefix.

Using an editor (such as **notepad** on Microsoft Windows or **vi** on UNIX-like systems), create a file named `hello.c` containing the following simple program:

```
#include <stdio.h>

int
main (void)
{
  printf("Hello World!\n");
  return 0;
}
```

Compile and link this program using the command:

```
> arm-none-eabi-gcc -o hello hello.c -T script
```

Sourcery G++ requires that you specify a linker script with the `-T` option to build applications for bare-board targets. Linker errors like `undefined reference to `read'` are a symptom of failing to use an appropriate linker script. Default linker scripts are provided in `arm-none-eabi/lib`. Refer to Chapter 6, *CS3™: The CodeSourcery Common Startup Code Sequence* for information about the boards and linker scripts supported by Sourcery G++ Lite.

There should be no output from the compiler. (If you are building a C++ application, instead of a C application, replace **arm-none-eabi-gcc** with **arm-none-eabi-g++**.)

5.2. Running Applications on the Target System

Consult your target board documentation for instructions on loading programs onto the target, and running them. Alternatively, you can use the Sourcery G++ Debug Sprite from within GDB to download and run programs on the target via a supported hardware debugging device. See Chapter 7, *Sourcery G++ Debug Sprite* for details.

5.3. Running Applications in the Simulator

Sourcery G++ Lite includes a simulator that you can use on the host system to run programs compiled for the target system. Since you do not need target hardware, this is the easiest way to try out Sourcery G++.

To use the simulator run:

```
> arm-none-eabi-run hello
```

You should see the expected output:

```
Hello, world!
```

You can also use the simulator to execute target programs when debugging with GDB. See Section 5.4, "Running Applications from GDB" for more information.

5.4. Running Applications from GDB

You can run GDB, the GNU Debugger, on your host system to debug programs running remotely on a target board or system. You can also run and debug programs using the GDB simulator.

While this section explains the alternatives for using GDB to run and debug application programs, explaining the use of the GDB command-line interface is beyond the scope of this document. Please refer to the GDB manual for further instructions.

5.4.1. Connecting to the GDB Simulator

GDB includes a simulator that allows you to debug ARM EABI applications without target hardware. To start and connect to the simulator from within GDB, use this command:

```
(gdb) target sim
```

5.4.2. Connecting to the Sourcery G++ Debug Sprite

The Sourcery G++ Debug Sprite is a program that runs on the host system to support hardware debugging devices. You can use the Debug Sprite to run and debug programs on a target board without an operating system, or to debug an operating system kernel. See Chapter 7, *Sourcery G++ Debug Sprite* for detailed information about the supported devices.

You can start the Sprite directly from within GDB:

```
(gdb) target remote | arm-none-eabi-sprite arguments
```

Refer to Section 7.3, "Sourcery G++ Debug Sprite Options" for a full description of the Sprite arguments.

5.4.3. Connecting to an External GDB Server

From within GDB, you can connect to a running **gdbserver** or other debugging stub that uses the GDB remote protocol using:

```
(gdb) target remote host:port
```

where *host* is the host name or IP address of the machine the stub is running on, and *port* is the port number it is listening on for TCP connections.

Chapter 6
CS3™: The CodeSourcery
Common Startup Code Sequence

CS3 is CodeSourcery's low-level board support library. This chapter describes the organization of the system startup code and tells you how you can customize it, such as by defining your own interrupt handlers. This chapter also documents the boards supported by Sourcery G++ Lite and the compiler and linker options you need to use with them.

Many developers turn to the GNU toolchain for its cross-platform consistency: having a single system support so many different processors and boards helps to limit risk and keep learning curves gentle. Historically, however, the GNU toolchain has lacked a consistent set of conventions for processor- and board-level initialization, language run-time setup, and interrupt and trap handler definition.

The CodeSourcery Common Startup Code Sequence (CS3) addresses this problem. For each supported system, CS3 provides a set of linker scripts describing the system's memory map, and a board support library providing generic reset, startup, and interrupt handlers. These scripts and libraries all follow a standard set of conventions across a range of processors and boards.

6.1. Startup Sequence

CS3 divides the startup sequence into three phases:

- In the *hard reset phase*, we initialize the memory controller and set up the memory map.

- In the *assembly initialization phase*, we prepare the stack to run C code, and jump to the C initialization function.

- In the *C initialization phase*, we initialize the data areas, run constructors for statically-allocated objects, and call `main`.

The hard reset and assembly initialization phases are necessarily written in assembly language; at reset, there may not yet be stack to hold compiler temporaries, or perhaps even any RAM accessible to hold the stack. These phases do the minimum necessary to prepare the environment for running simple C code. Then, the code for the final phase may be written in C; CS3 leaves as much as possible to be done at this point.

The CodeSourcery board support library provides default code for all three phases. The hard reset phase is implemented by board-specific code. The assembly initialization phase is implemented by code specific to the processor family. The C initialization phase is implemented by generic code.

6.1.1. The Hard Reset Phase

This phase is responsible for initializing board-specific registers, such as memory base registers and DRAM controllers, or scanning memory to check the available size. It is written in assembler and ends with a jump to `_start`, which is where the assembly initialization phase begins.

The hard reset code is in a section named `.cs3.reset`. The section must define a symbol named `__cs3_reset_`*sys*, where *sys* is a name for the board being initialized; for example, the reset code for a Cyclone III Cortex-M1 board would be named `__cs3_reset_cycloneiii_cm1`. The linker script defines the symbol `__cs3_reset` to refer to this reset address. If you need to refer to the reset address from generic code, you can use this non-specific `__cs3_reset` name.

When using the Sourcery G++ Debug Sprite, the Sprite is responsible for carrying out the initialization done in this phase. In this case execution begins at the start of the assembly initialization phase, except for simulators as described below.

Some simulators provide a supervisory operation to determine the amount of available memory. This operation is performed in the hard reset phase. Thus for simulators, execution always begins at `__cs3_reset_`*sys*.

The CodeSourcery board support library provides reasonable default reset code, but you may provide your own reset code by defining `__cs3_reset_`*sys* in an object file or library, in a `.cs3.reset` section.

6.1.2. The Assembly Initialization Phase

This phase is responsible for initializing the stack pointer and creating an initial stack frame. It ends with a call or jump to `__cs3_start_c`. The symbol `_start` marks the entry point of the assembly initialization code.

The value of the symbol `__cs3_stack` provides the initial value of the stack pointer. The Code-Sourcery linker scripts provide a default value for this symbol, which you may override by defining `__cs3_stack` yourself.

Some processors initialize the stack pointer automatically on reset. However, because the assembly initialization phase is executed during debugging, it is required to set the stack pointer explicitly here in all cases.

The initial stack frame is created for the use of ordinary C and C++ calling conventions. The stack should be initialized so that backtraces stop cleanly at this point; this might entail zeroing a dynamic link pointer, or providing hand-written DWARF call frame information.

Finally, we call the C function `__cs3_start_c`. This function never returns, and `_start` need not be prepared to handle a return from it.

As with the hard reset code, the CodeSourcery board support library provides reasonable default assembly initialization code. However, you may provide your own code by providing a definition for `_start`, either in an object file or a library.

The symbol `_start` lacks the `__cs3` prefix, because many debuggers and integrated development environments assume that name is used for this purpose.

6.1.3. The C Initialization Phase

Finally, C code can be executed. The C startup function is declared as follows:

```
void __cs3_start_c (void) __attribute__ ((noreturn));
```

In this function we take the following steps:

- Initialize all `.data`-like sections by copying their contents.

- Clear all `.bss`-like sections.

- Run constructors for statically-allocated objects, recorded using whatever conventions are usual for C++ on the target architecture.

  CS3 reserves priorities from 0 to 100 for use by initialization code. You can handle tasks like enabling interrupts, initializing coprocessors, pointing control registers at interrupt vectors, and so on by defining constructors with appropriate priorities.

- Call `main` as appropriate.

- Call `exit`, if it is available.

As with the hard reset and assembly initialization code, the CodeSourcery board support library provides a reasonable definition for the `__cs3_start_c` function. You may override this by providing a definition for `__cs3_start_c`, either in an object file or in a library.

The CodeSourcery-provided definition of `__cs3_start_c` can pass command-line arguments to `main` using the normal C `argc` and `argv` mechanism if the board support package provides corresponding definitions for `__cs3_argc` and `__cs3_argv`. For example:

```
int __cs3_argc;
char **__cs3_argv;
```

These variables should be initialized using a constructor function, which is run by `__cs3_start_c` after it initializes the data segment. Use the `constructor` attribute on the function definition:

```
__attribute__((constructor))
static void __cs3_init_args (void) {
   __cs3_argc = ...;
   __cs3_argv = ...;
}
```

The constructor function may have an arbitrary name; `__cs3_init_args` is used only for illustrative purposes here.

If definitions of `__cs3_argc` and `__cs3_argv` are not provided, then the default `__cs3_start_c` function invokes `main` with zero as the `argc` argument and a null pointer as `argv`.

6.2. Exit and Embedded Systems

A program running on an embedded system is usually designed never to exit — if it needs to halt it may simply power down the system. The C and C++ standards leave it unspecified as to whether `exit` is called at program termination. If the program never exits, then there is no reason to include `exit`, facilities to run functions registered with `atexit`, or global destructors. This code would never be run and would therefore just waste space in the application.

The CS3 startup code, by itself, does not cause `exit` to be present in the application. It dynamically checks whether `exit` is present, and only calls it if it is. If you require `exit` to be present, either refer to it within your application, or add `-Wl,-u,exit` to the linking command line.

Similarly, code to register global destructors is only invoked when `atexit` is already in the executable; CS3, by itself, does not cause `atexit` to be present. If you require `atexit`, either refer to it within your application, or add `-Wl,-u,atexit` to the linking command line.

6.3. Memory Layout

The header file `cs3.h` declares variables and types that describe the layout of memory on the system to C code. The variables are defined by the CS3 linker script or in the board support library.

The following variables describe the regions of memory to be initialized at startup:

```
/* The number of elements in __cs3_regions.  */
const size_t __cs3_region_num;

/* An untyped object, aligned on an eight-byte boundary.  */
typedef unsigned char __cs3_byte_align8
                      __attribute__ ((aligned (8)));

struct __cs3_region
{
```

```
  /* Flags for this region.  None defined yet.  */
  unsigned flags;

  __cs3_byte_align8 *init;  /* Region's initial contents.  */
  __cs3_byte_align8 *data;  /* Region's start address.  */

  /* These sizes are always a multiple of eight.  */
  size_t init_size;     /* Size of initial data.  */
  size_t zero_size;     /* Additional size to be zeroed.  */
};

/* An array of memory regions.  __cs3_regions[0] describes
   the region holding .data and .bss.  */
extern const struct __cs3_region __cs3_regions[];
```

The following variables describe the area of memory to be used for the dynamically-allocated heap:

```
/* The addresses of these objects are the start and end of
   free space for the heap, typically following .data and .bss.
   However, &__cs3_heap_end may be zero, meaning that we must
   determine the heap limit in some other way --- perhaps via a
   supervisory operation on a simulator, or simply by treating
   the stack pointer as the limit.  */
extern __cs3_byte_align8 __cs3_heap_start[];
extern __cs3_byte_align8 __cs3_heap_end[];

/* The end of free space for the heap, or zero if we haven't been
   able to determine it.  It usually points to __cs3_heap_end,
   but in some configurations, may be overridden by a supervisory
   call in the reset code.  */
extern void *__cs3_heap_limit;
```

For each region named R, cs3.h declares the following variables:

```
/* The start of the region.  */
extern unsigned char __cs3_region_start_R[]
                  __attribute__ ((aligned (8)));

/* The region's size, in bytes.  */
extern const size_t __cs3_region_size_R;
```

At the assembly level, the linker script also defines symbols with the same names and values.

If the region is initialized, then cs3.h also declares the following variables, corresponding to the region's element in __cs3_regions:

```
/* The data we initialize the region with.  */
extern const unsigned char __cs3_region_init_R[]
                       __attribute__ ((aligned (8)));

/* The size of the initialized portion of the region.  */
extern const size_t __cs3_region_init_size_R;

/* The size of the additional portion to be zeroed.  */
extern const size_t __cs3_region_zero_size_R;
```

Any of these identifiers may actually be defined as preprocessor macros that expand to expressions of the appropriate type and value.

Like the `struct __cs3_region` members, these regions are all aligned on eight-byte boundaries, and their sizes are multiples of eight bytes.

CS3 linker scripts place the contents of sections named `.cs3.region-head.R` at the start of each memory region. Note that CS3 itself may want to place items (like interrupt vector tables) at these locations; if there is a conflict, CS3 raises an error at link time.

6.4. Interrupt Vectors and Handlers

CS3 provides standard handlers for interrupts, exceptions and traps, but also allows you to easily define your own handlers as needed. In this section, we use the term *interrupt* as a generic term for this entire class of events.

Different processors handle interrupts in various ways, but there are two general approaches:

- Some processors fetch an address from an array indexed by the interrupt number, and jump to that address. We call these *address vector* processors; Cortex-M1 systems are a typical example.

- Others multiply the interrupt number by some constant factor, add a base address, and jump directly to that address. Here, the interrupt vector consists of blocks of code, so we call these *code vector* processors; PowerPC systems are a typical example.

On address vector processors, the CS3 library provides an array of pointers to interrupt handlers named `__cs3_interrupt_vector_form`, occupying a section named `.cs3.interrupt_vector`, where *form* identifies the particular processor variant the vector is appropriate for. If the processor supports more than one variety of interrupt vector (for example, a full-length form and a shortened form), then *form* identifies the variety as well. Each entry in the vector holds a reference to a symbol named `__cs3_isr_int`, where *int* is the customary name of that interrupt on the processor, or a number if there is no consistently used name. The library further provides a reasonable default definition for each `__cs3_isr_int` handler routine.

To override an individual handler, provide your own definition for the appropriate `__cs3_isr_int` symbol. The definition need not be placed in any particular object file section.

Interrupt handlers may be written in C using the `interrupt` attribute. For example, to override the `__csr_isr_nmi` handler, use the following definition:

```
void __attribute__ ((interrupt)) __csr_isr_nmi (void)
{
  ... custom handler code ...
}
```

To override the entire interrupt vector, you can define `__cs3_interrupt_vector_form`, placing the definition in a section named `.cs3.interrupt_vector`. The linker script reports an error if the `.cs3.interrupt_vector` section is empty, to ensure that the definition of `__cs3_interrupt_vector_form` occupies the proper section.

You may define the vector in C with an array of pointers using the `section` attribute to place it in the appropriate section. For example, to override the interrupt vector on a Cyclone III Cortex-M1 board, make the following definition:

```
typedef void handler(void);
handler *__attribute__((section (".cs3.interrupt_vector")))
  __cs3_interrupt_vector_micro[] =
{ ... };
```

On code vector processors, we follow the same conventions, with the following exceptions:

- In addition to being named __cs3_isr_*int*, each interrupt handler must also occupy a section named .cs3.interrupt_*int*. Naturally, each handler must fit within a single interrupt vector entry.

- Instead of providing a default definition for __cs3_interrupt_vector_*form* in the library, the linker script gathers the .cs3.interrupt_*int* sections together, in the proper order and on the necessary address boundaries, and defines the __cs3_interrupt_vector_*form* symbol to refer to its start.

To override an individual handler on a code vector processor, you provide your own definition for __cs3_isr_*int*, placed in an appropriate section. The linker script ensures that each .cs3.interrupt_*int* section is non-empty, so that placing a handler in the wrong section elicits an error at link time.

CS3 does not allow you to override the entire interrupt vector on code vector processors, because the code vector must be constructed by the linker script, and thus cannot come from a library or object file. However, the portion of the linker script that constructs the interrupt vector occupies its own file, which other linker scripts can incorporate using the **INCLUDE** linker script command, making it easier to replace the linker script entirely and still take advantage of CS3's other features.

Some processors, like the Innovasic fido, use more than one interrupt vector: the processor provides several interrupt vector pointer registers, each used in different circumstances. Each register may point to a different vector, or some or all may share vectors.

On these processors, CS3 provides only a single pre-constructed interrupt vector, but defines a separate symbol for each interrupt vector pointer register; all the symbols point to the pre-constructed vector by default. The CS3 startup code initializes each register from the corresponding symbol. You can provide your own vectors by defining the appropriate symbols.

For example, the fido processor has five contexts, each of which can use its own interrupt vector; on this architecture, CS3 defines the standard __cs3_interrupt_vector_fido symbol referring to the pre-constructed vector, and then goes on to define per-context symbols __cs3_interrupt_vector_fido_ctx0, __cs3_interrupt_vector_fido_ctx1, and so on, all referring to __cs3_interrupt_vector_fido. The CS3 startup code sets each context's vector register to the value of the corresponding symbol. By default, all the contexts share an interrupt vector, but if your code provides its own definition for __cs3_interrupt_vector_fido_ctx1, then the startup code initializes context one's register to point to that vector instead.

This arrangement requires you to use a different approach to specify a handler for a secondary context that differs from the corresponding handler in the primary context. For example, to handle division-by-zero exceptions in context 1 with the function ctx1_divide_by_zero, you should write the following:

```
typedef void (*handler_type) (void);
handler_type __cs3_interrupt_vector_fido_ctx1[256];
extern handler_type __cs3_interrupt_vector_fido[256];
```

```
__attribute__((interrupt))
void
ctx1_divide_by_zero (void)
{
  /* Your code here.  */
}


__attribute__((constructor))
void
initialize_vector_ctx1 (void)
{
  /* Initialize our custom vector from the
     pre-constructed CS3 vector.  */
  memcpy (__cs3_interrupt_vector_fido_ctx1,
          __cs3_interrupt_vector_fido,
          sizeof (__cs3_interrupt_vector_fido));

  /* Initialize custom interrupt handlers.  */
  __cs3_interrupt_vector_fido_ctx1[5] = ctx1_divide_by_zero;
}
```

With this code in place, when a division-by-zero exception occurs in context 1, the processor calls `ctx1_divide_by_zero` to handle it. Defining `initialize_vector_ctx1` with the `constructor` attribute arranges for CS3 to call it before calling your program's `main` function.

6.5. Linker Scripts

CS3 provides linker scripts for each board that is supported. Each board may be used in a number of different configurations, and these are reflected in the linker script name. The linker scripts are named *board-profile-hosted*.ld, where *board* is the name of the board, *profile* describes the memory arrangement used and *-hosted* indicates whether semihosting is provided.

> **Caution**
>
> If you do not explicitly specify a linker script, Sourcery G++ produces a link error, preventing you from creating an executable program.

6.5.1. Program and Data Placement

Many boards have both RAM and ROM (flash) memory devices. CS3 provides distinct linker scripts to place the application either entirely in RAM, or in ROM where data is initialized during the C initialization phase.

Some boards have very small amounts of RAM memory. If you use large library functions (such as `printf` and `malloc`), you may overflow the available memory. You may need to use the ROM-based linker scripts for such programs, so that the program itself is stored in ROM. You may be able to reduce the total amount of memory used by your program by replacing portions of the Sourcery G++ runtime library and/or startup code.

6.5.2. Semihosting

CS3 is designed to support boards where there may be no operating system. To allow functions like `open` and `write` to work, a *semihosting* feature is supported, in conjunction with the debugger.

With semihosting enabled, these system calls are translated into equivalent function calls on your host system. You can only use these function calls while connected to the debugger; if you try to use them when disconnected from the debugger, you will get a hardware exception.

A good use of semihosting is to display debugging messages. For example, this program prints a message on the standard error stream on the host:

```
#include <unistd.h>

int main () {
  write (STDERR_FILENO, "Hello, world!\n", 14);
  return 0;
}
```

The hosted CS3 linker scripts provide the semihosting support, and as such programs linked with them may only be run with the debugger. The unhosted CS3 linker scripts provide stub versions of the system calls, which return an appropriate error value in `errno`. If such a stub system call is required in the executable, the linker also produces a warning. Such a warning may indicate that you have left debugging code active, and that your executable is larger than it might need to be.

6.5.3. Choosing a Linker Script from the Command Line

From the command line, you must add `-T script` to your linking command, where `script` is the appropriate linker script. For example, if you are using a Cyclone III Cortex-M1 board, you could link with `-T cycloneiii-cm1-ram-hosted.ld`.

6.6. Supported Boards for ARM EABI

The following linker scripts are provided for arm-none-eabi.

6.6.1. ARM M-profile Simulator

• Processor name: Cortex-M3

• Processor options: `-mcpu=cortex-m3 -mthumb`

• Memory regions: `ram`

Profile & Hosting	Linker Script	Reset name
Hosted	`generic-m-hosted.ld`	`__cs3_reset_generic_m`
Unhosted	`generic-m.ld`	`__cs3_reset_generic_m`

6.6.2. ARM Simulator (VFP)

• Processor name: ARM Simulator (VFP)

• Processor options: none

• Memory regions: `ram`

Profile & Hosting	Linker Script	Reset name
Hosted	`generic-vfp-hosted.ld`	`__cs3_reset_generic_vfp`
Unhosted	`generic-vfp.ld`	`__cs3_reset_generic_vfp`

6.6.3. ARM Simulator

- Processor name: ARM Simulator

- Processor options: none

- Memory regions: `ram`

Profile & Hosting	Linker Script	Reset name
Hosted	`generic-hosted.ld`	`__cs3_reset_generic`
Unhosted	`generic.ld`	`__cs3_reset_generic`

6.6.4. Cyclone III Cortex-M1

- Processor name: Cortex-M1

- Processor options: `-mcpu=cortex-m1 -mthumb`

- Memory regions: `ram`, `rom`, `itcm`

Profile & Hosting	Linker Script	Reset name
RAM Hosted	`cycloneiii-cm1-ram-hosted.ld`	`__cs3_reset_cycloneiii_cm1`
RAM Unhosted	`cycloneiii-cm1-ram.ld`	`__cs3_reset_cycloneiii_cm1`
ROM Hosted	`cycloneiii-cm1-rom-hosted.ld`	`__cs3_reset_cycloneiii_cm1`
ROM Unhosted	`cycloneiii-cm1-rom.ld`	`__cs3_reset_cycloneiii_cm1`

6.7. Interrupt Vector Tables

The ARM interrupt vector table (`__cs3_interrupt_vector_arm`) contents are:

Number	Name	Meaning
0	`__cs3_reset`	Reset entry point
1	`__cs3_isr_undef`	Undefined Instruction
2	`__cs3_isr_swi`	Software Interrupt/Supervisor Call
3	`__cs3_isr_pabort`	Prefetch Abort
4	`__cs3_isr_dabort`	Data Abort
5	`__cs3_isr_reserved`	
6	`__cs3_isr_irq`	External Interrupt (IRQ)
7	`__cs3_isr_fiq`	Fast Interrupt (FIQ)

The Microcontroller Profile interrupt vector table (`__cs3_interrupt_vector_micro`) contents are:

Number	Name	Meaning
0	`__cs3_stack`	Initial stack pointer
1	`__cs3_reset`	Reset entry point
2	`__cs3_isr_nmi`	Non Maskable Interrupt
3	`__cs3_isr_hard_fault`	Hardware fault
4	`__cs3_isr_mpu_fault`	MPU fault
5	`__cs3_isr_bus_fault`	Bus fault
6	`__cs3_isr_usage_fault`	Usage fault
7..10	`__cs3_isr_reserved_7..10`	Reserved for future use
11	`__cs3_isr_svcall`	System Vector Call
12	`__cs3_isr_debug`	Debug interrupt
13	`__cs3_isr_reserved_13`	Reserved for future use
14	`__cs3_isr_pendsv`	
15	`__cs3_isr_systick`	System Ticker
16..47	`__cs3_isr_external_0..31`	External interrupt

6.8. Regions and Memory Sections

The following regions are defined for ARM EABI.

Region	Contents
`itcm`	ITCM
`ram`	`.data` and `.bss` sections. In ram-based profiles, also contains `.text` and other program-like sections.
`rom`	`.text` and other program-like sections. Initialization values for data-like regions.

Note that not all regions are provided in every linker script or profile; see the documentation of the individual linker scripts in Section 6.6, "Supported Boards for ARM EABI", above.

Regions documented as "Memory regions" correspond to similarly-named program sections. For example, the linker script assigns the `.ram` section to the `ram` region. You can explicitly locate data or code in these sections using section attributes in your source C or C++ code. Section attributes are especially useful on code compiled for boards that include special memory banks, such as a fast on-chip cache memory, in addition to the default `ram` and/or `rom` regions. CS3 arranges for additional data-like sections to be initialized in the same way as the default `.data` section.

As an example to illustrate the attribute syntax, you can put a variable `v` in the `.ram` section using:

```
int v __attribute__ ((section (".ram")));
```

To declare a function `f` in this section, use:

```
int f (void) __attribute__ ((section (".ram"))) {...}
```

For more information about attribute syntax, see the GCC manual.

Regions documented as "Other regions" do not have a corresponding program section. Typically, these regions correspond to memory-mapped control and I/O registers that cannot be used for general

data or program storage. If you need to manipulate data in these regions, you can use the CS3 memory layout facilities declared in `cs3.h`, as described in Section 6.3, "Memory Layout".

Memory maps for boards supported by Sourcery G++ Lite for ARM EABI are documented in XML files in the `arm-none-eabi/lib/boards/` subdirectory of your Sourcery G++ installation directory.

Chapter 7
Sourcery G++ Debug Sprite

This chapter describes the use of the Sourcery G++ Debug Sprite for remote debugging. The Sprite allows you to debug programs running on a bare board without an operating system. This chapter includes information about the debugging devices and boards supported by the Sprite for ARM EABI.

Sourcery G++ Lite contains the Sourcery G++ Debug Sprite for ARM EABI. This Sprite is provided to allow debugging of programs running on a bare board. You can use the Sprite to debug a program when there is no operating system on the board, or for debugging the operating system itself. If the board is running an operating system, and you wish to debug a program running on that OS, you should use the facilities provided by the OS itself (for instance, using **gdbserver**).

The Sprite acts as an interface between GDB and external debug devices and libraries. Refer to Section 7.2, "Invoking Sourcery G++ Debug Sprite" for information about the specific devices supported by this version of Sourcery G++ Lite.

> **Important**
>
> The Sourcery G++ Debug Sprite is not part of the GNU Debugger and is not free or open-source software. You may use the Sourcery G++ Debug Sprite only with the GNU Debugger. You may not distribute the Sourcery G++ Debug Sprite to any third party. You may use the ARM SWD support (as used for debugging Luminary Micro Stellaris CPUs) only with target systems which contain Cortex-M1 or Cortex-M3 microprocessor managed under license from ARM.

7.1. Debug Sprite Example

This section demonstrates execution and debugging of a simple application. Start by creating a file named `fib.c`:

```c
#include <unistd.h>

static int Fib (unsigned n, unsigned a, unsigned b)
{
  unsigned count;

  for (count = 0; count != b; count++)
    write (1, ".", 1);
  write (1, "\n", 1);

  if (n)
    Fib (n - 1, b, a + b);
}

int main ()
{
  write (1, "Fibonacci\n", 10);
  Fib (9, 0, 1);
  return 0;
}
```

First compile and link the program for the target board. If it is a stand-alone program for a Cyclone III Cortex-M1 board use:

```
> arm-none-eabi-gcc -mcpu=cortex-m1 -mthumb \
  -Tcycloneiii-cm1-ram-hosted.ld fib.c -o fib -g
```

For other boards you must make appropriate substitutions in the preceding command. If your program is an operating system kernel such as uClinux or Linux, your usual build method should be adequate, as the kernel contains the necessary initialization code for interrupt handlers.

Verify that the Sourcery G++ Debug Sprite can detect your debug hardware:

```
> arm-none-eabi-sprite -i
```

This prints out a list of supported device types. For devices that can be autodetected, it additionally probes for and prints out a list of attached devices. For instance:

```
CodeSourcery ARM Debug Sprite
    (Sourcery G++ Lite Sourcery G++ Lite 2008q3-66)
rdi: (rdi-library=<file>&rdi-config=<file>) RDI Device
  rdi:/// - RDI Device
armusb: [speed=<n:0-7>] ARMUSB device
  armusb:/// - ARMUSB Device
```

This shows that RDI and ARMUSB devices are supported. The exact set of supported devices depends on your host system and the version of Sourcery G++ you have installed.

Start the debugger on your host system:

```
> arm-none-eabi-gdb fib
```

Connecting GDB to the board depends on the debug device you are using. If you are using a ARMUSB debug device, use:

```
(gdb) target remote | arm-none-eabi-sprite \
armusb:///?speed=2 lm3s8xx
Remote debugging using | arm-none-eabi-sprite \
armusb:///?speed=2 lm3s8xx
arm-none-eabi-sprite:Target reset
0x00008936 in ?? ()
```

If you are connecting via RDI, you must specify the full path to the RDI library file and configuration file for that library:

```
(gdb) target remote | arm-none-eabi-sprite \
"rdi:///?rdi-library=library&rdi-config=config"
Remote debugging using | arm-none-eabi-sprite \
"rdi:///?rdi-library=library&rdi-config=config"
ARMulator RVARMulatorISS1.4 [Build 297]
For support please contact support-sw@arm.com
Software supplied by: ARM Limited
ARM1136JF-S
ARM11 Instruction Set Simulator, May 24 2006
ARM Instruction Set Simulator for  [Build number 297]
, CP15, 8KB ICache, 8KB DCache 32KB DTCRam0 -Supports SmartCaching
32KB ITCRam0 -Supports SmartCaching , VFP11 (no support code), \
4GB, Pagetables, Mapfile, VIC - PL192
VIC: this is a RELEASE build
, Profiler, SIMRDI MemCallback, Tube, Millisecond [6666.67
cycles_per_millisecond], Tracer
Tracing: Instructions, Memory accesses, Events, Disassemble, \
Trace bus, Trace registers, Opcode Fetch Mask \
0x00000000-0x00000000, RDI Codesequences, Semihosting, \
CP14 Debug(6,2,2)
Little endian
```

```
arm-none-eabi-sprite:Missing config file; \
this may not work
arm-none-eabi-sprite:Target reset
0x00000000 in ?? ()
```

Refer to Section 7.2, "Invoking Sourcery G++ Debug Sprite" for more information about the supported devices in Sourcery G++ Lite and details about the command-line syntax for invoking the Sprite.

At this point you can use GDB to load your program onto the target board and control its execution as required:

```
(gdb) load
Loading section .text, size 0xaa0 lma 0x0
Loading section .ARM.exidx, size 0x8 lma 0xaa0
Loading section .data, size 0xfc lma 0xaa8
Start address 0x11, load size 2980
Transfer rate: 6231 bits/sec, 596 bytes/write.
```

Set a breakpoint so that the debugger stops when your program reaches main:

```
(gdb) break main
Breakpoint 1 at 0x20000524: file fib.c, line 17.
```

Allow the program to execute until it reaches main:

```
(gdb) continue
Continuing.
main () at fib.c:13
13          write (1, "Fibonacci\n", 10);
(gdb) next
Fibonacci
14          Fib (9, 0, 1);
```

Permit the program to finish executing with:

```
(gdb) continue
Continuing.
.
.
..
...
.....
........
.............
....................
................................
.......................................................

Program exited normally.
```

7.2. Invoking Sourcery G++ Debug Sprite

The Debug Sprite is invoked as follows:

```
arm-none-eabi-sprite [options] device-url board-file
```

The `device-url` specifies the debug device to use to communicate with the board. It follows the standard format:

```
scheme:scheme-specific-part[?device-options]
```

Most device URL schemes also follow the regular format:

```
scheme:[//hostname:[port]]/path[?device-options]
```

The meanings of `hostname`, `port`, `path` and `device-options` parts depend on the `scheme` and are described below. The following schemes are supported in Sourcery G++ Lite for ARM EABI:

rdi   Use an RDI debugging device. Refer to Section 7.4, "Remote Debug Interface Devices".

flashpro   Use a FlashPro debugging device. Refer to Section 7.5, "FlashPro Devices".

altera   Use an Altera FPGA. Refer to Section 7.6, "Altera Devices"

The optional `?device-options` portion is allowed in all schemes. These allow additional device-specific options of the form `name=value`. Multiple options are concatenated using `&`.

The `board-file` specifies an XML file that describes how to initialize the target board. If `board-file` refers to a file (via a relative or absolute pathname), it is read. Otherwise, `board-file` can be a board name, and the toolchain's board directory is searched for a matching file. See Section 7.8, "Supported Board Files" for the list of supported boards, or invoke the Sprite with the `-b` option to list the available board files. You can also write a custom board file; see Section 7.9, "Board File Syntax" for more information.

7.3. Sourcery G++ Debug Sprite Options

The following command-line options are supported by the Sourcery G++ Debug Sprite:

-b   Print a list of `board-file` files in the board config directory.

-h   Print a list of options and their meanings. A list of `device-url` syntaxes is also shown.

-i   Print a list of the accessible devices. If a `device-url` is also specified, only devices for that device type are scanned. Each supported device type is listed along with the options that can be appended to the `device-url`. For each discovered device, the `device-url` is printed along with a description of that device.

-l [host]:port   Specify the host address and port number to listen for a GDB connection. If this option is not given, the Debug Sprite communicates with GDB using stdin and stdout. If you start the Sprite from within GDB using the `target remote | arm-none-eabi-sprite ...` command, you do not need this option.

-m   Listen for multiple sequential connections. Normally the Debug Sprite terminates after the first connection from GDB terminates. This option instead makes it listen for a subsequent connection. To terminate the Sprite, open a connection and send the string `END\n`.

`-q`   Do not print any messages.

`-v`   Print additional messages.

If any of `-b`, `-i` or `-h` are given, the Debug Sprite terminates after providing the information rather than waiting for a debugger connection.

7.4. Remote Debug Interface Devices

Remote Debug Interface (RDI) devices are supported. The RDI device URL accepts no hostname, port or path components, so the *device-url* is specified as follows:

```
rdi:[///][?device-options]
```

The following *device-options* are required:

`rdi-library=`*library*   Specify the library (DLL or shared object) implementing the RDI target you wish to use.

`rdi-config=`*configfile*   Specify a file containing configuration information for *library*. The format of this file is specific to the RDI library you are using, but tends to constitute a list of *key=value* pairs. Consult the documentation of your RDI library for details.

7.5. FlashPro Devices

On Windows hosts, Sourcery G++ Lite supports FlashPro devices used with Actel Cortex-M1 development kits.

For FlashPro devices, the *device-url* has the following form:

```
flashpro:[//usb12345/][?jtagclock=rate]
```

The optional *usb12345* part indicates the ID of the FlashPro device to connect to, which is useful if you have more than one such device attached to your computer. If the ID is omitted, the Debug Sprite connects automatically to the first detected FlashPro device. You can enumerate the connected FlashPro devices by invoking the Sprite with the `-i` switch, as follows:

```
> arm-none-eabi-sprite -i flashpro:
```

The `jtagclock` option allows the communication speed with the target board to be altered. The *rate* is specified in Hz and may range between 93750 and 4000000. The default is 93750, the slowest speed supported by the FlashPro device. Depending on your target board, you may be able to increase this rate, but beware that communication errors may occur above a certain threshold. If you encounter communication errors with a higher-than-default speed selected, try reducing the speed.

7.5.1. Installing FlashPro Windows drivers

Windows drivers for the FlashPro device are included with the FlashPro software provided by Actel. Refer to Actel's documentation for details on installing this software. You must use the Actel FlashPro software to configure the FPGA on your Cortex-M1 board, but it does not need to be running when using the Debug Sprite.

Once you have set up your board using the FlashPro software, you can check that it is recognized by the Sourcery G++ Debug Sprite by running the following command:

```
> arm-none-eabi-sprite -i
flashpro: [jtagclock=<n:93750-4000000>] FlashPro device
  flashpro://usb12345/ - FlashPro Device
  ...
```

If output similar to the above does not appear, your FlashPro device is not working correctly. Contact CodeSourcery for further guidance in that case.

7.6. Altera Devices

The Debug Sprite can be used to debug applications running on a Cortex-M1 core embedded in an Altera FPGA supporting the System-Level Debug (SLD) architecture. Currently, the Sprite supports the Cyclone III FPGA Starter board on Microsoft Windows hosts.

The Debug Sprite accepts two forms of the *device-url* for Altera devices. For the common case where you have only one Altera Cortex-M1 device configured, you can use simply:

```
altera://
```

The full form of the *device-url* is:

```
altera://usbX/hubY/nodeZ
```

where X, Y, and Z are non-negative integers. The SLD architecture forms a hierarchy; there may be multiple USB Blaster devices (numbered by X), multiple Altera FPGAs (numbered by Y) per USB Blaster, and multiple nodes (numbered by Z) per FPGA.

The Debug Sprite can autodetect connected Altera Cortex-M1 devices. Invoking the Sprite with the -i option, as described in Section 7.3, "Sourcery G++ Debug Sprite Options", displays the *device-url* for each detected device:

```
arm-none-eabi-sprite -i
...
altera: Altera SLD Hub device
  altera://usb0/hub0/node1 - Altera Cortex-M Device
```

7.6.1. Setting Up the Altera Device

Follow these steps for initial installation and set up of the Altera device.

1. Install Quartus II Web Edition (or any equivalent), available from Altera.

2. Install drivers for USB Blaster, also available from Altera.

3. Install Sourcery G++ Lite for ARM EABI. See Chapter 4, *Installation and Configuration*.

4. Connect the board and the host computer with a USB cable.

5. Turn on the board.

6. Use Quartus II to download a .sof file including a Cortex-M1 core to the FPGA.

7. Use **arm-none-eabi-sprite -i** to verify that the Sprite can detect the installed Cortex-M1 core.

7.6.2. Hardware Breakpoints

The Cortex-M1 core only permits hardware breakpoints to be set in the first 512MB of its address space. Because both external SRAM and flash memory are located at higher addresses, you cannot set hardware breakpoints in these memory regions.

7.7. Debugging a Remote Board

You can run the Sourcery G++ Debug Sprite on a different machine from the one on which GDB is running. For example, if your board is connected to a machine in your lab, you can run the debugger on your laptop and connect to the remote board. The Sourcery G++ Debug Sprite must run on the machine that is connected to the target board.

To use this mode, you must start the Sprite with the `-l` option and specify the port on which you want it to listen. For example:

```
> arm-none-eabi-sprite -l :10000 device-url board-file
```

starts the Sprite listening on port 10000. Use the following command to connect GDB to the remote Sprite:

```
(gdb) target remote host:10000
```

to connect to the remote Sprite, where `host` is the name of the remote machine. After this, debugging is just as if you are debugging a target board connected to your host machine.

7.8. Supported Board Files

The Sourcery G++ Debug Sprite for ARM EABI includes support for the following target boards. Specify the appropriate `board-file` as an argument when invoking the sprite from the command line.

Board	Config
Cyclone III Cortex-M1	`cycloneiii-cm1`

7.9. Board File Syntax

The `board-file` can be a user-written XML file to describe a non-standard board. The Sourcery G++ Debug Sprite searches for board files in the `arm-none-eabi/lib/boards` directory in the installation. Refer to the files in that directory for examples.

The file's DTD is:

```
<!-- Board description files

     Copyright 2007, 2008 CodeSourcery.  All rights reserved.

     This file is licensed only for use with Sourcery G++.  No \
other use is
     permitted.
     -->

<!ELEMENT board
```

```
 (properties?, feature?, initialize?, memory-map?)>

<!ELEMENT properties
 (description?, property*)>

<!ELEMENT initialize
 (write-register | write-memory | delay
  | wait-until-memory-equal | wait-until-memory-not-equal)* >
<!ELEMENT write-register EMPTY>
<!ATTLIST write-register
         address CDATA   #REQUIRED
                         value   CDATA   #REQUIRED
                         bits    CDATA   #IMPLIED>
<!ELEMENT write-memory EMPTY>
<!ATTLIST write-memory
         address CDATA   #REQUIRED
                         value   CDATA   #REQUIRED
                         bits    CDATA   #IMPLIED>
<!ELEMENT delay EMPTY>
<!ATTLIST delay
         time CDATA   #REQUIRED>
<!ELEMENT wait-until-memory-equal EMPTY>
<!ATTLIST wait-until-memory-equal
         address CDATA   #REQUIRED
                         value   CDATA   #REQUIRED
                         timeout CDATA   #IMPLIED
                         bits    CDATA   #IMPLIED>
<!ELEMENT wait-until-memory-not-equal EMPTY>
<!ATTLIST wait-until-memory-not-equal
         address CDATA   #REQUIRED
                         value   CDATA   #REQUIRED
                         timeout CDATA   #IMPLIED
                         bits    CDATA   #IMPLIED>

<!ELEMENT memory-map (memory-device)*>
<!ELEMENT memory-device (property*,  description?)>
<!ATTLIST memory-device
                         address CDATA   #REQUIRED
         size    CDATA   #REQUIRED
         type    CDATA   #REQUIRED
                         device  CDATA   #IMPLIED>

<!ELEMENT description (#PCDATA)>
<!ELEMENT property (#PCDATA)>
<!ATTLIST property name CDATA #REQUIRED>

<!ENTITY % gdbtarget SYSTEM "gdb-target.dtd">
%gdbtarget;
```

All values can be provided in decimal, hex (with a `0x` prefix) or octal (with a `0` prefix). Addresses and memory sizes can use a `K`, `KB`, `M`, `MB`, `G` or `GB` suffix to denote a unit of memory. Times must use a `ms` or `us` suffix.

The following elements are available:

`<board>`	This top level element encapsulates the entire description of the board. It can contain `<properties>`, `<features>`, `<initialize>` and `<memory-map>` elements.
`<properties>`	The `<properties>` element specifies specific properties of the target system. This element can occur at most once. It can contain a `<description>` element. It can also contain the following `<property>` elements:

`<banked-regs>`	The `<banked-regs>` element specifies that the CPU of the target board has banked registers for different processor modes (supervisor, IRQ, etc.).
`<has-vfp>`	The `<has-vfp>` element specifies that the CPU of the target board has VFP registers.
`<system-v6-m>`	The `<system-v6-m>` element specifies that the CPU of the target board has ARMv6-M architecture system registers.
`<system-v7-m>`	The `<system-v7-m>` element specifies that the CPU of the target board has ARMv7-M architecture system registers.

`<initialize>`	The `<initialize>` element allows board devices to be initialized before any attempt is made to download a program to it. It can contain `<write-register>`, `<write-memory>` and `<delay>` elements.
`<feature>`	This element is used to inform GDB about additional registers and peripherals available on the board. It is passed directly to GDB; see the GDB manual for further details.
`<memory-map>`	This element describes the memory map of the target board. It is used by GDB to determine where software breakpoints may be used and when flash programming sequences must be used. This element can occur at most once. It can contain `<memory-device>` elements.
`<memory-device>`	This element specifies a region of memory. It has four attributes: `address`, `size`, `type` and `device`. The `address` and `size` attributes specify the location of the memory device. The `type` attribute specifies that device as `ram`, `rom` or `flash`. The `device` attribute is required for `flash` regions; it specifies the flash device type. The `<memory-device>` element can contain a `<description>` element.
`<write-register>`	This element writes a value to a control register. It has three attributes: `address`, `value` and `bits`. The `bits` attribute is optional and defaults to 32.
`<write-memory>`	This element writes a value to a memory location. It has three attributes: `address`, `value` and `bits`. The `bits` attribute is optional and defaults to 32. Bit widths of 8, 16 and 32 bits are supported. The address written to must be naturally aligned for the size of the write being done.
`<delay>`	This element introduces a delay. It has one attribute, `time`, which specifies the number of milliseconds, or microseconds to delay by.

<description>   This element encapsulates a human-readable description of its enclosing element.

<property>   The <property> element allows additional name/value pairs to be specified. The property name is specified in a name attribute. The property value is the body of the <property> element.

Chapter 8
Next Steps with Sourcery G++

This chapter describes where you can find additional documentation and information about using Sourcery G++ Lite and its components.

8.1. Sourcery G++ Knowledge Base

The Sourcery G++ Knowledge Base is available to registered users at the Sourcery G++ Portal[1]. Here you can find solutions to common problems including installing Sourcery G++, making it work with specific targets, and interoperability with third-party libraries. There are also additional example programs and tips for making the most effective use of the toolchain and for solving problems commonly encountered during debugging. The Knowledge Base is updated frequently with additional entries based on inquiries and feedback from customers.

For more information on CodeSourcery support, see Chapter 2, *Sourcery G++ Subscriptions*.

8.2. Manuals for GNU Toolchain Components

Sourcery G++ Lite includes the full user manuals for each of the GNU toolchain components, such as the compiler, linker, assembler, and debugger. Most of the manuals include tutorial material for new users as well as serving as a complete reference for command-line options, supported extensions, and the like.

When you install Sourcery G++ Lite, links to both the PDF and HTML versions of the manuals are created in the shortcuts folder you select. If you elected not to create shortcuts when installing Sourcery G++ Lite, the documentation can be found in the `share/doc/sourceryg++-arm-none-eabi/` subdirectory of your installation directory.

In addition to the detailed reference manuals, Sourcery G++ Lite includes a Unix-style manual page for each toolchain component. You can view these by invoking the **man** command with the pathname of the file you want to view. For example, you can first go to the directory containing the man pages:

```
> cd $INSTALL/share/doc/sourceryg++-arm-none-eabi/man/man1
```

Then you can invoke **man** as:

```
> man ./arm-none-eabi-gcc.1
```

Alternatively, if you use **man** regularly, you'll probably find it more convenient to add the directory containing the Sourcery G++ man pages to your `MANPATH` environment variable. This should go in your `.profile` or equivalent shell startup file; see Section 4.7, "Setting up the Environment" for instructions. Then you can invoke **man** with just the command name rather than a pathname.

Finally, note that every command-line utility program included with Sourcery G++ Lite can be invoked with a `--help` option. This prints a brief description of the arguments and options to the program and exits without doing further processing.

---

[1] https://support.codesourcery.com/GNUToolchain/