White paper

Embedded development using the GNU toolchain for ARM® processors

DOCUMENT IDENTIFICATION

ASW-WPGNU	February 2012

REVISION

First version	
Second version	Update April 2015

Atollic AB
Science Park
Gjuterigatan 7
SE- 553 18 Jönköping
Sweden

+46 (0) 36 19 60 50

E-mail: sales@atollic.com
Web: www.atollic.com

Atollic Inc
241 Boston Post Road West (1st Floor),
Marlborough,
Massachusetts 01752
USA

+1 (973) 784 0047 (Voice)
+1 (877) 218 9117 (Toll Free)
+1 (973) 794 0075 (Fax)

E-mail: sales.usa@atollic.com
Web: www.atollic.com

Contents

Tables

No table of figures entries found.

ABSTRACT

In the past decade, there has been an unmistakable trend in the embedded industry towards the use of GNU build tools and open-source build management tools such as ECLIPSE™. GNU tools and the Eclipse IDE are freely available, of high quality, and are becoming the de-facto standard development environment in the software industry.

This white paper will focus on considerations of using the GNU C/C++ compiler and associated tools for embedded systems development, for ARM-based projects and together with the ECLIPSE™ integrated development environment.

The information outlined in this white paper will give the reader a better understanding of the advantages of using the GNU compiler and debugger tools, as well as familiarity with the areas where the open-source solutions have limitations in embedded development work. It will also enlighten the readership as to the extent of effort required to configure, build, integrate and test an Eclipse/GNU tool solution along with ancillary components that are incorporated for productivity enhancement.

INTRODUCTION

The GNU compiler is one of the most widely used C/C++ compilers in the world. It is the basic build tool for building all embedded Linux and Android systems, as well as all desktop or server Linux operating systems and their applications. The GNU compiler is also used to build many commercial real-time operating systems, such as products from Enea®, QNX®, WindRiver® and many more.

GNU tools are being constantly improved upon by both commercial companies and private individuals. The GNU compiler is widely disseminated in academia as a teaching tool, and as the subject of computer science research projects. The expertise and enhancements developed there quickly become state-of-the-art compiler technology that finds its way into practical GNU tools use, and is of enormous benefit to real-world developers. Few companies, if any, are by themselves able to keep up with the pace of the collaborative development efforts done on GNU tools by a large amount of individuals, companies and academic research institutions.

The GNU toolchain is open-source, and thus freely available. It is available for most microprocessor architectures, including the different ARM processor cores. This makes it an ideal starting point for embedded software development. However, prospective GNU tools users must weigh a number of critically important factors before deciding whether or not GNU and open-source tools are the best option for their particular situation. Among these are cost, time, and requirements for capabilities that pertain to the software build process.

- Build the tools yourself or buy a readymade solution?
- How to handle quality assurance and validation?
- What level of integration with the Eclipse IDE is needed?
- How to connect the debugger to a JTAG debug probe?
- How will Flash loading and Flash debugging be handled?
- What about device specific debug features?
- Do I want to perform system analysis and real-time tracing using the ARM Serial Wire Viewer (SWV) technology?
- Do I need to use ETM/ETB instruction tracing?
- Can I use the hard fault crash analysis capabilities of Cortex-M cores?
- Is there any support for dual core debugging?
- What about RTOS kernel aware debug support?

And so while the GNU tools are freely available, and can be built from scratch by any individual or organization with the right skills and enough available time, this may not be the best strategy for professional development teams.

Readymade commercial GNU-based development tools, such as Atollic TrueSTUDIO®, provide a pre-built and validated tool that works “out-of-the-box,” conferring immediate productivity. While commercial products represent an up-front cost over downloading and building the tools manually, they are typically less expensive in the long run, and provide a very strong return on investment, especially when the cost of on-going maintenance is factored in.

Additionally, commercially offered products, like Atollic TrueSTUDIO, typically gives a lot of extra features that professional developers value and use productively; such as project wizards, readymade example projects for hundreds of ARM devices and evaluation boards, advanced debug features such as peripheral register (SFR) viewers and Serial Wire Viewer (SWV) real-time event- and data tracing, ETM/ETB instruction tracing, hard fault crash analysis, RTOS kernel aware debugging, as well as Flash support and integration with JTAG probe debug hardware, such as Segger J-Link.

Compared to similar commercial tools with proprietary compilers, you typically get a matching toolchain at a considerable lower price if you decide upon a commercially packaged tool that is based on the GNU toolchain and the Eclipse IDE. And you are joining the strongly emerging de-facto standard at the same time.

THE GNU TOOLS

A C or C++ compiler, assembler and linker are at the heart of any embedded development toolchain. The same goes for a critical tool such as a debugger. The GNU toolchain is not just build and link tools, but a whole suite of high-quality command line tools for management, build, debugging and testing ARM-based embedded applications:

- Make utility
- C/C++ compilers
- Assembler
- Linker
- C/C++ runtime and math libraries
- Librarian
- Other binary utilities
- Debugger

While not part of the GNU toolchain as such, other software are often used in combination with the GNU tools, such as an Eclipse-based IDE, and gdbservers that enables the GNU debugger (gdb) to connect to embedded JTAG probe debug hardware required to debug embedded targets.

THE GNU MAKE UTILITY (MAKE)

Although technically not part of the GNU C/C++ compiler toolchain, the GNU make utility is often used to drive the build process. The make utility reads a makefile which is normally maintained by the developer and then builds the input files in an appropriate manner using the assembler, compiler and linker in conjunction with the dependency rules defined in the makefile.

The make utility and associated makefiles are perfectly usable, but many professional development teams of today prefer a tool-managed build process that does not require the manual maintenance of a makefile. This is a considerable time-saving feature because the IDE is managing the makefile on behalf of the developer. This capability is offered in Eclipse-based IDE’s, such as Atollic TrueSTUDIO.

THE GNU C/C++ COMPILERS (GCC AND G++)

The GNU compilers are highly optimizing compilers that support both the C and C++ languages. The GNU C compiler (gcc) supports the C language, while the GNU C++ compiler (G++) supports the C++ language.

The GNU C/C++ compilers have a number of extensions that make them well suited for embedded development, including fine-grained control using many command-line options, inline assembly, and compiler #pragmas that helps control the compiler behavior in great detail. Furthermore, the GNU compilers have extensive support for a large number of target processors and language dialect standards.

Some of the feature highlights are:

- Highly optimizing C/C++ compiler
- Supports inline assembly
- Supports interrupt handlers in C
- Supports a large number of generic and target specific #pragmas
- Generates optimized production code or debug instrumented code
- Generates information for linker optimization (dead code & data removal)
- Generates reports (disassembly list files)
- Extensive set of command line options to control the operation

The following C language standards are supported:

- C89 (original ANSI-C, X3.159-1989)
- C90 (original ANSI-C, ISO/IEC 9899:1990)
- C94/C95 (amendment 1, AMD1)
- C99 (ISO/IEC 9899:1999, supported with few exceptions)
- C11 (supported with some exceptions)
- GNU language extensions

The following C++ language standards are supported:

- C++98 (ISO/IEC 14882:1998, supported with few exceptions)
- C++03 (ISO/IEC 14882:2003, supported with few exceptions)
- C++11 (ISO/IEC 14882:2011, preliminary support)

The GNU compilers support almost all CPU architectures in common use, such as Intel x86, Renesas SH and RX, PowerPC, MIPS etc. This white paper is focused on ARM processors, so it should be mentioned that the GNU compilers have excellent support for the ARM cores, including for example:

- ARM7™
- ARM9™
- Cortex™-M0
- Cortex-M0+
- Cortex-M1
- Cortex-M3
- Cortex-M4
- Cortex-M7
- Cortex-R4
- Cortex-R5
- Cortex-R7
- Cortex-A5
- Cortex-A7
- Cortex-A8
- Cortex-A9
- Cortex-A15
- Cortex-A17
- Cortex-A53
- Cortex-A57
- Cortex-A72

The GNU C and C++ compilers are open-source, but their licenses (GPL) do not affect the application you build with them. In other words, the GNU compilers and associated libraries are open-source, but applications built using these tools can be as proprietary or closed-source as desired.

THE GNU MACRO ASSEMBLER (GAS)

The GNU assembler (gas) is used to convert assembler source code to object files (machine code). Some of the feature highlights are:

- Assembles CPU specific assembler source code to a format suitable for linking
- C/C++ preprocessor directives like #define and #ifdef
- C multiline comments in the form of /* ... */
- Numerical expressions are evaluated by the assembler
- More than 100 assembler directives for various purposes
- Macro's with support for loops
- Extensive set of command line options to control the operation

THE GNU LINKER (LD)

The GNU linker (ld) links object files and pre-compiled library files into an application binary file. Some of the feature highlights are:

- Links object files and pre-compiled library files into a binary ELF file
- Removes unused C/C++ functions and code sections (dead code removal)
- Removes unused C/C++ variables and data sections (dead data removal)
- Powerful linker script command and configuration language
- Linker script language supports C/C++ numerical expressions
- Generate memory usage reports (map files)
- Extensive set of command line options to control the operation

THE C/C++ RUNTIME AND MATH LIBRARIES

Any compiler needs runtime and math libraries in order to build anything other than trivial applications. The GNU compiler is no exception. The following libraries are normally included in the toolchain:

- C runtime library

- C math library
- C++ runtime library

The most commonly used C runtime library for use in embedded systems is newlib, while the most common C++ runtime library is stdlibc++. These compiler libraries are open-source, but their licenses do not affect the application you develop. In other words, the GNU compilers and associated libraries are open-source, but applications built using these tools can be as proprietary or closed-source as desired.

In addition to newlib, a somewhat smaller version called newlib nano is also available. Additionally, Atollic provides a super-tiny implementation of printf()-styled functions, making use of only about 1KB of code size, in the Atollic TrueSTUDIO IDE.

THE GNU LIBRARIAN (AR)

The GNU toolchain includes the GNU archive utility (ar) that is used to create and manage pre-built software libraries, thus simplifying management and linking of software components. Some of the feature highlights are:

- Create a new library in a format suitable for linking
- Add, delete, rearrange and extract files in a library
- Print the contents of a library
- Optional script mode
- Extensive set of command line options to control the operation

OTHER BINARY UTILITIES

The GNU toolchain additionally contains a number of other command line utilities. These perform various tasks, such as:

- Convert application binary ELF-files into other formats (Intel Hex or Motorola S-Record format) better suited for flash loading using external flash loading equipment
- List symbols in object files
- Display information in object files
- Disassemble machine code in object files
- Generate symbol index in archive library files

- Display the contents of ELF format files
- List file section sizes and total size
- List printable strings from files
- Discard symbols in files
- Demangle encoded C++ symbols
- Convert addresses into file names and line numbers

THE GNU DEBUGGER (GDB)

The GNU debugger (gdb) is a powerful and flexible command line debugger. It supports all traditional features one expects from a debugger:

- Loading debug instrumented application binary (ELF) files
- Running or stepping code
- Simple or complex code and data breakpoints
- Inspecting variables, CPU registers or memory
- Script language

The gdb debugger supports over 800 commands that can be typed on its command line. Additionally, gdb commands can be scripted using a script language that supports evaluation of mathematical expressions, loops and conditional execution. This facilitates for automation of advanced debug tasks.

Unfortunately the gdb debugger by itself is not enough to create a productive debug solution for embedded ARM targets. It does not handle DRAM or Flash support or JTAG-probe integration as-is.

Nor does it handle device specific debug features required by professional developers, like providing a graphical IDE debugger, peripheral register (SFR) visualization, real-time event- and data tracing using the ARM Serial Wire Viewer (SWV) technology, instruction tracing using the ETM/ETB technology, etc. It also does not include a Cortex-M hard fault crash analyzer or RTOS kernel aware debugging. These issues and solutions to them will be discussed in detail in the chapter below.

PRACTICAL CONSIDERATIONS

The GNU tools have come a long way in recent years, and are now of very high quality and are rapidly being deployed in more and more large-scale projects. For example, Atollic has been involved in a mission-critical development project in the defense industry with over 100 developers, all using the GNU compiler. Even though the GNU tools as such are very good, there are still a number of factors to consider.

If the choice is made for a development team to use GNU tools, then the important question becomes whether it is better to download and build the GNU tools in-house, or buy a readymade and tested commercial product that includes the GNU tools. Additionally, there are many practical considerations that need to be weighed before making the build-or-buy decision. Some of the relevant factors are:

- Is there sufficient expertise in-house to identify the correct tool sources to download?
- Is there sufficient expertise to find, apply or create patches that introduces needed improvements or modifications?
- Is there sufficient expertise to configure and build the tools for embedded applications development?
- What level of testing is required by the user to ensure that the tools will meet the organization's criteria for accuracy, usability, and reproducibility?
- What is the expectation for a debug solution?
- How will flash programming, JTAG-integration and device specific debugging be handled?
- How much time is available for downloading, configuring, building, testing, and integration of components such as gdbservers, flash programmers and other productivity tools?
- Have time and resources been allocated for ongoing maintenance?
- What convenience functions and advanced capabilities are needed? For example, do you need:
    - Project wizard with out-of-the-box support for your device, board and JTAG probe
    - Readymade JTAG probe configuration and debug launching system
    - SFR peripheral register viewer compatible with the ARM CMSIS-SVD standard format
    - Real-time event- and data tracing using Serial Wire Viewer (SWV)
    - Instruction tracing using ETM/ETB

  - Cortex-M hard fault crash analyzer
  - Kernel aware RTOS debugging
  - MISRA-C coding standards compliance checking
  - Etc

In short, an organization must have realistic expectations about the amount of time and expertise that is required to "roll your own." There is a basic philosophical and practical question that must be answered, which is whether or not it is cost effective to mimic the actions of a professional tools development company that spends tens of thousands of engineering hours to perfect similar tool solutions, or if the user wishes to get a tool solution that is productive immediately, and focus time and resources on making their own product unique and ready for market.

BUILDING THE GNU TOOLCHAIN

When considering to build the GNU tools yourself or buying a readymade and quality-assured solution (such as Atollic TrueSTUDIO), it is important to understand the efforts required to roll-your-own GNU toolchain. Building the complete set of GNU tools from scratch is a complex, time consuming task. A short overview of the many steps involved is given here:

- Obtaining the correct source code trees
- Finding and applying patches
- Creating a working build environment
- Configuring the build system
- Building the GNU tools
- Testing and quality assurance
- Optimizing the runtime libraries
- Packaging the toolchain for deployment

OBTAINING THE CORRECT SOURCE CODE TREES

There are many versions of the GNU tools available in source code format on the internet. First of all, you need to decide upon which source code tree of the various components you want to use. Secondly, you will have to ensure that the different software versions are compatible with each other and with the target system you will use in your project.

Also be aware of what a version number means in the world of GNU tools. A specific version number of a tool, say the gcc compiler, applies to the source code. Dependent on the build configuration, the gcc compiler of that version number can be built in many different ways with vastly different feature-set and thus behavior.

APPLYING PATCHES

In addition to the full source code tree of a specific version number, there might be a large amount of patches available from various sources, which fixes bugs or adds features or other improvements. Some of these might be of importance to you, and some do not.

When building the GNU toolchain from scratch, it is important to be aware of what patches are available from various sources, and decide upon if you need to add them to your tools build or not. Atollic for example have created and applied many patches in various GNU tools to correct problems or add new features that is relevant to embedded ARM developers. Atollic also works closely with ARM Ltd. to help improve the GNU tools for ARM processors, for example by sharing patches or bug reports between the two companies.

CREATE A WORKING BUILD ENVIRONMENT

Building Microsoft Windows-based GNU tools that target an embedded microprocessor is a complicated task. The steps typically involved are:

1. Build the GNU toolchain targeting ARM, for running on a Linux PC host. This will be used to build the ARM runtime libraries.
2. Build the GNU toolchain targeting a Windows PC, for running on a Linux PC host. This will be used to build the GCC for ARM compiler as a Windows executable file.
3. Build a Microsoft Windows-based GNU toolchain targeting ARM, using the compiler created in step 2 above, and using the libraries created in step 1 above.

The process of using one system to build a compiler that targets a second system, and runs on a third system, is called a Canadian build. To build a GNU ARM compiler that runs on Microsoft Windows, a Linux computer is typically used, thus leveraging a Canadian build system.

Additionally, the GNU tools were originally designed to run on Unix-styled operating systems, like Linux. This causes some additional problems when building the GNU compilers for execution on Microsoft Windows host PC’s. When building the GNU tools to run on Windows based host PC’s, there are two options:

- Cygwin Unix emulator
- MingW native Windows library

If you build the GNU compiler for use with Cygwin (a UNIX emulation environment on Microsoft Windows); the GNU compiler for ARM cannot be run outside the Cygwin environment. The Cygwin emulator must thus be installed in all Windows computers where the compiler shall be run. If you build the compiler using MingW, the compiler will be a self-contained Windows executable file that that can be executed in any Windows PC without additional software being installed.

It is easier to build the GNU toolchain for use with the Cygwin emulation environment, but also less convenient for the users of the tools. Vice versa, it is more difficult to build the GNU tools into self-contained Windows applications using MingW, but it is obviously more convenient for the users of the toolchain.

Atollic and other commercial suppliers of quality-assured GNU toolchains use fully automated build systems that builds the compiler and debugger toolchain automatically. Atollic builds our GNU toolchains for use with MingW, to provide an easy-to-use toolchain with an “out-of-the-box” experience for our customers.

CONFIGURING THE BUILD SYSTEM

The GNU tools typically come with a build script that helps automating the build process; once the “build recipe” has been configured properly. There are a huge amount of options that can affect the target support, feature-set or performance. Understanding which build settings is suitable or even necessary is not trivial for engineers new to the task.

All runtime libraries must be built with required combinations of relevant code generation options, such as different CPU core variants, big- or little endian, software- or hardware floating point support, ARM- or Thumb mode, optimization for speed or code-size, etc. These different library-builds are commonly called multi-libraries, or “multilibs”.

In addition to understanding how to configure the build system, developers need to understand how to manage and test the multilibs too.

BUILDING THE GNU TOOLS

Once the build system is installed and configured properly, it is generally quite easy to actually build the GNU toolchain and associated multi-libraries. Dependent on the number multi-libraries to be built, the build time for a GNU toolchain can be anything between an hour and a day.

TESTING AND QUALITY ASSURANCE

Once the GNU toolchain has been successfully built, it must be tested such that you know you can trust it for production use. The GNU toolchain and associated libraries comprises over 48.000 source code files, with a total of over 8 million lines of code. This is obviously an advanced piece of software by any standard, which is not trivial to test and maintain. In fact, there are many tests that ought to be run to validate the toolchain, including:

- Compilation tests that ensure that legally formed programs do compile, and that malformed programs do not compile or at least generate appropriate warnings.
- Execution tests where test cases are compiled, downloaded and executed in target hardware to verify the correctness of the behavior of the generated machine code on each supported processor core or instruction set.
- Testing that the compiler generated output file can be loaded into the gdb debugger, and that the debugger works as expected with the generated ELF binary files.

- Testing that the Eclipse debugger IDE works as expected when wrapped over the new gdb debugger build, using ELF files generated by the new gcc compiler.
- Additionally other test suites, like the Plum-Hall language conformance test suites or the EEMBC performance benchmark test suites might be run to test language standards compliance or performance of the generated code.

A number of test suites are available to aid in many of the above test scenarios. The GNU compilers, runtime libraries, binary utilities and the GNU debugger all have regression test suites, which are based on the DejaGNU test framework. The DejaGNU test system uses the Expect programming language, which is based on Tcl. For example, the regression test suite for the C and C++ compilers alone contains some 100.000 test cases.

To make DejaGNU and the regression test suites to work in your test environment, board configuration file(s) must be developed for the target board(s) being used by the test system. Considerations is how to make the test system connect to the target board using a JTAG probe, make them run in the embedded target, etc.

This test suite may have to be re-run many times, to test the compiler on several different host operating systems, to test the generated machine code on various CPU core variants or instruction sets, such as various cores in the ARM7, ARM9 or Cortex-M product families. Moreover, the compiler regression test suites may have to be re-run for different optimization or code generation options, or using other command line option combinations that modify the compiler behavior.

In practice, the need for regression testing becomes multi-dimensional, and quickly explodes in number of permutations. For example, even if the compiler shall only be used on Microsoft Windows hosts, it might still have to be tested many times on different versions of the operating system.

<u>Regression testing on a number of host operating systems:</u>

- Microsoft Windows XP (32-bit)
- Microsoft Windows XP (64-bit)
- Microsoft Windows Vista (32-bit)
- Microsoft Windows Vista (64-bit)
- Microsoft Windows 7 (32-bit)
- Microsoft Windows 7 (64-bit)
- Etc.

Matters quickly get worse if different service packs or several operating system variants (Home Premium, Professional, Ultimate, etc.) should be taken into account. The same applies if you must support Linux or MacOS X host operating systems too.

Typically, a compiler toolchain is not built to support only one CPU core, but a number of CPU cores with different instruction set architectures or variants. Even if you don’t plan to support CPU architectures from other vendors, like PowerPC, MIPS or x86, there might still be a number of different CPU core variants to support and test.

<u>Regression testing on a number of ARM cores:</u>

- ARM7 (ARMv4t)
- ARM9 (ARMv5te)
- Cortex-M0 and Cortex-M1 (ARMv6-M)
- Cortex-M3 (ARMv7-M)
- Cortex-M4 (ARMv7E-M)
- Cortex-R4(F) (ARMv7-R)
- Cortex-A5, Cortex-A8 and Cortex-A9 (ARMv7-A)
- Etc.

Regression tests on the supported CPU cores might have to be done on both big- and little endian systems, using software emulated floating point as well as hardware accelerated floating point implementations, etc.

And finally, compiler generated code will likely have to be tested using a number of different combinations of compiler settings, such as different optimization levels or other command line options that modify the code generation.

<u>Regression testing on a number of build settings:</u>

- No compiler optimization (-O0)
- Compiler optimization level 1 (-O1)
- Compiler optimization level 2 (-O2)
- Compiler optimization level 2 (-O3)
- Compiler optimization for size (-Os)
- Compiler optimization for speed (-Ofast)
- Compiler optimization for debugging (-Og)
- Etc.

It can easily be understood that the number of permutations that need to be re-tested using the full regression test system explodes quickly. As one re-run of the compiler regression test suite may take up to a day or more of machine time to execute on one host-OS for one target architecture, testing time may become substantial if the test suite have to be re-run often in interactive development (applying or testing more patches) or if it has to be run many times for a large number of different test permutations.

A computing farm with a grid of test computers might well be needed for efficient regression testing of the tool suite if a large number of permutations (such as host O/S, CPU core variants, different command line options) shall be tested properly. This is not easily done without also building test scripts that drive such testing automatically. Matters are complicated further by having different test runs using different test hardware, such as different hardware boards (ARM7, ARM9, Cortex-M4, big- and little- endian variants, with or without hardware floating point, etc.).

All these problems are typically taken care of when buying a commercial distribution of embedded tools that is based on the GNU toolchain, such as Atollic TrueSTUDIO.

OPTIMIZING THE RUNTIME LIBRARIES

Newlib is developed for use in resource constrained embedded systems, but is in some cases still too large for systems with very little memory. A classic example is printf(), that contains a large amount of rarely-used formatting options, including floating point formatting support. The majority of embedded systems never use the advanced formatting options in the ANSI-C compliant printf(), and a code-size optimized version with less functionality is often of benefit.

For this reason, commercial vendors of GNU tools often provide improvements to the runtime library. Atollic for example, provides code-size optimizations of the runtime library with its Atollic TrueSTUDIO product.

In embedded systems, the C runtime library often needs to be remapped to the underlying O/S or H/W platform, such that function calls like printf() is remapped to peripherals like UART channels or LCD displays, fopen() and fclose() is remapped to API calls to an RTOS or Flash file system, etc. Newlib supports this kind of remapping in embedded systems.

Furthermore, the libraries should be instrumented with linker optimization information, that enable the optimizing linker to remove library functions not used by the application, thus reducing the memory overhead for small embedded systems. This must be considered when building the runtime libraries in the first place.

For C++, it is advisable to avoid using Runtime type information (RTTI) and exceptions, due to code size or runtime predictability.

PACKAGE THE TOOLCHAIN FOR DEPLOYMENT

Once the GNU toolchain and all its components have been built and tested, it must be prepared for deployment. On a Windows PC, this typically involves creating a graphical installation program that handles installation and uninstallation, setting up PATH variables and other environment variables, etc.

If you plan to use the GNU command line tools integrated into a graphical IDE like Eclipse, there is additional work required to configure Eclipse to use the new GNU toolchain, and perhaps create Eclipse plug-ins (written in Java) to provide for graphical configuration of build options, like supporting certain compiler or linker command line options, etc. Furthermore, considerations have to be made regarding bundling of 3rd party gdbservers, and the legal implications thereof.

Atollic package both the GNU toolchain and a much-extended Eclipse IDE into a Windows installer where all components, finely tuned to work well together, is installed with a few simple mouse clicks.

USING GCC FOR EMBEDDED DEVELOPMENT

With a properly built and tested compiler and debugger toolchain, you are ready to start development of your application (or so you may think). Unfortunately this is often not the case. When it comes to embedded development, there are some additional considerations to make.

START-UP CODE AND EXAMPLE PROJECTS

With a freshly built gcc or g++ compiler, you get no example applications that can boot the processor on your particular microcontroller device or electronic board. Therefore, you may have to write your own start-up code and learn how to boot the electronic board yourself. Note that the start-up logic may well be device-model or board-model specific, and any code for a similar processor or board you may find somewhere else might well not work on your almost similar hardware.

Managing the start-up sequence includes having a deep understanding of:

- Writing a linker configuration file
- Handling power-on-reset logic, often using assembler language
- Defining the interrupt vector table
- Writing and connecting the required interrupt handlers
- Setting up the stack pointer
- Initializing required hardware, such as configuring pin function controllers, enabling chip-select signals or setting up DRAM refresh logic needed to continue execution using hardware resources that are not enabled on reset
- Initializing the C runtime environment (copy initialized variables from Flash to RAM and clear zero-initialized variables)
- Initializing the C++ runtime environment (executing static constructors)
- Jump to main()
- Writing a sample main() application that can be used to exercise the hardware and test the debug connection

Commercially offered GNU toolchains, such as Atollic TrueSTUDIO, contain built-in support for the above items, providing out-of-the-box support for thousands of microcontroller device models and evaluation boards. Getting readymade startup logic that works out-of-the-box for your evaluation board can alone justify the cost of buying a commercially offered GNU toolchain compared to make one yourself and handle this from scratch.

CMSIS

In recent years, ARM Ltd. has started to standardize how the lower-level software shall be written, using its Cortex Microcontroller Software Interface Standard (CMSIS). CMSIS standardize things like:

- File structure and file naming conventions for lower-layer software
- Power-on-reset logic and interrupt handling
- Naming convention on software wrappers over hardware interface such as device driver API names, peripheral registers, bitfield definitions, etc.
- Real-time operating system (RTOS) API conventions
- SFR peripheral register definition files

It is recommended to follow the CMSIS standard, and you may thus have to write or integrate CMSIS compliant startup-code or device drivers yourself. Commercial suppliers of GNU tools, like Atollic, typically provide CMSIS compliant example projects that take care of these things for you.

INTEGRATION WITH THE ECLIPSE IDE

While the C/C++ part of Eclipse has native support for GNU tools out-of-the-box, this is only partly true in an embedded context. For example, some features that are missing in the standard Eclipse solution are:

- No project wizard that auto-generates readymade example projects for hundreds of ARM devices and boards with a couple of mouse clicks
- The build settings dialog box is not suitable for embedded use, as it does not provide GUI configuration of build settings related to embedded development
- The default Eclipse build system does sometimes not feel intuitive to some embedded C/C++ developers with experience from other tools.

Atollic TrueSTUDIO for example has addressed these issues with a powerful project wizard that auto-generate embedded projects for hundreds of ARM devices and boards. Additionally, the build configuration GUI has been extended by Atollic to be perfectly adapted for embedded needs.

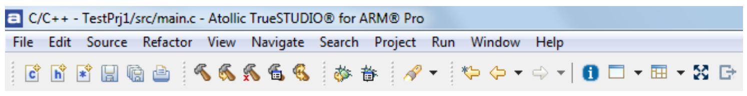

C/C++ - TestPrj1/src/main.c - Atollic TrueSTUDIO® for ARM® Pro
File Edit Source Refactor View Navigate Search Project Run Window Help

Figure 1 - The more C/C++ oriented build system in Atollic TrueSTUDIO

And finally, some of the GUI behaviors in standard Eclipse that feels unintuitive to many C/C++ developers have been replaced by Atollic, offering a more easy-to-use and productive build environment.

USING GDB FOR EMBEDDED DEBUGGING

Once building and testing the tools and associated runtime libraries have been resolved, the practical considerations for embedded in-target debugging using gdb need to be handled. While gdb is a powerful debugger for execution-control and target inspection of the CPU core itself, it is not practical to use as-is in professional embedded development due to a number of reasons, as outlined below.

SUPPORT FOR JTAG PROBE DEBUG HARDWARE

In order to debug embedded applications running in a target board, a JTAG hardware debug probe (for example Segger J-Link or ST-LINK) is needed. While gdb facilitates for remote debugging of target hardware using a JTAG probe, gdb requires a debugger back-end (a gdbserver) to connect to the JTAG probe and thus to the embedded micro-controller. Such gdbservers are not part of gdb and needs to be obtained elsewhere.

While gdb is an open-source and freely available debugger, the required gdbserver is most often not open-source software. For example, to debug an ARM board using a Segger J-Link or ST-LINK JTAG probe, a proprietary gdbserver is needed from a commercial vendor like Segger or Atollic.

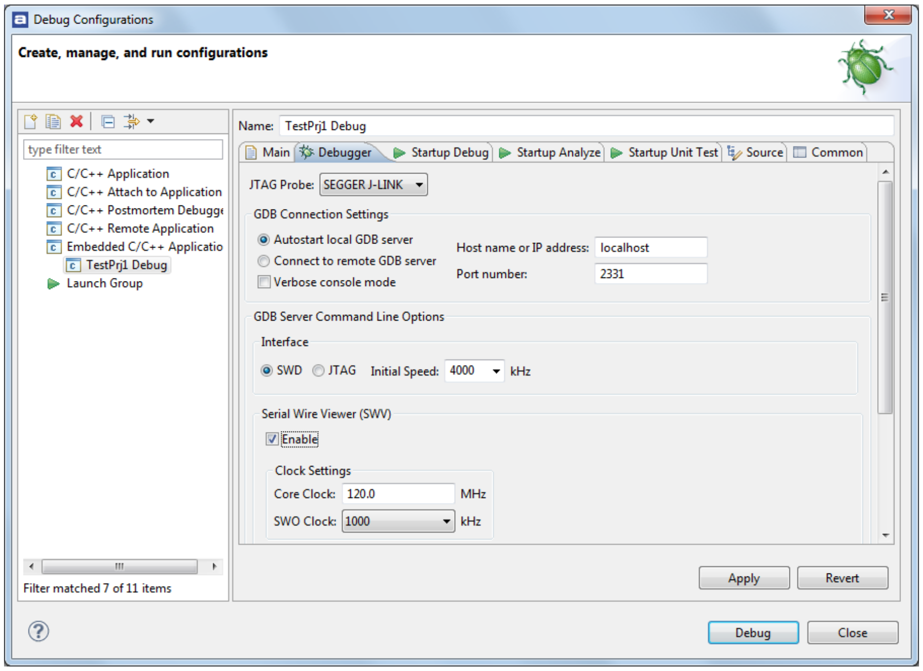
If you have obtained the required gdbserver, the open-source software modules (including standard-Eclipse) do not auto-start and auto-stop the gdbserver on each debug session, creating an inefficient work environment for professional developers. Also, gdb does not connect to JTAG-probe debug hardware or associated gdbservers out-of-the-box; thus it can be difficult to setup a working solution for new users.

Most commercial packages of embedded GNU tools, such as Atollic TrueSTUDIO, handle these problems out-of-the-box.

INTEGRATION WITH THE ECLIPSE DEBUGGER

As gdb is a command-line debugger, a graphical debugger user interface (typically Eclipse) is needed for convenient use. But the standard, open-source, Eclipse C/C++ debugger lacks a lot of embedded-specific features that is needed to create an efficient debug environment; ranging from small usability issues to major integration or feature omissions.

For example, the standard Eclipse implementation does not automatically switch from the debug perspective to the editing perspective when the debug session is terminated, creating frustration with many developers. Finely tuned commercial Eclipse-based products, like Atollic TrueSTUDIO, handle this type of usability improvements.


Debug Configurations

Create, manage, and run configurations

type filter text

- C/C++ Application
- C/C++ Attach to Application
- C/C++ Postmortem Debugge
- C/C++ Remote Application
- Embedded C/C++ Applicatio
  - TestPrj1 Debug
- Launch Group

Filter matched 7 of 11 items

Name: TestPrj1 Debug

Main | Debugger | Startup Debug | Startup Analyze | Startup Unit Test | Source | Common

JTAG Probe: SEGGER J-LINK

GDB Connection Settings

- [✓ Autostart local GDB server]
- Connect to remote GDB server
- Verbose console mode

Host name or IP address: localhost

Port number: 2331

GDB Server Command Line Options

Interface

[✓ SWD] JTAG Initial Speed: 4000 kHz

Serial Wire Viewer (SWV)

[✓ Enable]

Clock Settings

Core Clock: 120.0 MHz

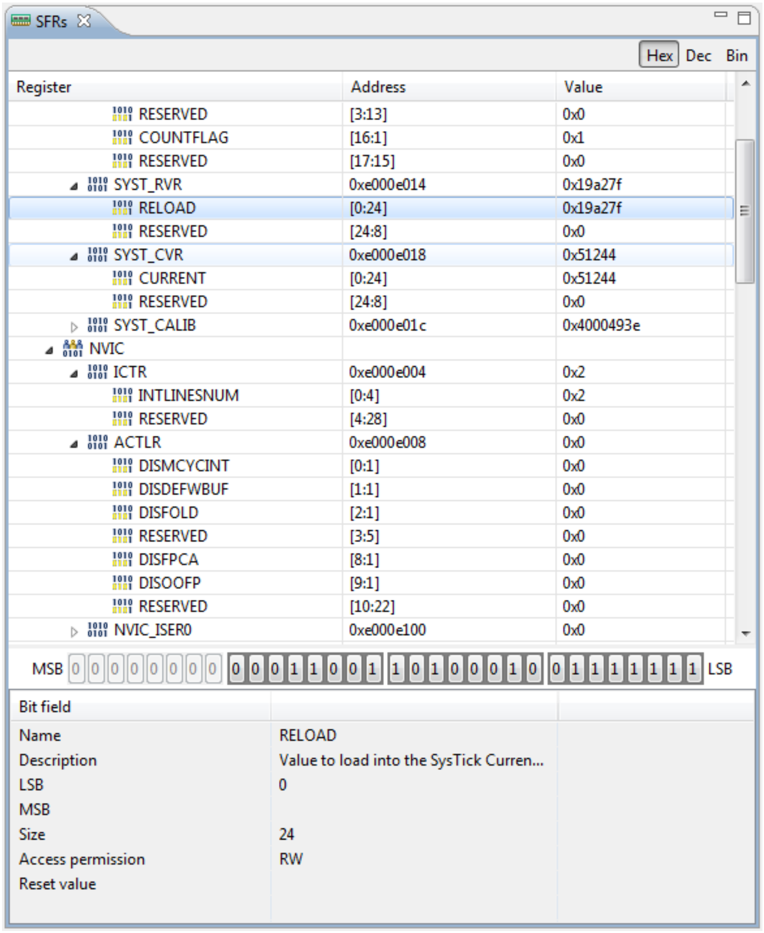
SWO Clock: 1000 kHz

Figure 2 - JTAG probe integration in Atollic TrueSTUDIO

Integrating the Eclipse debugger GUI and the gdb command-line debugger can be troublesome for inexperienced users. And even if gdb is complemented by an Eclipse debugger GUI, there is no configuration GUI for the gdbserver or JTAG probe in the standard Eclipse solution. All this is taken care of when choosing a commercial solution like Atollic TrueSTUDIO.

DEVICE SPECIFIC DEBUGGING

gdb is a good generic debugger when it comes to the CPU core itself, but it does not have any readymade “out-of-the-box” configurations for any specific microcontroller devices or evaluation boards, making it difficult to get started for many new users. gdb provides generic debug support for the CPU core, but it provides no device or board specific debug features, like peripheral register (SFR) visualisation.



SFRs

Hex Dec Bin

Register	Address	Value
RESERVED	[3:13]	0x0
COUNTFLAG	[16:1]	0x1
RESERVED	[17:15]	0x0
SYST_RVR	0xe000e014	0x19a27f
RELOAD	[0:24]	0x19a27f
RESERVED	[24:8]	0x0
SYST_CVR	0xe000e018	0x51244
CURRENT	[0:24]	0x51244
RESERVED	[24:8]	0x0
SYST_CALIB	0xe000e01c	0x4000493e
NVIC		
ICTR	0xe000e004	0x2
INTLINESNUM	[0:4]	0x2
RESERVED	[4:28]	0x0
ACTLR	0xe000e008	0x0
DISMCYCINT	[0:1]	0x0
DISDEFWBUF	[1:1]	0x0
DISFOLD	[2:1]	0x0
RESERVED	[3:5]	0x0
DISFPCA	[8:1]	0x0
DISOOFP	[9:1]	0x0
RESERVED	[10:22]	0x0
NVIC_ISER0	0xe000e100	0x0

MSB 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 0 0 0 1 0 0 1 1 1 1 1 1 1 LSB

Bit field	
Name	RELOAD
Description	Value to load into the SysTick Curren...
LSB	0
MSB	
Size	24
Access permission	RW
Reset value	

Figure 3 - The Peripheral register (SFR) viewer in Atollic TrueSTUDIO

Another area that can cause problems is how to get gdb to be able to download and debug code in Flash or DRAM memory. The gdb debugger knows nothing about configuring DRAM devices (enabling chip-select pins, configuring DRAM refresh settings, etc.) on specific evaluation boards, sometimes making it difficult to setup for use in boards with DRAM-based memory configurations.

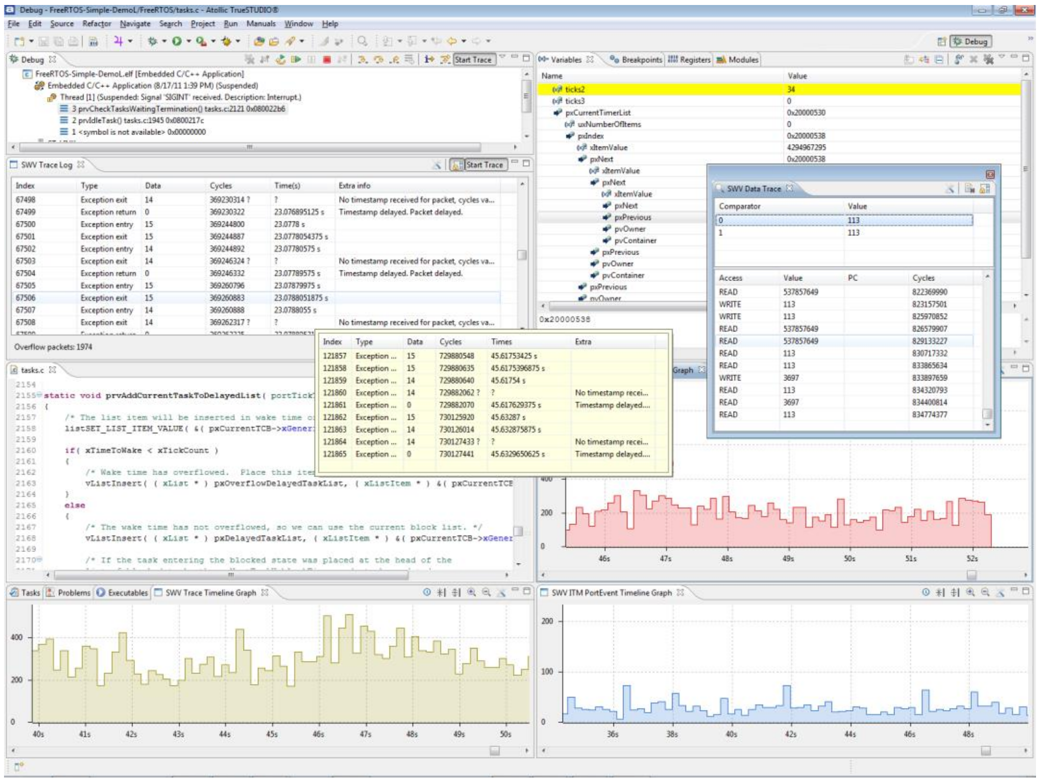
Similarly, the gdb debugger knows nothing about what Flash memory is used on your board and what Flash programming algorithm must be supported to download and debug your application in Flash memory. To debug code in Flash, gdb must be complimented by a gdbserver that knows how to use the Flash memory in your device or on your board.

Board and/or JTAG-probe specific initialization and configurations might be needed to enable certain debug modes (such as JTAG or SWD mode), or to enable use of DRAM or Flash memory, etc.

These kinds of issues are typically taken care of when choosing a commercial solution, like Atollic TrueSTUDIO.

SERIAL WIRE VIEWER (SWV) REAL-TIME TRACING

Another limitation with the gdb debugger is that it cannot handle system analysis and real-time tracing using ARM's proprietary Serial Wire Viewer (SWV) technology, which any ARM developer would like to use due to the powerful debug features it enables at low cost in the debug (JTAG probe) hardware.


Debug - FreeRTOS-Simple-DemoL/FreeRTOS/tasks.c - Atollic TrueSTUDIO®
File Edit Source Refactor Navigate Search Project Run Manuals Window Help
Debug
Start Trace
Variables
Breakpoints
Registers
Modules
SWV Trace Log
tasks.c
SWV Data Trace
Graph
Tasks
Problems
Executables
SWV Trace Timeline Graph
SWV ITM PortEvent Timeline Graph

```
FreeRTOS-Simple-DemoL.elf [Embedded C/C++ Application]
  Embedded C/C++ Application (8/17/11 1:39 PM) (Suspended)
    Thread [1] (Suspended: Signal 'SIGINT' received. Description: Interrupt.)
      3 prvCheckTasksWaitingTermination() tasks.c:2121 0x080022b6
      2 prvIdleTask() tasks.c:1945 0x0800217c
      1 <symbol is not available> 0x00000000
```

Index	Type	Data	Cycles	Time(s)	Extra info
67498	Exception exit	14	369230314 ?	?	No timestamp received for packet, cycles va...
67499	Exception return	0	369230322	23.076895125 s	Timestamp delayed. Packet delayed.
67500	Exception entry	15	369244800	23.0778 s	
67501	Exception exit	15	369244887	23.0778054375 s	
67502	Exception entry	14	369244892	23.07780575 s	
67503	Exception exit	14	369246324 ?	?	No timestamp received for packet, cycles va...
67504	Exception return	0	369246332	23.07789575 s	Timestamp delayed. Packet delayed.
67505	Exception entry	15	369260796	23.07879975 s	
67506	Exception exit	15	369260883	23.0788051875 s	
67507	Exception entry	14	369260888	23.0788055 s	
67508	Exception exit	14	369262317 ?	?	No timestamp received for packet, cycles va...

Overflow packets: 1974

```
static void prvAddCurrentTaskToDelayedList( portTick
{
    /* The list item will be inserted in wake time o
    listSET_LIST_ITEM_VALUE( &( pxCurrentTCB->xGener

    if( xTimeToWake < xTickCount )
    {
        /* Wake time has overflowed.  Place this ite
        vListInsert( ( xList * ) pxOverflowDelayedTaskList, ( xListItem * ) &( pxCurrentTCB
    }
    else
    {
        /* The wake time has not overflowed, so we can use the current block list. */
        vListInsert( ( xList * ) pxDelayedTaskList, ( xListItem * ) &( pxCurrentTCB->xGener

        /* If the task entering the blocked state was placed at the head of the
```

Index	Type	Data	Cycles	Times	Extra
121857	Exception ...	15	729880548	45.61753425 s	
121858	Exception ...	15	729880635	45.6175396875 s	
121859	Exception ...	14	729880640	45.61754 s	
121860	Exception ...	14	729882062 ?	?	No timestamp recei...
121861	Exception ...	0	729882070	45.617629375 s	Timestamp delayed....
121862	Exception ...	15	730125920	45.63287 s	
121863	Exception ...	14	730126014	45.632875875 s	
121864	Exception ...	14	730127433 ?	?	No timestamp recei...
121865	Exception ...	0	730127441	45.6329650625 s	Timestamp delayed....

Name	Value
ticks2	34
ticks3	0
pxCurrentTimerList	0x20000530
uxNumberOfItems	0
pxIndex	0x20000538
xItemValue	4294967295
pxNext	0x20000538
xItemValue	
pxNext	
xItemValue	
pxNext	
pxPrevious	
pvOwner	
pvContainer	
pxPrevious	
pvOwner	
pvContainer	
pxPrevious	
pvOwner	

0x20000538

Comparator	Value
0	113
1	113

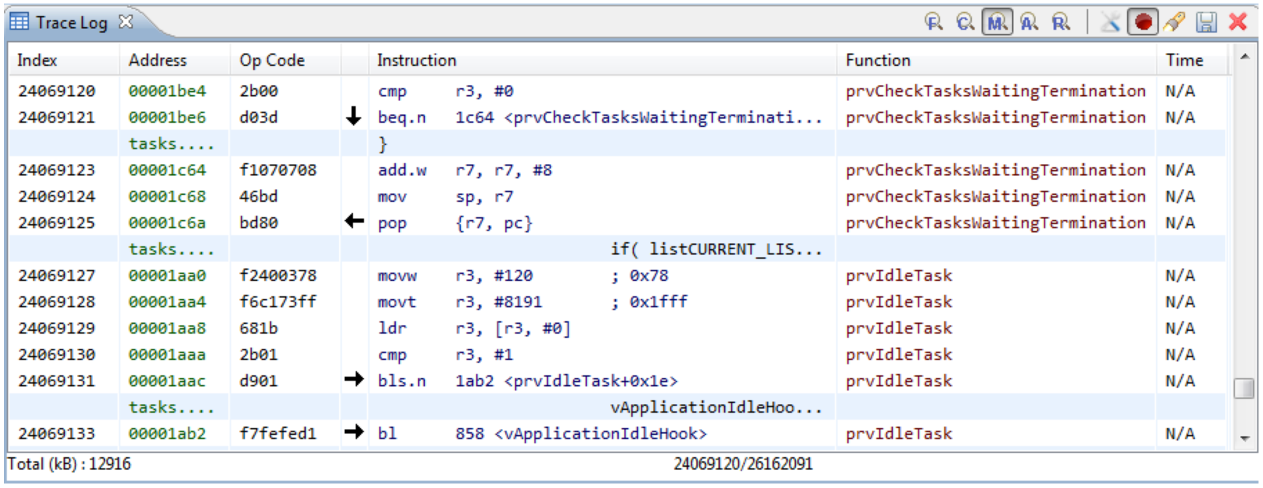
Access	Value	PC	Cycles
READ	537857649		822369990
WRITE	113		823157501
WRITE	113		825970852
READ	537857649		826579907
READ	537857649		829133227
READ	113		830717332
READ	113		833865634
WRITE	3697		833897659
READ	113		834320793
READ	3697		834400814
READ	113		834774377

Figure 4 - Real-time tracing using SWV in the Atollic TrueSTUDIO debugger

Some commercial development tools, like Atollic TrueSTUDIO, bypass the limitations in gdb and provide a state-of-the-art implementation of Serial Wire Viewer event- and data real-time tracing, also when using the GNU debugger. Features that is provided is event-, memory access- and interrupt tracing, data monitoring and visualisation, execution time profiling, printf() redirection back to the debugger GUI, etc.

INSTRUCTION TRACING USING ETM/ETB

The standard Eclipse/gdb solution does not support instruction tracing, which is a critically important debug technique in embedded systems. The capability to record the execution history for later analysis is very useful when trying to find certain types of bugs, or when you are not allowed to stop on breakpoints for single-stepping (for example motor control applications, where you damage hardware equipment if execution stops).



Trace Log

Index	Address	Op Code		Instruction	Function	Time
24069120	00001be4	2b00		cmp r3, #0	prvCheckTasksWaitingTermination	N/A
24069121	00001be6	d03d	↓	beq.n 1c64 <prvCheckTasksWaitingTerminati...	prvCheckTasksWaitingTermination	N/A
	tasks....			}		
24069123	00001c64	f1070708		add.w r7, r7, #8	prvCheckTasksWaitingTermination	N/A
24069124	00001c68	46bd		mov sp, r7	prvCheckTasksWaitingTermination	N/A
24069125	00001c6a	bd80	←	pop {r7, pc}	prvCheckTasksWaitingTermination	N/A
	tasks....			if(listCURRENT_LIS...		
24069127	00001aa0	f2400378		movw r3, #120 ; 0x78	prvIdleTask	N/A
24069128	00001aa4	f6c173ff		movt r3, #8191 ; 0x1fff	prvIdleTask	N/A
24069129	00001aa8	681b		ldr r3, [r3, #0]	prvIdleTask	N/A
24069130	00001aaa	2b01		cmp r3, #1	prvIdleTask	N/A
24069131	00001aac	d901	→	bls.n 1ab2 <prvIdleTask+0x1e>	prvIdleTask	N/A
	tasks....			vApplicationIdleHoo...		
24069133	00001ab2	f7fefed1	→	bl 858 <vApplicationIdleHook>	prvIdleTask	N/A

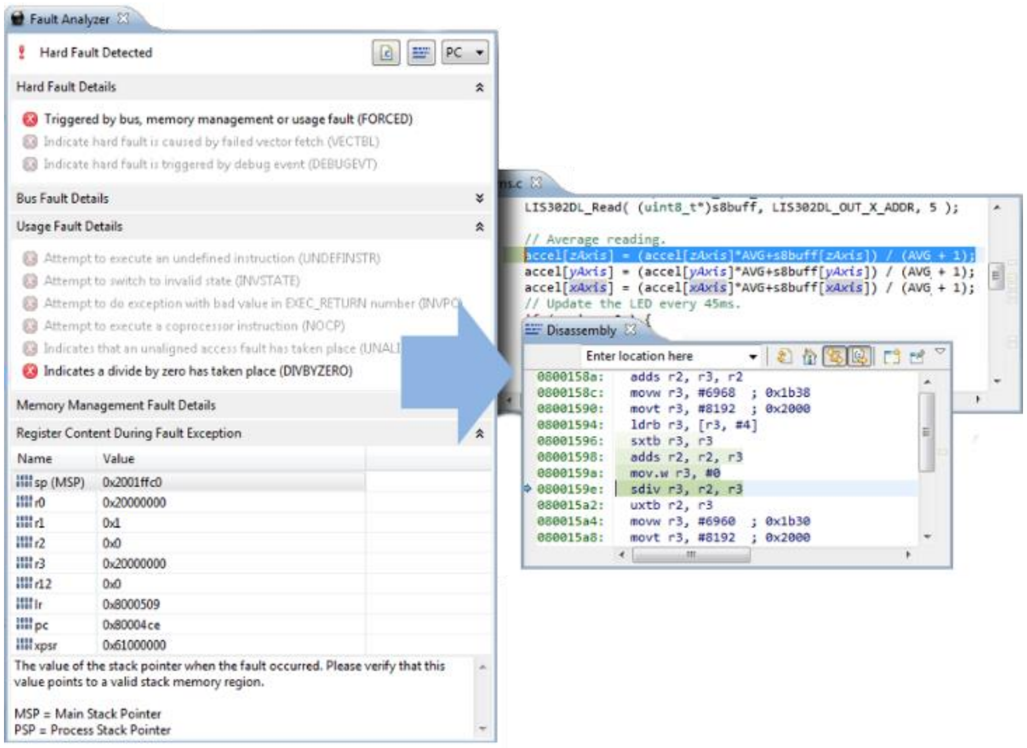
Total (kB) : 12916 24069120/26162091

Figure 5 - Instruction tracing in Atollic TrueSTUDIO

If you need instruction tracing with gdb, commercial solutions like Atollic TrueSTUDIO is the only way to go.

HARD FAULT CRASH ANALYSER

The Cortex-M core enables powerful debuggers to implement a hard fault crash analyser, telling you why the software brought the CPU into a hardware fault state, what code line made it, and why. Typical examples are division by zero, illegal memory access, etc.



Hard Fault Detected

Hard Fault Details
- Triggered by bus, memory management or usage fault (FORCED)
- Indicate hard fault is caused by failed vector fetch (VECTBL)
- Indicate hard fault is triggered by debug event (DEBUGEVT)

Bus Fault Details

Usage Fault Details
- Attempt to execute an undefined instruction (UNDEFINSTR)
- Attempt to switch to invalid state (INVSTATE)
- Attempt to do exception with bad value in EXEC_RETURN number (INVPC)
- Attempt to execute a coprocessor instruction (NOCP)
- Indicates that an unaligned access fault has taken place (UNALI
- Indicates a divide by zero has taken place (DIVBYZERO)

Memory Management Fault Details

Register Content During Fault Exception

Name	Value
sp (MSP)	0x2001ffc0
r0	0x20000000
r1	0x1
r2	0x0
r3	0x20000000
r12	0x0
lr	0x8000509
pc	0x80004ce
xpsr	0x61000000

The value of the stack pointer when the fault occurred. Please verify that this value points to a valid stack memory region.

MSP = Main Stack Pointer
PSP = Process Stack Pointer

```
LIS302DL_Read( (uint8_t*)s8buff, LIS302DL_OUT_X_ADDR, 5 );

// Average reading.
accel[zAxis] = (accel[zAxis]*AVG+s8buff[zAxis]) / (AVG + 1);
accel[yAxis] = (accel[yAxis]*AVG+s8buff[yAxis]) / (AVG + 1);
accel[xAxis] = (accel[xAxis]*AVG+s8buff[xAxis]) / (AVG + 1);
// Update the LED every 45ms.
```

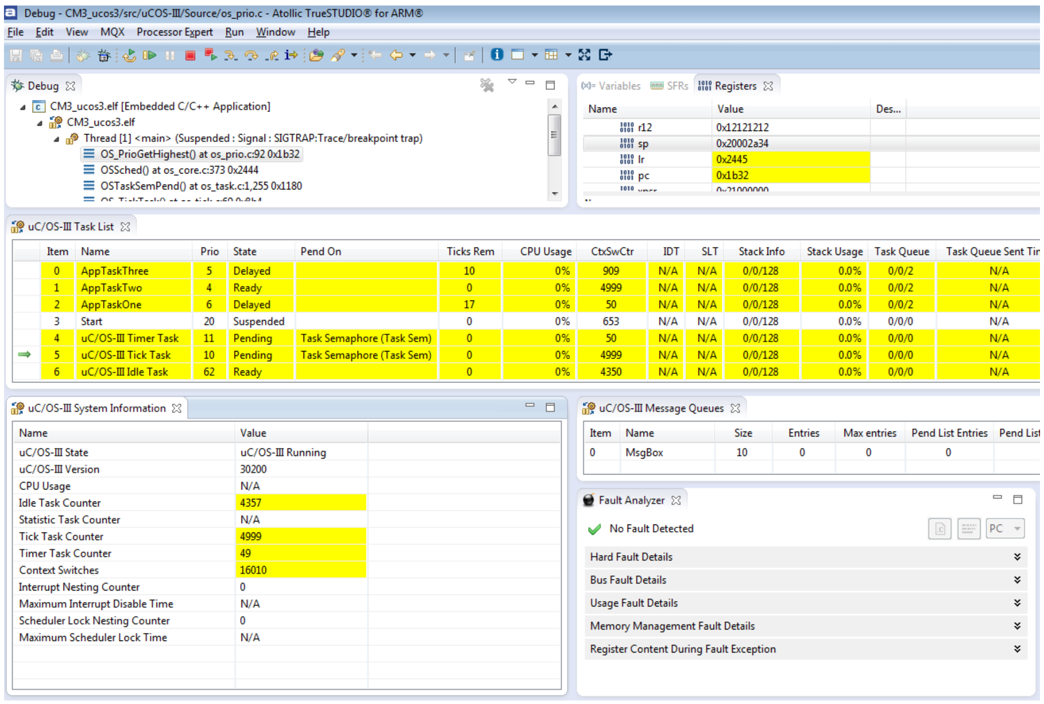
```
0800158a:   adds r2, r3, r2
0800158c:   movw r3, #6968   ; 0x1b38
08001590:   movt r3, #8192   ; 0x2000
08001594:   ldrb r3, [r3, #4]
08001596:   sxtb r3, r3
08001598:   adds r2, r2, r3
0800159a:   mov.w r3, #0
0800159e:   sdiv r3, r2, r3
080015a2:   uxtb r2, r3
080015a4:   movw r3, #6960   ; 0x1b30
080015a8:   movt r3, #8192   ; 0x2000
```

This type of very powerful debugger techniques are implemented in Atollic TrueSTUDIO, but not available using standard Eclipse/GDB debugger solutions.

KERNEL AWARE RTOS DEBUGGING

If you use an RTOS, such as FreeRTOS, Segger embOS, Micrium uC/OS or ThreadX from ExpressLogic, you really want the debugger to be RTOS aware and display RTOS specific debug information.

Again, these types of functionalities are not available in standard Eclipse/gdb, but are available in commercial high-end implementations of Eclipse/GNU tools, like Atollic TrueSTUDIO.


Debug - CM3_ucos3/src/uCOS-III/Source/os_prio.c - Atollic TrueSTUDIO® for ARM®
File Edit View MQX Processor Expert Run Window Help
Debug
Variables SFRs Registers
uC/OS-III Task List
uC/OS-III System Information
uC/OS-III Message Queues
Fault Analyzer

```
CM3_ucos3.elf [Embedded C/C++ Application]
  CM3_ucos3.elf
    Thread [1] <main> (Suspended : Signal : SIGTRAP:Trace/breakpoint trap)
      OS_PrioGetHighest() at os_prio.c:92 0x1b32
      OSSched() at os_core.c:373 0x2444
      OSTaskSemPend() at os_task.c:1,255 0x1180
```

Name	Value	Des...
r12	0x12121212	
sp	0x20002a34	
lr	0x2445	
pc	0x1b32	

	Item	Name	Prio	State	Pend On	Ticks Rem	CPU Usage	CtxSwCtr	IDT	SLT	Stack Info	Stack Usage	Task Queue	Task Queue Sent Tir
	0	AppTaskThree	5	Delayed		10	0%	909	N/A	N/A	0/0/128	0.0%	0/0/2	N/A
	1	AppTaskTwo	4	Ready		0	0%	4999	N/A	N/A	0/0/128	0.0%	0/0/2	N/A
	2	AppTaskOne	6	Delayed		17	0%	50	N/A	N/A	0/0/128	0.0%	0/0/2	N/A
	3	Start	20	Suspended		0	0%	653	N/A	N/A	0/0/128	0.0%	0/0/0	N/A
	4	uC/OS-III Timer Task	11	Pending	Task Semaphore (Task Sem)	0	0%	50	N/A	N/A	0/0/128	0.0%	0/0/0	N/A
⇒	5	uC/OS-III Tick Task	10	Pending	Task Semaphore (Task Sem)	0	0%	4999	N/A	N/A	0/0/128	0.0%	0/0/0	N/A
	6	uC/OS-III Idle Task	62	Ready		0	0%	4350	N/A	N/A	0/0/128	0.0%	0/0/0	N/A

Name	Value
uC/OS-III State	uC/OS-III Running
uC/OS-III Version	30200
CPU Usage	N/A
Idle Task Counter	4357
Statistic Task Counter	N/A
Tick Task Counter	4999
Timer Task Counter	49
Context Switches	16010
Interrupt Nesting Counter	0
Maximum Interrupt Disable Time	N/A
Scheduler Lock Nesting Counter	0
Maximum Scheduler Lock Time	N/A

Item	Name	Size	Entries	Max entries	Pend List Entries	Pend List
0	MsgBox	10	0	0	0	

No Fault Detected

Hard Fault Details
Bus Fault Details
Usage Fault Details
Memory Management Fault Details
Register Content During Fault Exception

Figure 6- Kernel aware RTOS debugging in Atollic TrueSTUDIO

Atollic TrueSTUDIO supports most RTOS products on the market, offering a powerful and convenient debug solution.

COMMERCIAL EXTENSIONS TO THE GNU DEBUGGER

Some professional development tools built using the GNU tools, such as Atollic TrueSTUDIO, handles the problems outlined in this chapter, providing an easy-to-use and pre-configured tool solution with the following benefits:

- A simple installation of the complete toolchain (Eclipse, gcc, gdb, gdbservers, etc.) using only a couple of mouse clicks with a readymade graphical installation program. The different components are finely tuned to work well together and are pre-configured for the selected device or board.

- Out-of-the-box JTAG probe connection is provided using bundled gdbservers for the most popular JTAG probes, including an integrated configuration GUI for them. Additionally, professional IDE's should auto-start and auto-stop the gdbserver on each debug session, which for example Atollic TrueSTUDIO does.

- Minor usability features like automatically switching from the Eclipse debug perspective to the editing perspective should be addressed, as is the case with Atollic TrueSTUDIO.

- Integrated Flash and DRAM support with pre-set configurations and example projects for hundreds of devices and boards.

- Integrated peripheral register (SFR) viewer enables efficient debugging of device driver and other hardware related code, providing a browser for peripheral registers, bit fields and bits during in-target debugging.

- Integrated real-time tracing using the ARM Serial Wire Viewer (SWV) interface, supporting advanced system analysis including event-, interrupt- and data tracing. Products for professional developers, like Atollic TrueSTUDIO, even provides a large number of real-time charts and graphs to visualize event tracing in an easy-to-understand manner.

- Execution time analysis using statistical profiling and the Serial Wire Viewer (SWV) interface.

- printf() redirection to the debugger GUI using the JTAG probe and the Serial Wire Viewer instrumentation trace macro cell (ITM) interface.

- Instruction tracing using ETM/ETB

- Hard fault crash analysis

- Kernel aware RTOS debugging

It is clear that to provide a commercial-grade debug solution using the gdb debugger, choosing a readymade development tool like Atollic TrueSTUDIO makes a lot of sense. Hundreds or thousands of engineering hours can easily be spent on trying to get it to work yourself, and this will most likely still provide an inferior solution with poor integration of JTAG probes, offering no device specific debugging or real-time tracing. With a commercial solution, all these issues are handled and you get an "out-of-the-box" solution that is immediately productive.

SUMMARY

The GNU tools are high-quality build- and debug tools in widespread use throughout the embedded industry, and are used in large systems like embedded Linux and Android platforms. Many individuals, academia and commercial vendors, including Atollic and ARM Ltd., extend and improve upon these tools.

As the GNU toolchain is open-source, an option is to roll-your-own tools, but this approach have a large cost in terms of acquiring the skill set and investing in a lot of engineering hours to build, test and maintain such a tool solution. Furthermore, the IDE integration and debug solution becomes less than ideal, especially when it comes to JTAG probe integration, Flash programming, device specific debugging, instruction tracing, RTOS debug, crash analysis and Serial Wire Viewer (SVW) real-time tracing.

A better solution for commercial development teams and professional developers are often to purchase a readymade tool solution like Atollic TrueSTUDIO, which provides a well-tested and finely tuned tool solution with a lot of extras, like CMSIS compliant example projects for hundreds of devices and boards, JTAG integration with Flash support, peripheral register (SFR) viewers and advanced real-time tracing capabilities.

Either way, the strong industry trends moving towards ARM-based microcontrollers, in combination with the quick adoption of GNU tools and Eclipse-based IDE’s is likely to get additional momentum in the months and years to come. These solutions are well proven, flexible and powerful, and provide a very compelling return on investment.

More information about Atollic and ***Atollic TrueSTUDIO*** is available here:

www.atollic.com

www.atollic.com/truestudio