

GNU TOOLCHAIN FOR EMBEDDED DEVELOPMENT: BUILD OR BUY?

MARK MITCHELL, DIRECTOR OF OPEN SOURCE TOOLS, MENTOR GRAPHICS



E M B E D D E D

S O F T W A R E

W H I T E P A P E R

www.mentor.com

INTRODUCTION

Increasingly, embedded software developers are choosing the GNU toolchain for open source development. The GNU toolchain contains an optimizing compiler targeting most embedded processors. The toolchain also supports programming in C, C++, assembly language, and compiler and linker extensions which are specifically designed to assist embedded programmers. Further, support for multiple target platforms makes porting code between processors simpler since developers can use the same tools on multiple platforms.

Mentor® Embedded Sourcery™ CodeBench, a complete, reliable, and convenient GNU toolchain offers today's software developers all of these advantages.


Of course, the core components of the toolchain (an IDE, C compiler, C++ compiler, assembler, linker, debugger, and other tools) are available as open source software, so developers have the option of building the toolchain themselves. But deciding to use the GNU toolchain, one has to make a "build vs. buy" decision: should you build all the components yourself, or should you buy a pre-built toolchain like Sourcery CodeBench?

This whitepaper presents some of the technical issues involved in building and validating the toolchain. By considering these issues, developers will be better able to decide whether they can commit the resources required to "do it yourself."


"Toolchain Components" contains a review of the various components of the toolchain.

"Building the Toolchain" outlines the toolchain build process, and explains the importance of automating the process to provide reliability.

"Validating the Toolchain" describes tools and techniques for validating the toolchain.

 **NOTE: A flag icon indicates an important decision. If you're seriously considering building your own toolchain, you'll need to answer these critical questions.**

TOOLCHAIN COMPONENTS

 **Which tools and libraries will you include in your GNU toolchain?**

The GNU toolchain is more than just a compiler. For embedded software developers, a complete toolchain generally includes:

- C/C++ Compilers
- Assembler
- Linker
- Runtime Libraries
- Debugger
- Debug Stub(s)
- Integrated Development Environment

This section discusses each of the components listed above. Other options include profiling tools, a flash programmer, or a simulator (which are not described in this section).

GNU C/C++ COMPILERS

The GNU C and GNU C++ compilers (GCC and G++) are the heart of the GNU toolchain. The compilers transform source code written in C or C++ into assembly code for your target processor. The GNU C and C++ compilers support most popular CPUs and operating systems.

Do you develop both C and C++ code?

If you are confident that you will not need to use C++, you can save yourself time and effort by not building the C++ compiler. Although G++ works on virtually all systems supported by GCC, it can be difficult to build the GNU C++ runtime library for some target operating systems. Some C++ features, like exceptions or automatic initialization of global objects, require additional configuration and need to be validated carefully.

GNU BINARY UTILITIES

The GNU binary utilities include the GNU assembler, GNU linker, and tools that can display the contents of object files and convert object files from one format to another. Some of these tools are invoked automatically by the compiler driver in the process of building an application; you will use others directly in creating, debugging, and deploying your application.

RUNTIME LIBRARIES

Several runtime libraries are required to build complete applications. These include:

- Compiler Support Library

The `libgcc` library provided by the compiler itself supplies low-level compiler support routines such as software floating-point emulation and support for exception-handling. This library is used by virtually all programs compiled with GCC.

Which C runtime library is appropriate for your target system?

- C Library

The C runtime library contains ISO C functions, like `printf` and `memcpy`. The three most popular C libraries used with the GNU toolchain are the GNU C Library (GLIBC), `uClibc`, and `Newlib`.

GLIBC is a full-featured, POSIX-compliant C library designed for use with the Linux kernel. In addition to those functions required by ISO C, it includes functions required by POSIX, functions provided in other UNIX C libraries, and GNU extensions.

`uClibc` is a smaller footprint library (also designed for use with the Linux® kernel) that contains a subset of the GLIBC functionality. You can use `uClibc` on either GNU/Linux or `uClinux`.

`Newlib`, generally used on bare-metal targets, has an even smaller footprint, but provides significantly reduced functionality.

If your system already has a C library, you may not need to build a C library at all—but you may need to modify GCC or the C++ runtime library to work with your C library.


- C++ Library

The C++ runtime library includes classes required by the ISO C++ standard, including `std::ostream` and `std::vector`. There is only one C++ runtime library in widespread use with the GNU toolchain. The GNU C++ Library (also called `libstdc++`) is provided in the same source package as the GNU C and C++ compilers.

GNU DEBUGGER

The GNU Debugger (GDB) provides both source- and assembly-level debugging. GDB allows you to view registers on your target, connect to a running system, set both software and hardware breakpoints and watchpoints, step through your application, disassemble code, and modify data on the target system. You can use GDB directly (from the command line) or as the “back end” for a graphical environment, such as Eclipse.

DEBUG STUB

 ***How will the GNU debugger communicate with your target system? Does your ICE or JTAG unit work with GDB?***

 ***How will you program Flash memory?***

In an embedded environment it's generally not practical to run the debugger directly on the target system. Instead, the debugger runs on the host system and communicates with the target via a “stub” over an ICE or JTAG unit, TCP/IP, or serial connection. The stub acts as an intermediary between GDB and the target, allowing GDB to start and stop programs, look at the contents of registers, and perform other necessary functions on the target system.


GDB, as distributed by the Free Software Foundation, does not contain any stubs that can be used with ICE or JTAG units, but it does provide a TCP/IP protocol for communicating with third party stubs. Stubs are available from some JTAG and board vendors, either in firmware or as part of their software kits, and from toolchain distributors, including Mentor Embedded. You may also write your own stub using the documentation in the GDB manual.

Without a full-featured stub, you cannot take advantage of all of GDB's features for debugging applications on your actual target hardware. You should look for a stub with support for all of the following features:

- Burning Flash memory
- Semihosting (i.e., permitting target applications to read and write files on the host system)
- Displaying extended registers (such as control registers, memory-mapped I/O registers, etc.)
- Setting both hardware and software breakpoints and watchpoints
- Operation on both Microsoft Windows and GNU/Linux

Mentor Embedded's debug stubs support burning Flash memory, setting hardware breakpoints, and many other features. Mentor Embedded has developed a GNU Debugger Interface Library (GDIL) to manage the connection interface between the GNU Debugger and the stub. GDIL contains thousands of lines of code and is used on a variety of host and target systems.

ECLIPSE IDE

 ***Are you willing to work exclusively from the command-line? Or would you prefer a graphical development environment?***

Eclipse is the integrated development environment of choice for use with the GNU toolchain. You should include the Eclipse IDE in your toolchain unless you are comfortable developing entirely from the command line.

Eclipse contains editors for writing code, a class hierarchy viewer, and support for debugging. If Eclipse, GDB, and a GDB stub are properly integrated, developers can edit, compile, link, and debug applications running from Flash memory on an embedded target board without ever leaving Eclipse. In addition, a large number of third-party Eclipse plug-ins are available that go beyond the functionality provided by the base IDE.

Mentor Embedded plug-ins seamlessly integrate Sourcery CodeBench with Eclipse. For example, you can select the target board from a drop-down menu. When you want to debug an application program on a system using Flash memory, Mentor Embedded's plug-ins automatically burn the image into Flash before running the program in the debugger.


BUILDING THE TOOLCHAIN

Building a complete GNU toolchain is a complex process. You must:

- Create your build environment
- Obtain the source code
- Make decisions about configuration options
- Understand and perform the build process
- Package the toolchain

The remainder of this section examines each of these steps in more detail.

BUILD ENVIRONMENT

 *Do you have the right tools to build your toolchain?*

Mentor Embedded's internal build process is fully automated. A single command configures, builds, and packages the toolchain. Mentor Embedded's build system also archives the source code used to build the tools and produces a complete log of every command executed during the build process so that the same toolchain can be rebuilt in future, with any necessary improvements. Mentor Embedded's build system contains more than 20,000 lines of code.

In order to build your GNU toolchain you'll need tools, of course. And these "build tools" (i.e., the compilers, libraries, and related tools that you have installed on your system) can influence the quality of the resulting toolchain. For example, it is most convenient to build the toolchain using another version of GCC—but if the version of GCC you are using to build your toolchain has defects, then it may incorrectly compile the toolchain.

Because the libraries on your build system will be linked into the toolchain, your toolchain will only run on machines compatible with yours. So, you must be careful to build on a system that is the "lowest common denominator" of the set of systems you wish to support.

Do you want to use “Canadian cross” compilers to build toolchains for multiple host systems?

If you need to support multiple host operating systems (i.e., if your toolchain will be used on both Microsoft Windows and GNU/Linux), then you must either (a) set up multiple build environments, or (b) build “Canadian cross” toolchains.

A Canadian cross toolchain is a toolchain that is built on one system, runs on a second system, and generates code for a third system. For example, you might use a GNU/Linux build system to build a toolchain that will run on Microsoft Windows and generate code for an embedded target. The advantage to building a Canadian cross compiler is that you can then do all of your builds on a single build system. However, in order to build a Canadian cross toolchain, you must first build an ordinary cross toolchain (e.g., from GNU/Linux to Microsoft Windows) and then use that cross toolchain to build the Canadian cross toolchain.

Mentor Embedded builds Sourcery CodeBench on a GNU/Linux system. Canadian cross compilers are used to build toolchains for use on other systems. Mentor Embedded’s internal build environment includes runtime libraries for GNU/Linux, Microsoft Windows, and Solaris that have been selected to provide maximum compatibility across all versions of these operating systems.

If you need to apply a patch and rebuild the toolchain, how will you ensure that you have not changed the build environment?

If, after deploying your toolchain, you find a defect that requires a rebuild, you will need to perform the entire build process again. Therefore, you should archive all of the source code for the toolchain, the build tools used to build the toolchain, and the commands you used to build the toolchain.

On Windows, will you use Cygwin or the Win32 API?

If your toolchain will run on a Microsoft Windows host system, you must decide whether you want to use the Win32 API or the Cygwin UNIX-emulation environment. A Cygwin toolchain will behave somewhat differently from a Win32 toolchain. In particular, a Cygwin toolchain will use UNIX-style path names, while a Win32 toolchain will use Windows path names.

Because the toolchain was originally developed to run on UNIX-like systems, it is easier to build the toolchain to depend on Cygwin. In addition, a Cygwin toolchain is convenient if you plan to build GNU/Linux software for your target system.

On the other hand, many Windows tools do not understand UNIX-style paths, so it may be challenging to integrate your toolchain into the rest of your environment. In addition, Cygwin imposes some performance overhead relative to the Win32 API. Finally, all Cygwin binaries require a copy of the Cygwin DLL. If you provide this DLL yourself, it may conflict with other copies of the DLL that users of your toolchain already have. On the other hand, if you do not provide this DLL, users of your toolchain will have to download it themselves.

Sourcery CodeBench uses the Win32 API and therefore does not depend on Cygwin. However, Mentor Embedded’s toolchains understand Cygwin’s UNIX-style paths and symbolic links, so you can use Sourcery CodeBench to build a Linux kernel or other GNU/Linux applications for your target system.

SOURCE CODE AND PATCHES

What versions of the various toolchain components will you use?

Before you can build the toolchain, you must obtain all the relevant source code. Your first step should be to determine which version of each component will suit your needs. You should ensure that the version you select contains support for your target hardware and that it is compatible (at the source and binary level) with any software that you will be incorporating into your project.

For example, some source code may rely on language features that were not supported in older versions of GCC. Other source code may contain errors that prevent compilation by the very newest (and most ISO-compliant) versions of GCC. If you are not sure what version to use, you should generally use the most recent release.

You should remember that in the free and open source software communities, version numbers do not mean the same thing that they do in other environments. For a proprietary software package, a particular version number or build number indicates the *binary* in use. In contrast, a version of GCC (such as 4.1.1) only denotes a version of the *source* code. The binary code generated from the source code depends on the configuration options selected and the build environment used.

What patches should be applied to the toolchain?

While there is a single FSF source release of GCC (or any other toolchain component) with a given version number, there are myriad patches to that release available from various locations.

Between FSF releases, Mentor Embedded and others in the open-source community make substantial enhancements to add support for more hardware architectures, improve code generation, and eliminate defects.

Some of these changes may be useful for your toolchain, and some may not. Some patches may introduce new problems of conflict with other patches. You must evaluate which of these patches you wish to apply to the base source version.

Unfortunately, there is no way to know whether you have all the patches that you need, and some important patches may not be easy to find. It often takes months for a patch to make its way into the FSF source repositories. Some patches are available from public mailing list archives or defect-tracking systems. In other cases, the patches may only be available to direct customers of a toolchain distributor.

Mentor Embedded works with its partners and customers (including many of the most popular semiconductor and operating system vendors) to include optimizations or support for new processors, operating systems, and target boards in its toolchains. Sourcery CodeBench also contains patches for problems revealed by its validation process and for problems reported by customers.


CONFIGURATION OPTIONS

How should you configure the various toolchain components?

Most toolchain components contain a “configure script” which you must use to select the target platform and to specify which of the available features will be included in your toolchain. Taken together, these configure scripts have thousands of options, covering everything from the locations in which components should be

installed to whether or not the `_cxa_atexit` routine should be used to register destructors for C++ objects with static storage duration.

Some configuration options are purely a matter of preference, some affect performance but not correctness, and still others have a definitive right answer on particular systems. After building your GNU toolchain, you should validate it to ensure that you have picked options that work correctly in your target environment.

 ***What CPUs will be used on your target systems? Will both big-endian and little-endian code be required?***

The default runtime libraries provided for a given architecture may not be appropriate for your target environment. For example, the default runtime libraries for a “bare-metal” ARM environment are optimized for a little-endian ARM V5 processor. If you have an ARM V4 processor, or a big-endian XScale processor, these libraries will not work on your system. On the other hand, if you have an ARM V6 processor, these runtime libraries will not take full advantage of your hardware. To adjust the set of runtime libraries present in your toolchain, you will often need to make modifications to the source code for GCC. You must also build the C library multiple times, once for each of the configurations you require.

BUILD PROCESS

The build process is itself complex, especially for a GNU/Linux toolchain. While the GNU coding standards suggest that all packages should be built with a uniform “configure, make, make install” sequence, there are variations on that theme in the various toolchain components. In general, it is necessary to provide additional options to each of the three phases in order to obtain optimal results.

There is an interdependency between the compiler and the C library. In particular, GCC examines the C library to determine certain configuration parameters, but, of course, you cannot build the C library until you have a compiler.

The Eclipse IDE (which is written in Java) uses a build process entirely different from that used by most of the other toolchain components. The Eclipse web site contains additional information about configuring and building the IDE.

Mentor Embedded builds the C compiler as many as three times, depending on the configuration. The runtime libraries shipped with Mentor Embedded’s packages are built with the final compiler. Mentor Embedded builds plug-ins for Eclipse from the command-line, rather than from within Eclipse itself, so that the plug-in build process can be fully automated.

PACKAGING

 ***Are there standard package formats for your host operating system(s)?***

 ***Would a graphical installation process make it easier to install your toolchain?***

Once you have built your toolchain, you will want to package it for easy installation throughout your organization. On Microsoft Windows, a graphical installer makes it easier for users to install the toolchain, place the tools in the PATH, and to manage uninstallation. On GNU/Linux, you may want support for Red Hat Package Manager (RPM) packages, which are used on Red Hat Enterprise Linux, SuSE Enterprise Linux, and other popular GNU/Linux distributions.

In designing your packaging scheme, you should consider the possibility that users will install multiple toolchains. It's useful to share certain components (such as Eclipse) so that users do not have to set up their preferences for each toolchain. But, if files are shared between installations, then you must consider the impact of upgrading one toolchain independently of the others or uninstalling a single toolchain.

Once you have selected your packaging software and layout scheme, you must build the installers. You should automate that process so that you can easily rebuild your installation packages whenever you rebuild your toolchains.

Mentor Embedded builds its graphical installers using industry-standard installation software. With just a few clicks, users can install Sourcery CodeBench on either Windows or GNU/Linux host systems. Since each toolchain includes different files, a single wizard-generated installation script is insufficient. Instead, Mentor Embedded's internal build software automatically generates installation scripts based on the files present after the toolchain has been built. Mentor Embedded has separate automated facilities for generating packages in other formats, including the Red Hat Package Manager (RPM) format.

VALIDATING THE TOOLCHAIN

Will your toolchain work?

Having built the toolchain, you should validate it before deploying it in a production environment. In general, the following techniques are used to test a compiler (and related tools):

- **Compiling programs and running those programs in the target environment.** This technique verifies that the generated code behaves as intended. You can also use this approach to test the performance of the code generated by the toolchain.
- **Compiling programs and inspecting the generated object file or executable image, without running the generated code.** This technique can be used to check that the generated code contains expected machine instructions, correct debugging information, conforms to a specified Application Binary Interface, etc.
- **Compiling fragments of a program with multiple compilers and linking the fragments together.** This technique checks that the compiler you wish to validate interoperates with a known-good compiler for the target platform.
- **Compiling invalid program fragments and checking for appropriate error messages.** This technique (called "negative testing") checks that the compiler is correctly enforcing constraints.

There are a number of additional techniques that you should apply to other components of the toolchain, including:

- **Interactive testing of the debugger.** You should compile and debug programs using the debugger with your target system to ensure that all debugger functionality works as expected.
- **Interactive testing of the IDE.** Like debuggers, IDEs are, by their nature, interactive. You should check that all desired IDE functionality (including, in particular, integration with GDB, the GDB stub, and the target system) works as intended.

A variety of testsuites are available to perform the various validation steps described above. The following sections provide a brief overview of some of the most useful testsuites.

GNU REGRESSION TESTSUITES

Does your toolchain meet basic correctness requirements? Do GNU source extensions work correctly?

Many of the GNU toolchain components (including the GNU Compiler Collection, GNU Binary Utilities, and GNU Debugger) include testsuites based on the DejaGNU framework. Every GNU toolchain build should be validated using these testsuites because only the DejaGNU testsuites provide test coverage for GNU extensions to the C and C++ programming languages and the wide variety of features available in other tools. The DejaGNU GDB testsuite performs live tests of the debugger on a running target system to ensure appropriate interactive behavior. Taken together, the DejaGNU testsuites contain tens of thousands of tests, and are usually expanded to include new tests whenever a defect is corrected or a new feature is added.

To use each of these testsuites, you must first develop a DejaGNU board configuration file for your target system. The board configuration file will contain code to perform a variety of basic operations, including, most critically:

- **Running programs on your target system.** This code must upload the program to the target system (or make it available via a network file system), execute the program, and report the results.
- **Rebooting your target system.** Many embedded systems lack memory protection. Therefore, when a test fails, the target system is likely corrupted, and the system must be rebooted. You may require specialized hardware to manage automatic rebooting, such as managed power strips.

The board configuration is written in the Expect programming language, which is an extension to the Tcl programming language. Unfortunately, documentation for DejaGNU is very sparse, so you will likely need to make use of existing board configurations as a starting point.

- TESTING INSTALLED TOOLCHAINS

The DejaGNU testsuites have customarily been run from the build directories in which the toolchain was built. However, testing in this manner requires invoking the tools in a substantially different way from that in which users will actually invoke them. More accurate results can be obtained by installing the toolchain first and then running the tests. Testing installed components ensures that the tools tested are the same binaries, invoked in the same way, as they will be by users.


However, testing installed toolchains is more complex than testing from the build directory. In particular, you must create a DejaGNU “site file” to describe your installation. You may also find that some of the DejaGNU testsuites require modification to support testing installed toolchains. If you wish to test installed toolchains, you may find it helpful to automate both the installation process and the generation of appropriate DejaGNU site files so that you can easily reproduce your testing.

- TESTING THE GNU C LIBRARY

The GLIBC testsuite does not yet make use of the DejaGNU framework. However, it does contain a set of tests that you should run. These tests cover much of the functionality provided in GLIBC, including, in particular tests for the Native POSIX Threads Library (NPTL), which provides high-performance threading support on GNU/Linux systems. These tests provide a powerful mechanism for testing the compiler, C library, and kernel, all of which must cooperate to provide support for threads. Because the GNU C Library testsuite does not support cross-testing (i.e., compiling the tests on one system and testing on another), you will have to modify the testsuite to support testing on embedded systems.

Mentor Embedded uses the DejaGNU regression testsuites to test all versions of Sourcery CodeBench. Mentor Embedded installs its toolchains before testing to ensure that its testing is as accurate as possible. For toolchains with multiple runtime libraries, Mentor Embedded runs these tests multiple times, with varying compilation options, to exercise all of the runtime libraries.

CONFORMANCE TESTSUITES

 ***Does your toolchain meet the requirements of relevant standards and specifications? Will standard-compliant source code be processed correctly?***

You should use conformance testsuites to validate the behavior of your toolchain relative to published specifications. You may wish to seek out additional conformance testsuites to validate functionality specific to your intended use of the compiler.


For example, the Plum-Hall C and C++ Validation Suites are comprehensive tests for conformance to the C and C++ programming language specifications. The Plum-Hall Testsuites contain tests for nearly every sentence of the published specifications, including both the programming languages proper and the associated runtime libraries. In addition, the Plum-Hall Testsuite can automatically generate a number of “expression tests” which contain complex arithmetic expressions. These expression tests have proven useful in identifying instances of incorrect code generation.

The OpenPOSIX Testsuite checks conformance of a compiler, C library, and operating system to the POSIX specification. This testsuite runs C programs that make heavy use of the C library and checks that the results returned by the library routines are correct. Mentor Embedded has frequently discovered problems in the C library and/or operating system kernel through the use of this testsuite.

Mentor Embedded’s C++ ABI Testsuite is a comprehensive test of the industry-standard C++ Application Binary Interface used by G++ and by proprietary compilers from ARM, Intel, and a variety of other providers. This testsuite checks for correct object layout, name mangling, virtual table layout, and other issues. It is not designed to check for language conformance. Instead, the C++ ABI Testsuite checks for issues that will prevent interoperation between G++ and other compilers (including other versions of G++ itself).

Mentor Embedded uses the Plum-Hall, C++ ABI, OpenPOSIX, and other testsuites to check for standards conformance.

PERFORMANCE TESTSUITES

 ***Does your toolchain meet performance requirements? Will generated code run as quickly as possible?***

Performance testsuites are designed to provide data about the speed at which the code generated by the toolchain executes. These testsuites can help to identify misconfigurations of the toolchain, such as situations in which the generated code is using software floating-point on a system that supports hardware floating-point. They are also useful in comparing newer versions of the toolchain with older versions. For example, before deploying an upgraded version of the toolchain for production use, you might wish to ensure that the code generated is in fact better (or, at least, no worse) than that generated by the current toolchain.

The EEMBC benchmarks are widely used as measurements of embedded system performance. These benchmarks are divided by application areas; for example, there are EEMBC benchmarks for networking applications, automotive applications, and for office automation.

Mentor Embedded uses EEMBC, SPEC, and other benchmarks to measure – and improve – performance. Mentor Embedded also uses these benchmarks to ensure that the performance of its toolchains does not degrade over time.

The SPEC CPU benchmarks are highly-regarded benchmarks for C, C++, and Fortran applications. Some of these benchmarks are “scientific” code while others focus on general-purpose computing.

TESTING MULTIPLE OPTIONS

🔗 *Has your toolchain been tested in all the ways it will be used?*

Once you have validated the compiler using a single set of options, you should expand your “validation matrix.” There are three important dimensions to the validation matrix:

- **Target System**
- **Level of Optimization**
- **Host System**

Target system options include the choice of target CPU and operating system, whether to compile big-endian or little-endian code, and other related options which specify the system on which the code generated by the toolchain will execute. It is not at all uncommon for the GNU toolchain to behave correctly for one target system, but not for a seemingly related system. For example, even though little-endian code works, big-endian code may not. These problems are especially likely if you have incorporated patches from hardware manufacturers designed to support a particular CPU or CPU family, as these vendors may well not have tested other configurations. If you intend to use your version of GCC with multiple targets, you should validate each target independently.

Optimization options include whether to generate code best-suited to debugging, code designed to run quickly, or code optimized for size. Different optimization options exercise different code paths in the compiler and can therefore have a substantial impact on test results. The three compilation modes used most often in development are: (a) debug (-g); (b) optimized for time (-O2); and (c) optimized for space (-Os). You should ensure that your testing exercises all of these operating modes.

Finally, you should validate the toolchain on all host systems (Microsoft Windows, GNU/Linux, etc.) on which you will be using the toolchain. In addition to checking that the generated code is correct, you should verify that the code generated is in fact identical on all host systems. Because most GNU toolchain developers use IA32 GNU/Linux systems for their own development, host support for Microsoft Windows has tended to be particularly problematic. For example, some versions of GCC have made incorrect assumptions about the behavior of the Windows C library that resulted in the generation of incorrect assembly code. Similarly, reliance on pointer values as hash-table keys has resulted in different code generated on different hosts.

ANALYZING & CORRECTING FAILURES

🔗 *Do failing tests indicate serious problems?*

Having gathered data about which tests pass and which fail, you must now evaluate the results. The GNU toolchain (like all toolchains) has defects. Therefore, in evaluating the toolchain you have built, you should attempt to determine whether or not the failures that you observe are unique to your toolchain. If the failures you encounter are not present in other builds of similar toolchains, then you may have built the toolchain incorrectly. However, even if the failures are not specific to your toolchain, you must evaluate whether or not they are sufficiently severe as to impede use of your toolchain.

Failing tests can be divided into the following categories:

- **Defects in the tests themselves.** This category includes tests that make incorrect assumptions about the target environment, such as tests that assume the target is a little-endian machine, has 32-bit pointers,

or treats plain char as a signed data type. These tests should be corrected. If the tests cannot be readily corrected, these failures should be ignored.

- **Defects in the hardware platform.** In some cases, testing the toolchain reveals microprocessor defects (such as incorrect handling of instructions in delay slots). In other cases, the target board may contain faulty parts or may use faulty software. It may be necessary to implement toolchain work-arounds for some of these problems, such as the insertion of NOP instructions to avoid CPU defects. In other cases, it may be sufficient to ignore the failures.
- **Resource limitations.** For example, some tests may require more memory than is available on the target system, or may require that operations complete faster than can reasonably be achieved on the target system. These failures should be ignored.
- **Defects in the toolchain.** These defects must be further analyzed to determine (a) the significance of the failure (i.e., its likely impact on software developers using the toolchain), and (b) the difficulty of fixing the defect.

You should categorize each of the failures you observe so that you can fully evaluate the quality of your toolchain.

How will you correct defects in your toolchain?

If you have found problems through the validation process, you will need to fix them before deploying your toolchain. Even if all of your validation has been successful, you may encounter problems in the course of using the toolchain. In either event, you will need to determine the cause of the problem and then develop a solution.

Since the toolchain contains several million lines of code, the first step is to identify the component that is causing the problem. Then, you will have to debug that component. You may have to spend some time becoming familiar with the source code for the toolchain before you can correct the defect. You may wish to post a description of your problem to the public mailing list for the affected component asking for help. Some components also have publicly accessible defect-tracking systems that can be used to report problems. When you have fixed the problem, you should consider contributing the change that you have made to the public source repository for the affected component so that others can benefit from your improvement, just as you have benefited from theirs.

Mentor Embedded has developed technology for comparing test logs from multiple runs and keeps track of “expected” test failures so that any new failures can be readily identified. This technology allows Mentor Embedded to ensure that new versions of Sourcery CodeBench are at least as reliable as older toolchains. Mentor Embedded’s expert team of software engineers (known as Sourcerers) includes well-recognized developers for all components of the GNU toolchain. The Sourcerers have made thousands of improvements to the GNU toolchain since its inception in 1997.

THE CODESOURCERY STORY

CodeSourcery was founded in 1997, to focus on innovation and the advancement of open source toolchains. In the 13 years leading up to the acquisition of CodeSourcery by Mentor Graphics, the “Sourcerers” established a world-class engineering organization, which has lead major advancements in GNU gcc, glibc, and the gdb projects.

The Sourcerers have also served as a key partners to numerous semiconductor vendors introducing GNU toolchains to new semiconductor architectures and implementing performance optimizations. As part of Mentor, the Sourcerers now build a wider array of software tools that enable Mentor customers to get the most out of their hardware platforms ranging from embedded devices to supercomputers. The mission remains the same: to create products and services that deliver on the promise of open source software and open standards.

WHICH VERSION OF SOURCERY CODEBENCH IS BEST FOR YOU?

Mentor Embedded Sourcery CodeBench is available in three versions:

- **Sourcery™ CodeBench Professional Edition:** Includes unlimited support directly from the Mentor Embedded engineers who have contributed thousands of enhancements to the GNU toolchain since 1997. Users receive quick responses through Mentor Embedded's online issue tracker.
- **Sourcery™ CodeBench Standard Edition:** Includes the same comprehensive suite of tools as the Professional Edition. Users receive unlimited technical support and access to updated releases during the duration of the subscription term.
- **Sourcery™ CodeBench Personal Edition:** An affordable solution for the individual developer or small enterprise. Users receive the quality and convenience of Sourcery CodeBench and free updates for one full year.

For more information, please visit mentor.com/embedded

Android is a trademark of Google Inc. Use of this trademark is subject to Google Permissions. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

For the latest product information, call us or visit: www.mentor.com

©2011 Mentor Graphics Corporation, all rights reserved. This document contains information that is proprietary to Mentor Graphics Corporation and may be duplicated in whole or in part by the original recipient for internal business purposes only, provided that this entire notice appears in all copies. In accepting this document, the recipient agrees to make every reasonable effort to prevent unauthorized use of this information. All trademarks mentioned in this document are the trademarks of their respective owners.

Corporate Headquarters Mentor Graphics Corporation 8005 SW Boeckman Road Wilsonville, OR 97070-7777 Phone: 503.685.7000 Fax: 503.685.1204 Sales and Product Information Phone: 800.547.3000 sales_info@mentor.com	Silicon Valley Mentor Graphics Corporation 46871 Bayside Parkway Fremont, CA 94538 USA Phone: 510.354.7400 Fax: 510.354.7467 North American Support Center Phone: 800.547.4303	Europe Mentor Graphics Deutschland GmbH Arnulfstrasse 201 80634 Munich Germany Phone: +49.89.57096.0 Fax: +49.89.57096.400	Pacific Rim Mentor Graphics (Taiwan) Room 1001, 10F International Trade Building No. 333, Section 1, Keelung Road Taipei, Taiwan, ROC Phone: 886.2.87252000 Fax: 886.2.27576027	Japan Mentor Graphics Japan Co., Ltd. Gotenyama Garden 7-35, Kita-Shinagawa 4-chome Shinagawa-Ku, Tokyo 140-0001 Japan Phone: +81.3.5488.3033 Fax: +81.3.5488.3004
---	--	--	--	---

