CHAPTER

# Getting Started with the GNU Compiler Collection (gcc)

17

## CHAPTER OUTLINE


**17.1 The GNU Compiler Collection (gcc) toolchain** .......... **561**
**17.2 Typical development flow** .......... **562**
**17.3 Creating a simple blinky project** .......... **565**
**17.4 Overview of the command line options** .......... **566**
**17.5 Flash programming** .......... **567**
17.5.1 Using Keil MDK-ARM .......... 569
17.5.2 Using third-party flash programming utilities .......... 569
**17.6 Using Keil™ MDK-ARM with GNU tools for ARM Embedded Processors** .......... **570**
**17.7 Using CoIDE with GNU tools for ARM® Embedded Processors** .......... **572**
**17.8 Commercial gcc-based development suites** .......... **577**
17.8.1 Atollic TrueSTUDIO for ARM® .......... 580
17.8.2 Red Suite .......... 582
17.8.3 CrossWorks for ARM® .......... 582


## 17.1 The GNU Compiler Collection (gcc) toolchain

The GNU C Compiler is the de facto compiler choice for many open source projects. Since you can get gcc for free, it is very popular with hobbyists and academic users. Although you can built the gcc toolchain for Cortex®-M processors using the gcc source packages,[1] the build process requires an in-depth understanding of the tools. To make this easier, there are a number of pre-built packages available.

In this chapter we will cover the use of GNU Tools for ARM® Embedded Processors. You can download a pre-built package from the LaunchPad[2] website. The is package only contains the command line tools. However, you can use third-party IDE (Integrated Development Environment) tools with them. For example, you can use the GNU Tools for ARM Embedded Processors with Keil™ MDK, or CooCox CoIDE, a free IDE.

[1]You can get the packages from http://gcc.gnu.org and http://www.gnu.org/software/binutils/.
[2]At the moment this is hosted at https://launchpad.net/gcc-arm-embedded. In the long term the URL might change.

The Definitive Guide to ARM® Cortex®-M3 and Cortex-M4 Processors. http://dx.doi.org/10.1016/B978-0-12-408082-9.00017-8
Copyright © 2014 Elsevier Inc. All rights reserved.

## 17.2 Typical development flow

The gcc toolchain contains a C compiler, assembler, linker, libraries, debugger, and additional utilities. You can develop applications using C language, assembly language, or a mixture of both. The typical command names are shown in Table 17.1.

The prefix of commands reflects the type of the pre-built toolchain. In this case, the command names shown in the third column of Table 17.1 are pre-built for ARM® EABI[3] without a specific target OS platform, hence the prefix “none.” Some GNU toolchains could be created for developing applications for Linux platforms, and in those cases the prefix would be “arm-linux-.”

A typical flow of software development using gcc is shown in Figure 17.1. Unlike using the ARM Compilation toolchain (i.e., armcc), it is common for have the compile and link operations combined in one gcc run. This is easier and less error prone, as the compiler can invoke the linker automatically, generate all the required link options, and pass on all required libraries.

To compile a typical project, you will need to have the files listed in Table 17.2.

In order to make software development easier, the microcontroller vendors normally provide a set of files which include some of the items listed in Table 17.2. Sometimes these are called CMSIS-compliant device driver libraries, or microcontroller software packages. These packages might also include example projects or additional driver libraries.

For example, in a simple project that toggles LEDs on a STM32F4 Discovery board (based on the Cortex®-M4 processor), you might have the following files in your project (as shown in Figure 17.2).

**Table 17.1** Command Names (Note: the command names for toolchains from other vendors can be different)

| Tools | Generic Command Name | Command Name in GNU Tools for ARM Embedded Processors |
|---|---|---|
| C compiler | gcc | arm-none-eabi-gcc |
| Assembler | as | arm-none-eabi-as |
| Linker | ld | arm-none-eabi-ld |
| Binary file generation tool | objcopy | arm-none-eabi-objcopy |
| Disassembler | objdump | arm-none-eabi-objdump |

[3]The Embedded-Application Binary Interface (EABI) specifies standard conventions for file formats, data types, register usage, stack frame organization, and function parameter passing of an embedded software program.

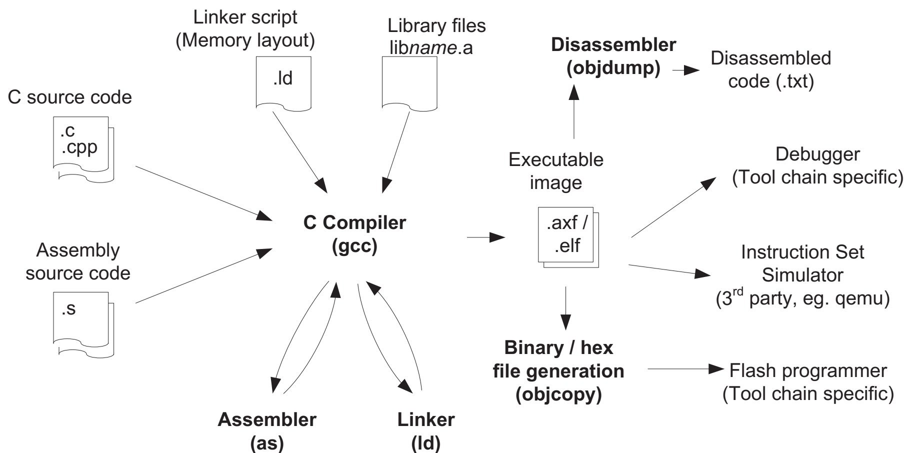



**FIGURE 17.1**

Typical program development flow

**Table 17.2** Typical Required Files for the Project

| File Type | Descriptions |
|---|---|
| Application code | Source code of your application. |
| Device-specific CMSIS Header files | The definition header files for the microcontroller you use. This is provided by the microcontroller vendor. |
| Device-specific startup code for gcc | The device specific startup code for the microcontroller you use. This is provided by the microcontroller vendor. |
| Device-specific system initialization files | This contains the SystemInit() function (system initialization) which is specified by CMSIS-Core, and additional functions for system clock updates. This is provided by the microcontroller vendor. |
| Generic CMSIS Header files | This is typically included in the device driver library package or included in tool installation. Or you can download it from ARM (www.arm.com/cmsis) |
| Linker script | The linker script is device specific. The complete linker script for a project can be composed of several files, with one file to specify the memory layout of the device and other files to define the settings required for gcc itself. The installation of GNU Tools for ARM Embedded Processors already provided an example linker script to make it easier. |
| Library files | This included the runtime libraries provided by the toolchain (typically included in the installation). You can also add additional custom libraries if needed. |

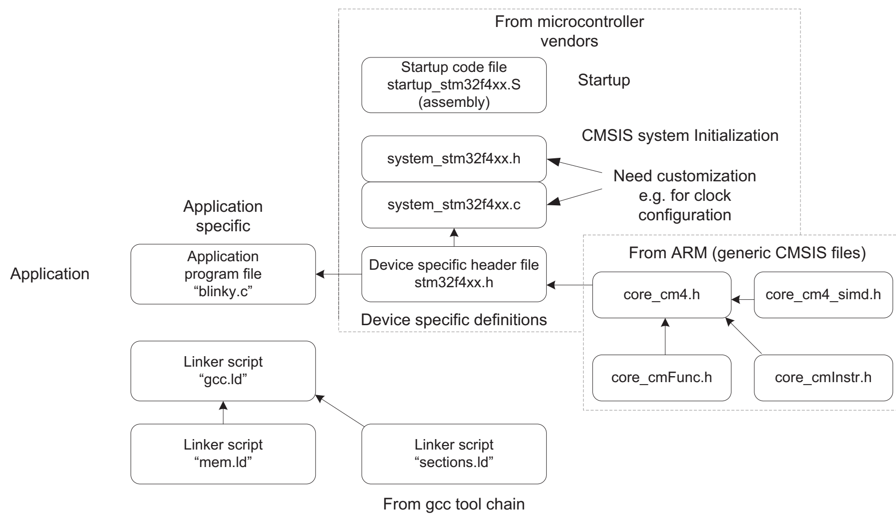



**FIGURE 17.2**

Example project with CMSIS-Core

The device-specific header file stm32f4xx.h defines all the peripheral registers so that you don't have to spend a long time creating peripheral definitions. The system_stm32f4xx.c provides the SystemInit() function that initializes the clocking system such as PLL and clock control registers.

Apart from the program files, you also need the linker script to define the memory layout of the executable image. The main linker script "gcc.ld" simply pulls in two other linker scripts:

```
/* Contents of gcc.ld */
INCLUDE "mem.ld"
INCLUDE "sections.ld"
```

- "mem.ld": This file defines the memory map (flash and SRAM) of the microcontroller you used.
- "sections.ld": This file defines the layout of information inside the executable image.

The "mem.ld" for STM32F4xx is defined as:

```
/* Specify the memory areas */
MEMORY
{
 FLASH (rx)  : ORIGIN = 0x08000000, LENGTH = 1024K
 RAM (xrw)   : ORIGIN = 0x20000000, LENGTH = 112K
}
```

The file "sections.ld" is already included in the GNU Tools for ARM Embedded Processors installation (e.g., <installation_directory>\share\gcc-arm-none-eabi\samples). You can use this file as is.

## 17.3 Creating a simple blinky project

The installation of GNU Tools for ARM® Embedded Processors only provides command line tools. You can invoke the compilation using the command line, make file (for Linux platform), batch file (for Windows platform), or using a third-party IDE. We will first demonstrate how to create a project using a batch file.

Assuming that we place the files listed in Figure 17.2 in a project directory, and the generic CMSIS include files in a subdirectory called CMSIS/Include, we can invoke the compilation and link process with a simple batch file:

```
rem Simple batch file for compiling the blinky project
rem Note the use of "^" symbol below is to allow multi-line commands
in Windows batch file.

set OPTIONS_ARCH=-mthumb -mcpu=cortex-m4
set OPTIONS_OPTS=-Os
set OPTIONS_COMP=-g -Wall
set OPTIONS_LINK=-Wl,--gc-sections,-Map=map.rpt,-lgcc,-lc,-lnosys -
ffunction-sections -fdata-sections
set SEARCH_PATH=CMSIS\Include
set LINKER_SCRIPT=gcc.ld
set LINKER_SEARCH="C:\Program Files (x86)\GNU Tools ARM Embedded\4.7
2012q4\share\gcc-arm-none-eabi\samples\ldscripts"

rem Compile the project
arm-none-eabi-gcc                         ^
   %OPTIONS_COMP% %OPTIONS_ARCH%          ^
   %OPTIONS_OPTS%                         ^
   -I %SEARCH_PATH% -T %LINKER_SCRIPT%    ^
   -L %LINKER_SEARCH%    ^
   %OPTIONS_LINK%        ^
   startup_stm32f4xx.S   ^
   blinky.c              ^
   system_stm32f4xx.c    ^
   -o blinky.axf
if %ERRORLEVEL% NEQ 0 goto end

rem Generate disassembled listing for debug/checking
arm-none-eabi-objdump -S blinky.axf > list.txt
if %ERRORLEVEL% NEQ 0 goto end
```

```
rem Generate binary image file
arm-none-eabi-objcopy -O binary blinky.axf blinky.bin
if %ERRORLEVEL% NEQ 0 goto end

rem Generate Hex file (Intel Hex format)
arm-none-eabi-objcopy -O ihex blinky.axf blinky.hex
if %ERRORLEVEL% NEQ 0 goto end

rem Generate Hex file (Verilog Hex format)
arm-none-eabi-objcopy -O verilog blinky.axf blinky.vhx
if %ERRORLEVEL% NEQ 0 goto end
```

Please note that apart from the assembly startup code files, all the other source files are identical to the blinky example in Chapter 15 and Chapter 16. The availability of CMSIS-Core enables much better software portability and reusability. Please refer to section 15.3 for detailed information about the source code.

The compilation and link process is carried out by arm-none-eabi-gcc. The rest of the compilation steps are optional. We added these steps to demonstrate how to create a binary file, hex file, and disassembled listing file.

## 17.4 Overview of the command line options

The GNU Tools for ARM® Embedded Processors can be used with a wide range of ARM processors, including Cortex®-M processors and Cortex-R processors. In the example in section 17.2 we used Cortex-M4 (without a floating point unit). You can specify which target processor is to be used and/or which architecture is to be used.

Table 17.3 lists target processor command line options.

Table 17.4 lists target architecture command line options.

Some of the other commonly used options are listed in Table 17.5.

By default the GNU C compiler uses a run-time library called Newlib. This library provides very good performance, but at the same time has larger code size. In version 4.7 of the GNU Tools for ARM Embedded Processors a new feature called Newlib-nano was introduced. It is optimized for size and can produce much smaller binary code. For example, with standard Newlib the blinky (binary image file) is 3700 bytes, and this reduced to just 1536 bytes when Newlib-nano is used.

There are a couple of areas that need attention when using Newlib-nano:

- Please note that --specs=nano.specs is a linker option. You must include this option in the linker option if the compiling and linking stages are separated.

**Table 17.3** Compilation Target Processor Command Line Options

| Processor | GCC Command Line Option |
|---|---|
| Cortex-M0+ | -mthumb -mcpu=cortex-m0plus |
| Cortex-M0 | -mthumb -mcpu=cortex-m0 |
| Cortex-M1 | -mthumb -mcpu=cortex-m1 |
| Cortex-M3 | -mthumb -mcpu=cortex-m3 |
| Cortex-M4 (no FPU) | -mthumb -mcpu=cortex-m4 |
| Cortex-M4 (soft FP) | -mthumb -mcpu=cortex-m4 -mfloat-abi=softfp -mfpu=fpv4-sp-d16 |
| Cortex-M4 (hard FP) | -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16 |

**Table 17.4** Compilation Target Architecture Command Line Options

| Architecture | Processor | GCC Command Line Option |
|---|---|---|
| ARMv6-M | Cortex-M0+, Cortex-M0, Cortex-M1 | -mthumb -march=armv6-m |
| ARMv7-M | Cortex-M3 | -mthumb -march=armv7-m |
| ARMv7E-M (no FPU) | Cortex-M4 | -mthumb -march=armv7e-m |
| ARMv7E-M (soft FP) | Cortex-M4 | -mthumb -march=armv7e-m -mfloat-abi=softfp -mfpu=fpv4-sp-d16 |
| ARMv7E-M (hard FP) | Cortex-M4 | -mthumb -march=armv7e-m -mfloat-abi=hard -mfpu=fpv4-sp-d16 |

- Formatted input/output of floating point numbers are implemented as weak symbols. When using %f in printf or scanf, you have to pull in the symbol by explicitly specifying the "-u" command option:

  ```
  -u _scanf_float
  -u _printf_float
  ```

  For example, to output a float, the command line is:

  ```
  $ arm-none-eabi-gcc --specs=nano.specs -u _printf_float
  $(OTHER_OPTIONS)
  ```

## 17.5 Flash programming

After the program image has been generated, we need to test it by downloading the image into the flash memory of the microcontroller for testing. However, the GNU

**Table 17.5** Commonly Used Compilation Switches

| Options | Descriptions |
|---|---|
| "-mthumb" | Specifies Thumb instruction set |
| "-c" | Compile or assemble the source files, but do not link. Object file is generated for each source file. This is used when you have a project setup that separates compile and link stages. |
| "-S" | Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified. |
| "-E" | Stop after the preprocessing stage. The output is in the form of preprocessed source code, which is sent to the standard output. |
| "-Os" | Optimization level - It can be from optimization level 0 ("-O0") to 3 ("-O3"), or can be "-Os" for size optimization. |
| "-g" | Include debug information |
| "-D<*macro*>" | User defined preprocessing macro |
| "-Wall" | Enable all warnings |
| "-I <directory>" | Include directory |
| "-o <output file>" | Specify output file |
| "-T <linker script>" | Specify linker script |
| "-L <ld script path>" | Specify search path for linker script |
| "-Wl,option1,option2" | "-Wl" passes options to linker. It can provide multiple options, separate by commas. |
| "--gc-sections" | Remove sections that are not used. Be careful with this option because it could also remove sections that are indirectly referenced. You can check linker map report to see what is removed and use KEEP() function in the linker script to ensure that certain data/code are not removed. |
| "-lgcc" | Link against libgcc.a |
| "-lc" | Instructs the linker to search in the system-supplied standard C library for functions not supplied by your own source files. This is the default choice, and is opposition of the "-nostdlib" option which force the linker NOT to search in the system-supplied libraries. |
| "-lnosys" | Specific no semihosting (use libnosys.a for linking). If semihosting is required, for example, using RDI monitor for semihosting support, you can use "--specs=rdimon.specs -lrdimon." |
| "-lm" | Link with math library |
| "-Map=map.rpt" | Generate map report file (map.rpt is the filename of the report) |
| "-ffunction-sections" | Put every function in its own section. Use with "--gc-sections" to reduce code size. |
| "-fdata-sections" | Put each data in its own section. Use with "--gc-sections" to reduce code size. |

| Table 17.5 Commonly Used Compilation Switches—*Cont'd* | |
|---|---|
| **Options** | **Descriptions** |
| "--specs=nano.specs" | Use Newlib-nano runtime library (introduced in version 4.7 of GNU Tools for ARM Embedded Processors). |
| "-fsingle-precision-constant" | Treat a floating point constant as single precision constant instead of implicitly converting it to double precision |

Tools for ARM® Embedded Processors do not include any flash programming support, so you need to use third-party tools to handle the flash programming. There are a number of options, discussed in the following sections.

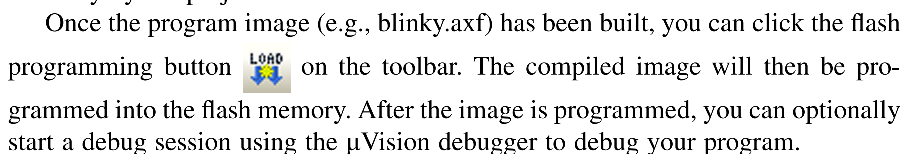
### 17.5.1 Using Keil MDK-ARM

If you have access to Keil™ MDK-ARM and a supported debug adaptor (e.g., ULINK2, or if the development board has a supported debug adaptor), you can use the flash programming feature in Keil MDK-ARM to program the image created into the flash memory.

To use Keil MDK-ARM to program your program image, the file extension of the executable needs to be changed to .axf.

The next step is to create a μVision project in the same directory (typically the project name should be the same as the executable, e.g., "blinky"). In the project creation wizard, select the microcontroller device you use. There is no need to add any source file to the project. When the project wizard asks whether it should copy the default startup code, you should select "no" to prevent the original startup file for gcc from being overwritten.

Set up the debug options to use your debug adaptor (for debug and flash programming; see Chapter 15). By default the flash programming algorithm should be set up correctly by the project creation wizard.

Once the program image (e.g., blinky.axf) has been built, you can click the flash programming button  on the toolbar. The compiled image will then be programmed into the flash memory. After the image is programmed, you can optionally start a debug session using the μVision debugger to debug your program.

### 17.5.2 Using third-party flash programming utilities

There are many different flash programming utilities available. A common one is the CoFlash from coocox.org. This flash programming tool supports Cortex®-M microcontrollers from a number of major microcontroller vendors and a number of debug adaptors.

When CoFlash is started, it first displays the Config tab. Set up the microcontroller device and debug adaptor as required. Figure 17.3 shows the configurations used with the STM32F4Discovery board.

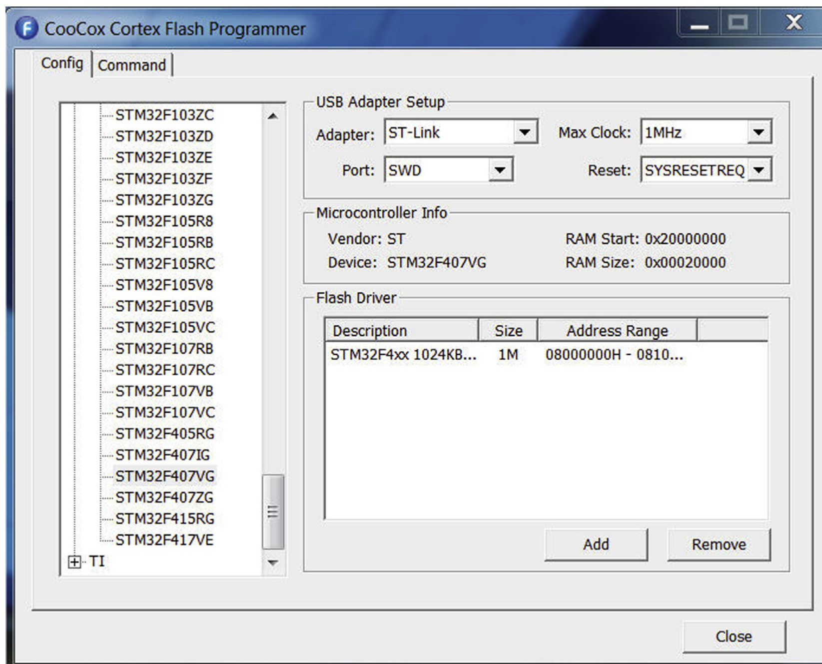


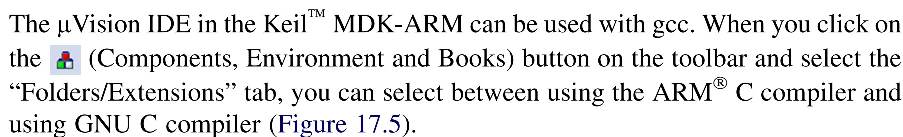

**FIGURE 17.3**

CoFlash configuration screen for STM32F4 Discovery board

Then switch to the command tab (Figure 17.4), where you can select the program image (can be binary or executable image ."elf"), and then you can click on the "Program" button to start the flash programming.

Use a third-party IDE together with GNU Tools for ARM® Embedded Processors. See section 17.6 on using Keil™ MDK, and section 17.7 on using CoIDE.

## 17.6 Using Keil™ MDK-ARM with GNU tools for ARM Embedded Processors

The µVision IDE in the Keil™ MDK-ARM can be used with gcc. When you click on the  (Components, Environment and Books) button on the toolbar and select the "Folders/Extensions" tab, you can select between using the ARM® C compiler and using GNU C compiler (Figure 17.5).

Once the toolchain path is set up, you can then add your program files to the projects by using the Keil MDK normally. Some of the project settings such as debug, trace, and flash programming are the same as the normal MDK environment. However, other project setting dialogs are different and are GNU toolchain specific.

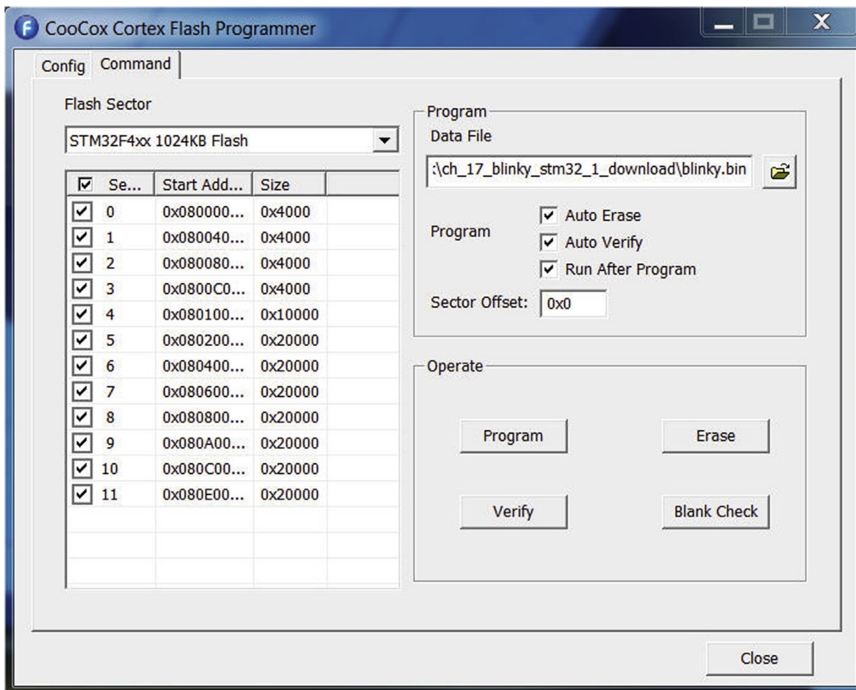



**FIGURE 17.4**

CoFlash command screen for STM32F4 Discovery board

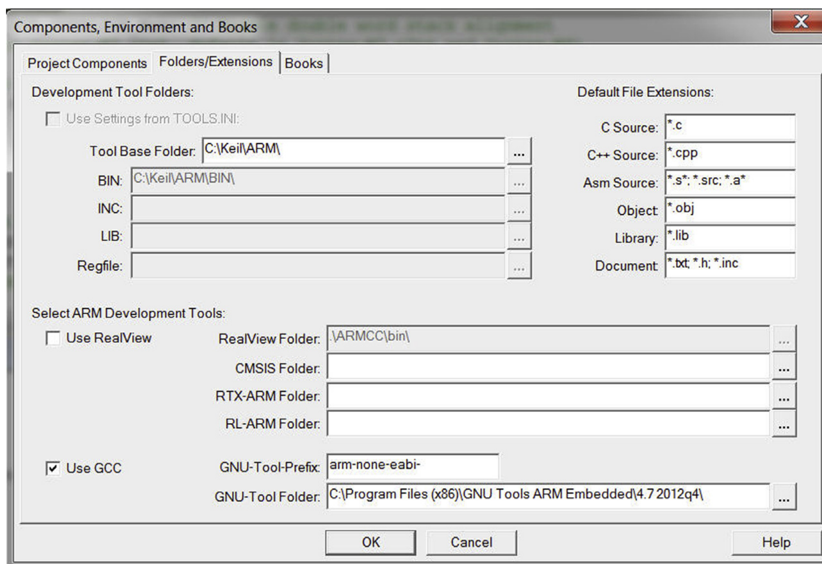



**FIGURE 17.5**

Keil MDK-ARM support for the use of GNU toolchain

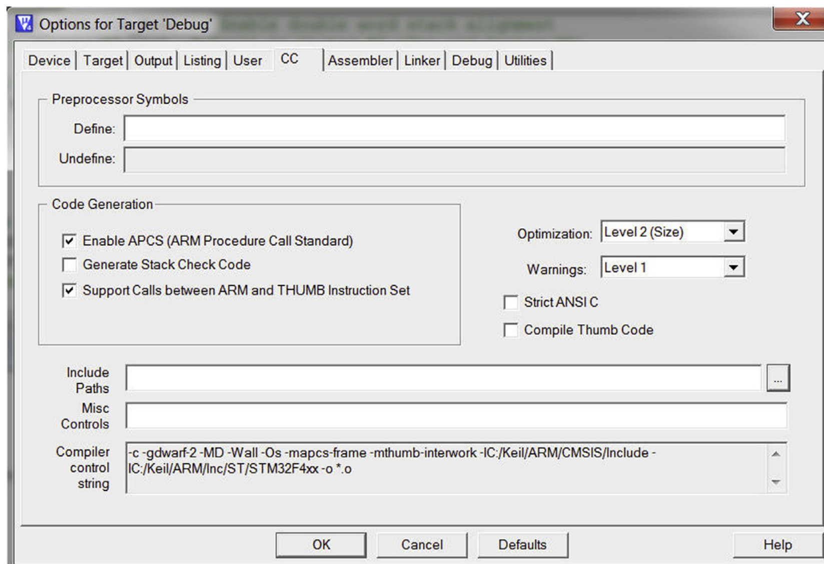



**FIGURE 17.6**

C compiler settings

For example, the C compile option settings (Figure 17.6) are different from the options available for the ARM C compiler (Figure 15.24). Assembler options are shown in Figure 17.7 and linker options are shown in Figure 17.8.

Please note that the generic CMSIS-Core include files do not need to be added to the project because µVision automatically adds the location of the CMSIS-Core files in the Keil MDK-ARM installation to the include path.

Once the project is compiled, you can download and debug the application. Note that some of the source-level debugging features might not be available in this environment.

## 17.7 Using CoIDE with GNU tools for ARM® Embedded Processors

The CoIDE is a popular choice for many users of the GNU toolchain. You can download it from the CooCox website (http://www.coocox.org). It supports a good number of the current Cortex®-M microcontrollers on the market. The CoIDE does not include the GNU toolchain, so the GNU toolchain still needs to be downloaded and installed separately.

After installing the GNU toolchain, and then CoIDE, the first step is to set up the GNU toolchain path in CoIDE. This can be done by accessing the "Select Toolchain Path" from the pull-down menu (Project → Select Toolchain Path) (see Figure 17.9).

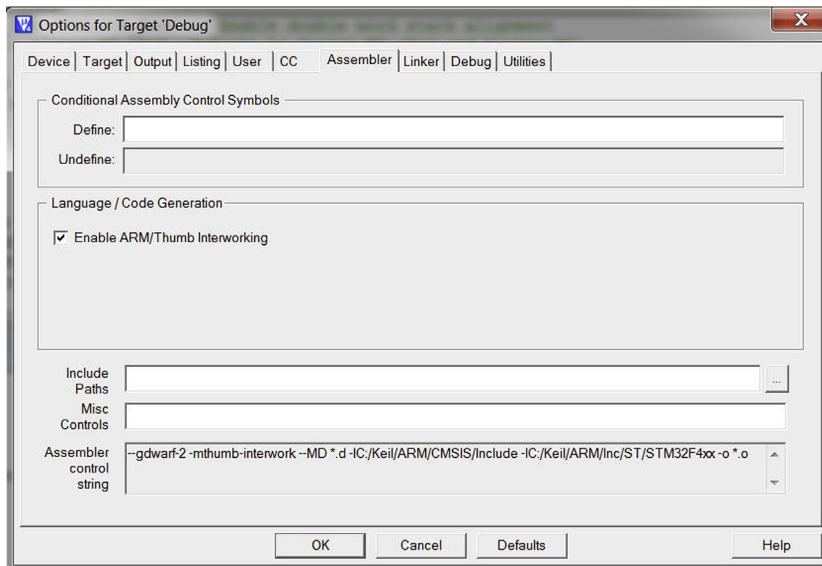



**FIGURE 17.7**

Assembler settings

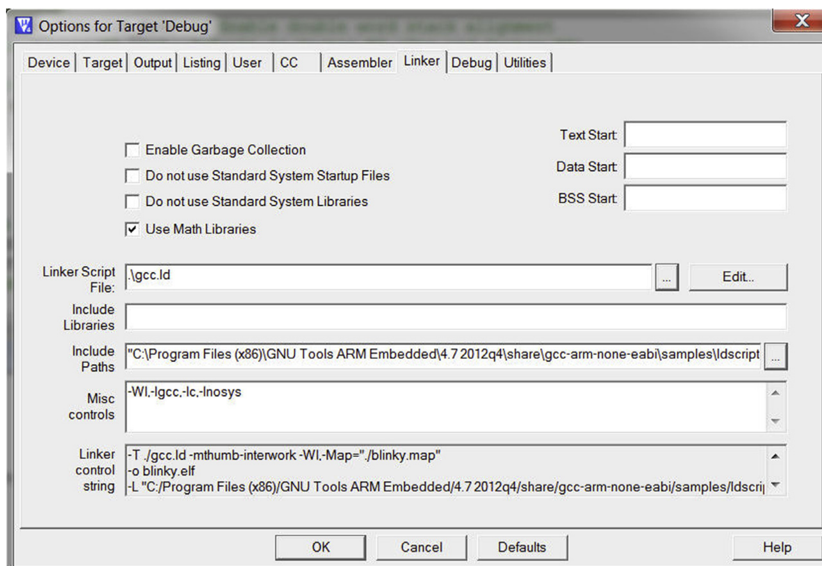



**FIGURE 17.8**

Linker settings

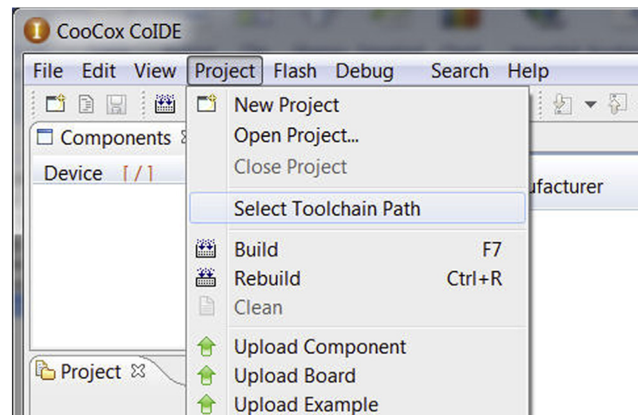



**FIGURE 17.9**

Select toolchain path

For example, in the system with GNU Tools for ARM Embedded Processors version 4.7, the selected path location is: C:\Program Files (x86)\GNU Tools ARM Embedded\4.7 2012q4\bin.

The process of setting up a new project is very easy. Many of the steps are shown as on-screen instructions. Here we will reuse the blinky project for STM32F4 Discovery board that we developed on the Keil™ MDK-ARM and IAR Embedded Workbench. To use this board with CoIDE, we also need to download and install the device driver for the ST-Link v2 debug adaptor. This can be downloaded from ST website.[4] The device driver is also included in the Keil MDK-ARM installation.[5]

The first step is to select the microcontroller vendor (Figure 17.10), and the second step is to select the microcontroller device (Figure 17.11).

Step 3 looks a bit more complex. The GUI presents a list of software components that you can include in your project. At minimum you will need the boot code. When you click on any component, a new dialog appears to ask you to select the location and name of the project (Figure 17.12).

Click on "yes" and create your project in a suitable folder. Afterwards we can add additional software components for this project. For the blinky project we select the C Library, Cortex-M4 CMSIS-Core, and CMSIS BOOT components (Figure 17.13).

Now you have a minimal project with just an endless loop in main.c. The rest of the project, like the boot code, CMSIS-Core header files, and device-specific system initialization files, have already been added to the project. You can examine the files in the project using the project browser on the left bottom corner of the screen. You

[4]http://www.st.com/internet/evalboard/product/251168.jsp (bottom of the Design Support tab)
[5]Typically located in C:\Keil\ARM\STLink.

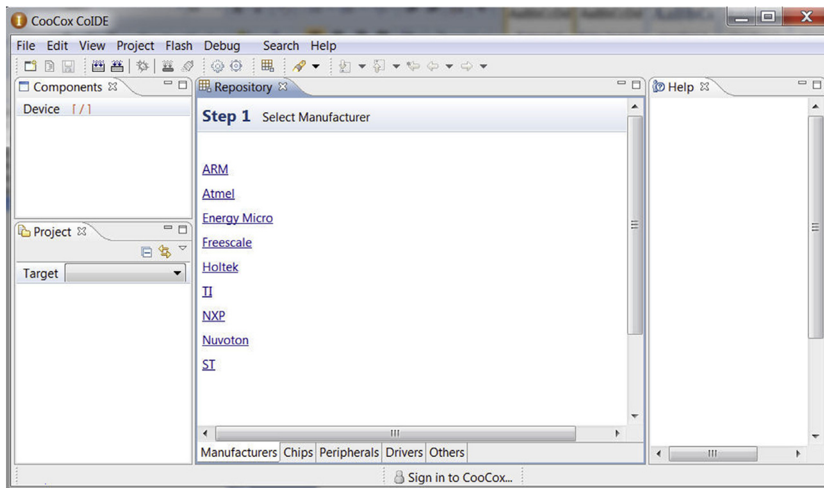



**FIGURE 17.10**

Step 1 — Select manufacturer

can also add additional files to the project by right-clicking on the project and selecting "Add files."

We can now copy the contents of the blinky.c we used in the previous example into main.c. A few additional edits of the source code are needed. First, the startup

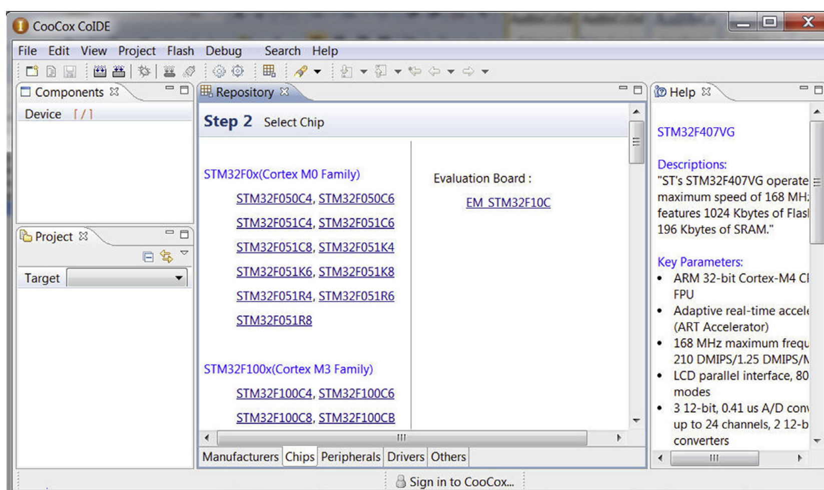



**FIGURE 17.11**

Step 2 — Select chip

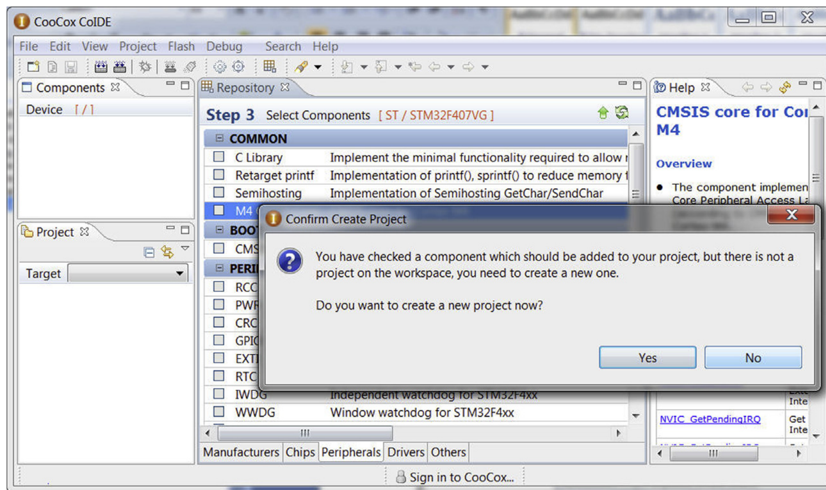



**FIGURE 17.12**

Step 3 — Select components

code (startup_stm32f4xx.c) does not include the call to SystemInit(), so we need to add this either in the startup code or in the beginning of the main.c.

Secondly, for the case of the STM32F4 Discovery board, we need to edit the system initialization code (system_stm32f4xx.c) to set the PLL_M parameter to 8 (as the board is using an 8MHz crystal). For other microcontroller boards you might also need to adjust the system clock settings accordingly.

In this example we add SystemInit() call in the beginning of main.c (line 11) and include system_stm32f4xx.h (line 2):

```
Modification of main.c (line and line 11)
 1:#include "stm32f4xx.h"
 2:#include "system_stm32f4xx.h"
 3:
 4:void Delay(uint32_t nCount);
 5:
 6:int main(void)
 7:{
 8:    SCB->CCR |= SCB_CCR_STKALIGN_Msk; // Enable double-word stack
       alignment
 9:    //(recommended in Cortex-M3 r1p1, default in Cortex-M3 r2px and
       Cortex-M4)
10:
11:    SystemInit();
       ...
```

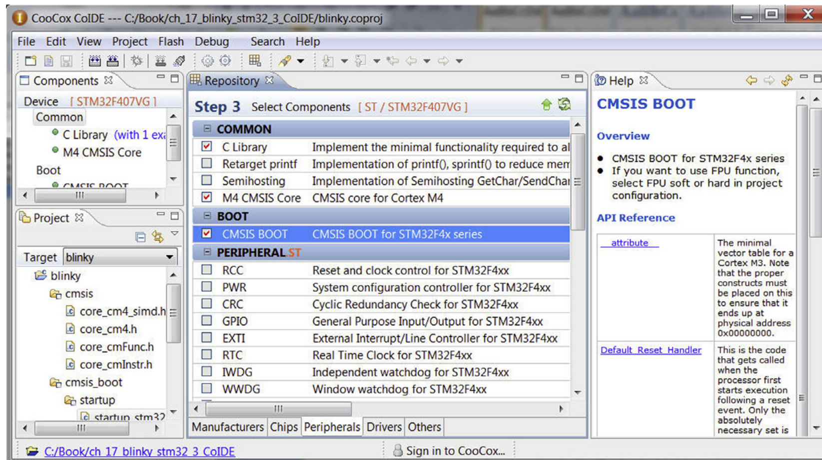



**FIGURE 17.13**

Step 3 — Components selected

Before we compile the project, we can review and modify some of the project settings such as optimization level and debug adaptor settings. These can be accessed by clicking on the "Configuration" icon on the toolbar (Figure 17.14), or by right-clicking on Blinky in the project window and selecting "Configuration."

Once the project settings are adjusted (e.g., optimizations, debug adaptor), we can compile the project using one of the following methods:

- Pull-down menu: "Project → Build,"
- Hot key F7, or
- Clicking on the "Build" button on the toolbar.

The compilation should complete with the display shown in Figure 17.15. When the compilation process is completed, we can start the debug session by clicking on the "Start Debug" icon on the toolbar, or by using Ctrl-F5 to start the debugger (Figure 17.16). The debugger screen has additional icons for debug operations (Figure 17.17).

## 17.8 Commercial gcc-based development suites

While you can use free versions of gcc to develop and debug your applications, there are a number of commercial development toolchains based on gcc that often offer a lot of additional features. In addition, using commercial toolchains gives you product support services that you do not get with free toolchains. For example, if a bug is found in a part of the toolchain, a commercial tool vendor can often develop a fix for you, but with free toolchains you do not have such an advantage. This is often critical for project development.

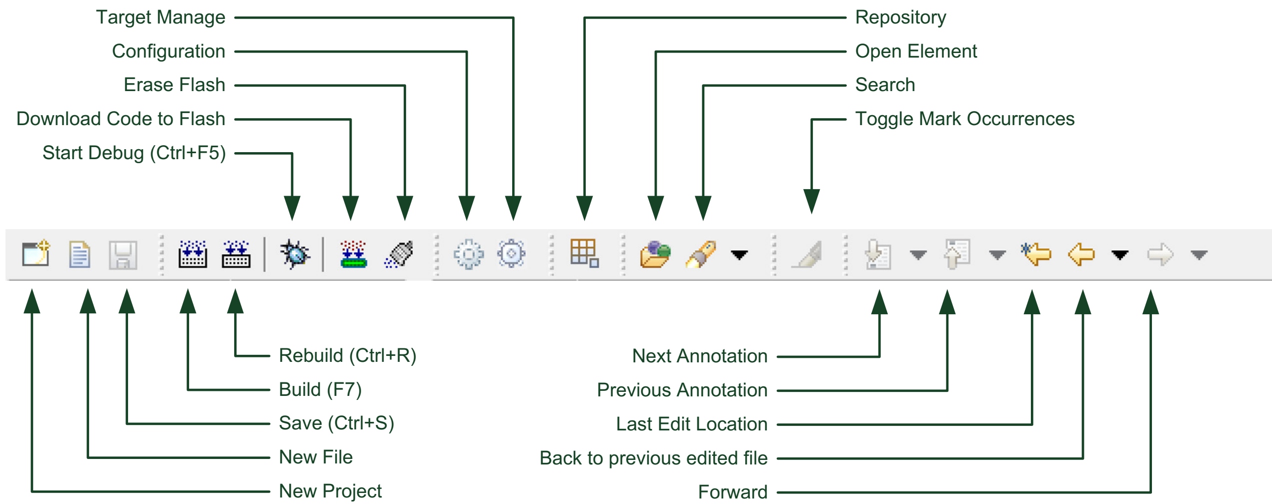



**FIGURE 17.14**

Icons on the CoIDE toolbar

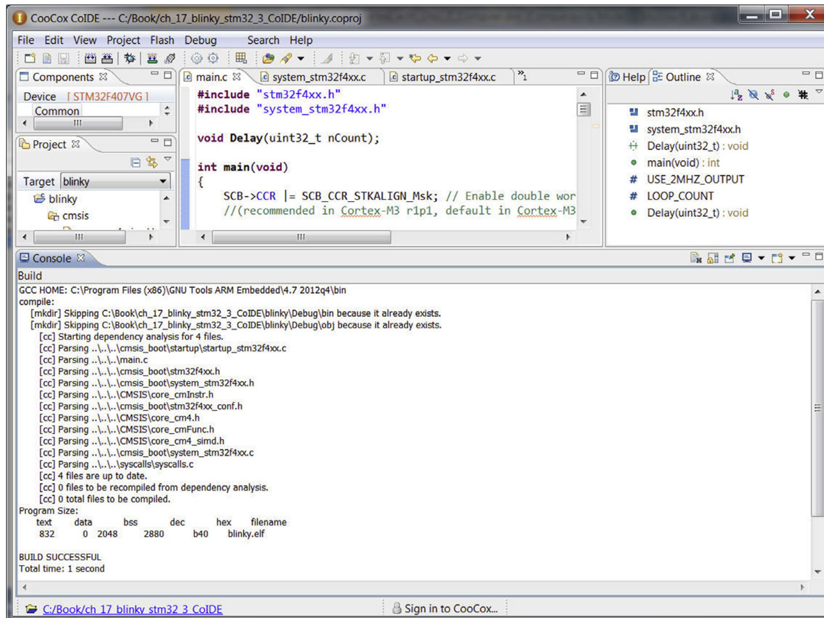



**FIGURE 17.15**

Compilation complete message

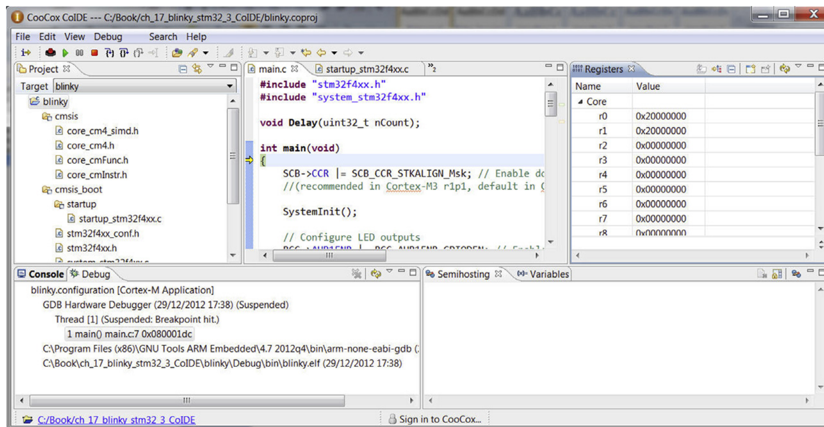



**FIGURE 17.16**

Debugger screen

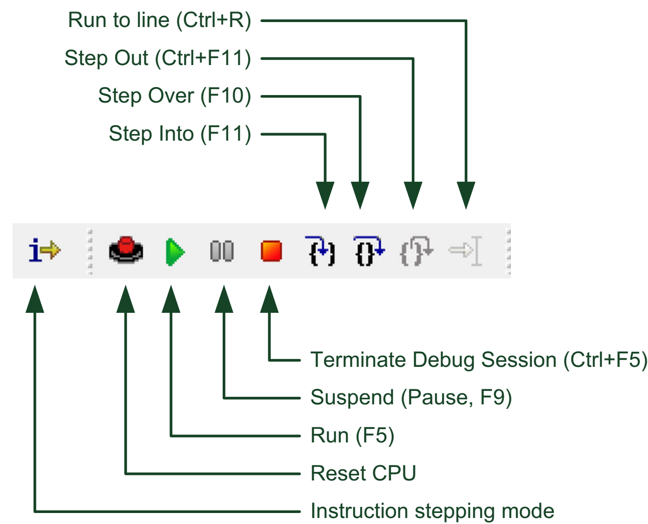



**FIGURE 17.17**

Icons on the debugger toolbar

### 17.8.1 Atollic TrueSTUDIO for ARM®

Atollic TrueSTUDIO for ARM is one of the commercial development suites that is based on the gcc and ECLIPSE IDE. The TrueSTUDIO product provides a complete solution for the majority of users:

- GNU toolchain including compiler, linker, etc.
- Eclipse-based IDE and project management
- Automatic linker script generation and easy-to-use project development flow
- Supports over 1100 ARM microcontroller devices, including flash programming support
- Example projects for over 80 development boards, with over 1000 example projects via Atollic TrueSTORE, a system that enables TrueSTUDIO users to download, install, and compile examples from a repository with just one mouse click
- Basic debugging, can be used with:
  - Segger J-Link
  - ST-LINK from STMicroelectronics (e.g., STM32F4 Discovery board)
  - Third-party gdbserver
  - OSJTAG
  - P&E Multilink probes
- Cortex®-M3/Cortex-M4 real-time trace with Serial Wire Viewer (SWV)
  - Data trace, with real-time data timeline oscilloscope and history log
  - Event trace (e.g., exception history)

    - Instrumentation Trace (e.g., using SWV for printf via Instrumentation Trace Macrocell, ITM)
- Advanced debug features
    - OS-aware debugging for most popular OSs
    - Execution profiling with statistical PC sampling and SWV trace
    - Multi-core debug support

Utilizing the SWV feature in the Cortex-M3 and Cortex-M4 processors, True-STUDIO provides a wide range of analysis features such as real-time "animated" timeline charts (where the charts scrolls automatically in real-time with execution/tracing progress). For example, it can visualize variable changes or changes in memory locations over a period of time.

In addition, TrueSTUDIO also provides various features for project management such as:

- Integrated source code reviews
- Integrated bug database clients (support Bugzilla, Trac, etc.)
- Integrated version control system client (e.g., GIT, subversion, CVS)
- Integrated Fault Analysis

More demanding users can extend TrueSTUDIO by adding optional add-on products including:

- Atollic TrueINSPECTOR – A static source code analysis tool, which can detect potential coding problems. It also allows you to check your code for MISRA-C (2004) compliance, and provides code complexity analysis.
- Atollic TrueANALYZER – A dynamic test tool for measuring various coverage metrics. It can highlight which parts of the program have not been tested, including untested conditions in conditional code.
- Atollic TrueVERIFIER – A software test automation tool that can analyze the program code and generate a test suite for various functions inside the code. It also automatically compiles, downloads, and executes the test suite to the target board.

Similar to other commercial development suite vendors, Atollic also provides a cut-down version of TrueSTUDIO for free. The Atollic TrueSTUDIO for ARM Lite[6] is limited to 32KB code size for ARMv7-M, and limited to 8KB code size for ARMv6-M (i.e., Cortex-M0, Cortex-M0+, Cortex-M1). This free version provides almost all of the features available in the professional version including IDE, compiler, and debugger (which includes advance debug features such as SWV real-time trace).

If you are interested in trying out the Atollic TrueSTUDIO, on the Atollic website there is a whitepaper called "Embedded development using the GNU toolchain for ARM processors,"[7] which might be a good starting point.

[6]http://www.atollic.com/index.php/download/truestudio-for-arm
[7]http://www.atollic.com/index.php/whitepapers

### 17.8.2 Red Suite

Red Suite from Code Red Technologies (http://www.code-red-tech.com, recently acquired by NXP) is a fully featured development suite for ARM®-based microcontrollers, which includes all the tools necessary to develop high-quality software solutions in a timely and cost-effective fashion. It provides a comprehensive C/C++ programming environment, and the Red Suite IDE is based on the popular Eclipse IDE with many ease-of-use and microcontroller-specific enhancements, like syntax-coloring, source formatting, function folding, online and offline integrated help, extensive project management automation, and integrated source repository support (CVS integrated or subversion via download).

It contains the following features:

- Wizards that create projects for all supported microcontrollers
- Automatic linker script generation including support for microcontroller memory maps
- Direct download to flash when debugging
- Inbuilt Flash programmer
- Built-in datasheet browser
- Support for Cortex®-M, ARM7TDMI™, and ARM926-EJ based microcontrollers

With Cortex-M3 and Cortex-M4 based microcontrollers, Red Suite can take advantage of its advanced features, including full support for Serial Wire Viewing (SWV) through a feature called Red Trace. Red Trace enables a high level of visualization of what is happening in the target device.

The debugger includes a peripheral viewer that provides complete visibility of all registers and bit fields in all target peripherals in a simple tree-structured display. A powerful processor-register viewer is provided which gives access to all processor registers and provides smart formatting for complex registers such as flags and status registers.

In addition, Code Red Technology also provides a free version of the GNU toolchain called LPCXpresso, which works with NXP LPCXpresso development boards.

### 17.8.3 CrossWorks for ARM®

CrossWorks for ARM® is a C, C++, and assembly development suite from Rowley Associates (http://www.rowley.co.uk/arm/index.htm). It contains an IDE called CrossStudio which integrates the GNU toolchain. The source-level debugger in CrossStudio can work with a number of debug adaptors including CrossConnect for ARM (from Rowley Associates) and third-party in-circuit debugger hardware such as the SEGGER J-Link and Amontec JTAGkey.

CrossWorks for ARM is available in various editions, including non-commercial, low-cost packages (personal and educational licenses).