atollic

Atollic TrueSTUDIO® for STMicroelectronics® STM32™

Quickstart Guide

COPYRIGHT NOTICE

TRADEMARK

DISCLAIMER

DOCUMENT IDENTIFICATION

TS-QSG-ARM Sept 2011

REVISION

7th	April 2011 – Minor changes to texts
8th	September 2011 – Updated for TrueSTUDIO® v2.2
9th	December 2011 – Updated for TrueSTUDIO® v2.3

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.
115 Route 46
Building F, Suite 1000
Mountain Lakes, NJ 07046-1668
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

Figures

Tables

INTRODUCTION

Welcome to the ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™*** Quickstart Guide. The purpose of this document is to help you get started with ***Atollic TrueSTUDIO®.*** The following products are bundled with ***Atollic TrueSTUDIO®*** and are described too:

- ***Atollic TrueINSPECTOR®*** – static source code inspection
- ***Atollic TrueANALYZER®*** – dynamic test- and code coverage analysis
- ***Atollic TrueVERIFIER®*** – embedded test automation

Commercial buyers of ***Atollic TrueSTUDIO®*** get a one-week fully working license of ***Atollic TrueINSPECTOR®***, ***Atollic TrueANALYZER®*** and ***Atollic TrueVERIFIER®***.

WHO SHOULD READ THIS GUIDE

This document is primarily intended for embedded systems developers who want to quickly get started with using the ***Atollic TrueSTUDIO®***, ***Atollic TrueINSPECTOR®, Atollic TrueANALYZER®*** and ***Atollic TrueVERIFIER®*** products, for development and testing of embedded applications for the STM32™ family of microcontrollers.

DOCUMENT CONVENTIONS

The text in this document is formatted to ease understanding and provide clear and structured information on the topics covered.

TYPOGRAPHIC CONVENTIONS

This document has the following typographic conventions:

Style	Use
`Computer`	Keyboard commands or source code.
Object names	Names of user interface objects (such as menus, menu commands, buttons and dialog boxes) that appear on the computer screen.
Cross references	A cross reference in this document or to other external documents.
Product name	Atollic company products.

	Identifies instructions specific to the graphical user interface (GUI).
	Identifies instructions specific to the command line interface (CLI).
	Identifies help tips and hints.
	Identifies a caution.

Table 1 – Typographical conventions

Section 1. GETTING STARTED

This section provides information on how to start to use ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™***. It covers information on the following topics:

- Before you start
- Starting the program
- Creating a new project
- Configuring the project
- Building the project
- Debugging

BEFORE YOU START

Atollic TrueSTUDIO® is built using the ECLIPSE™ framework, and thus inherits some characteristics that may be unfamiliar to new users. The following sections outline important information to users without previous experience with ECLIPSE™.

WORKSPACES & PROJECTS

As ***Atollic TrueSTUDIO®*** is built using the ECLIPSE™ framework. It inherits its project and workspace model. The basic concept is outlined here:

- A workspace contains projects. Technically, a workspace is a directory containing project directories.
- A project contains files. Technically, a project is a directory containing files (that may be organized in sub-directories).
- Project directories cannot be located outside a workspace directory, and project files can generally not be located outside its project directory.
- There can be many workspaces on your computer at various locations in the file system, and every workspace can contain many projects.
- Only one workspace can be active at the same time, but you can switch to another workspace at any time.
- You can access all projects in the active workspace at the same time, but you cannot access projects that are located in a different workspace.
- Switching workspace is a quick way of shifting work from one set of projects to another set of projects. It will trigger a quick restart of the product.

In practice, this creates a very structured hierarchy of workspaces with projects that contains files.

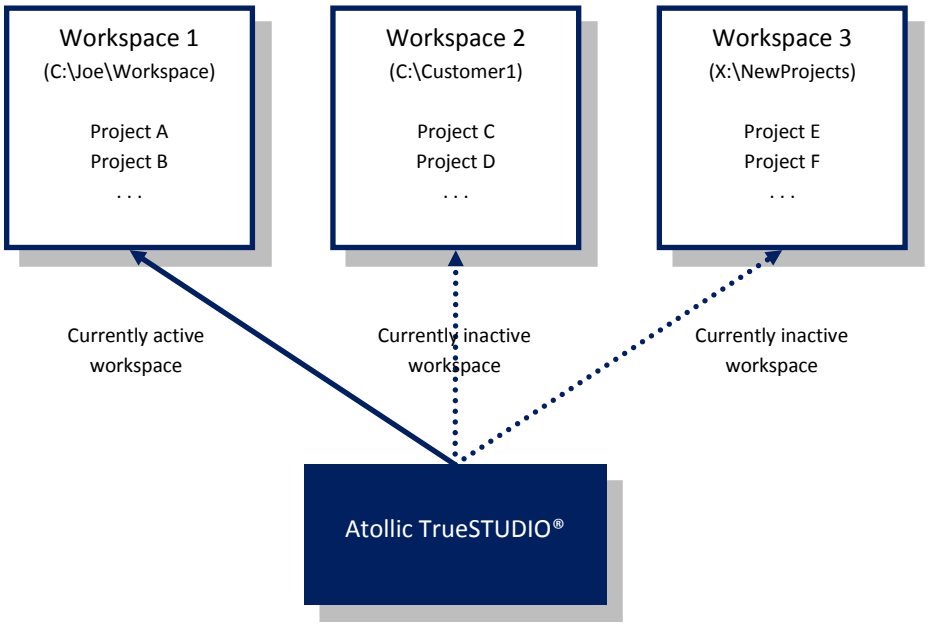


Figure 1 - Workspaces and projects

PERSPECTIVES & VIEWS

Atollic TrueSTUDIO® is a very powerful product, and some of its versions include a large number of docking views packed with features. If all docking views were displayed at the same time, developers would be overloaded with information from docking views that may not be relevant to the current work task.

To solve this problem, docking views can be organized in perspectives; where a perspective contains a number of predefined docking views. A perspective typically handles one work task, such as:

- C/C++ code editing
- Debugging
- Bug database
- Version control system
- Code coverage analysis
- etc.

As an example, the **C/C++** code editing perspective display docking views that relate to code editing (such as editor outline, class browser and so on), the **Debug** perspective display docking views that relate to debugging (breakpoints, CPU registers and so on).

Switching from one perspective to another is just a quick way to hide some docking views and display some other docking views.

Atollic TrueSTUDIO® comes with a number of ready-made perspectives, but developers can modify these, or create entirely new ones, as desired.

To switch to another perspective, select the **Window, Open Perspective** menu command

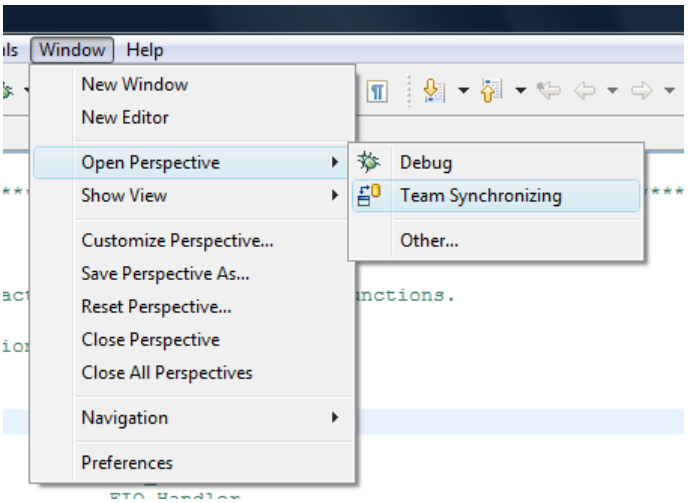


Figure 2 - Switch to another perspective

Alternatively, click any of the perspective buttons to the top right corner of the main window.

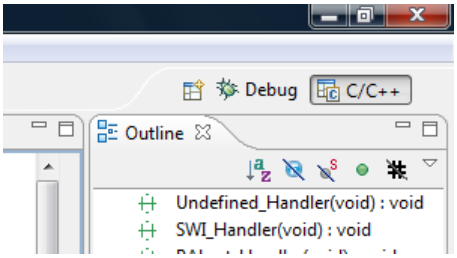


Figure 3 - Switch to another perspective

When ***Atollic TrueSTUDIO®*** is started the first time, the **C/C++** source code editing perspective is activated by default. This perspective (like other perspectives) does not show all relevant docking views by default, to reduce information over-load.

To get access to more features which are in fact built into the product, open additional docking views to access those features.

To open additional docking views, select the **Window, Show View, Other…** menu command.

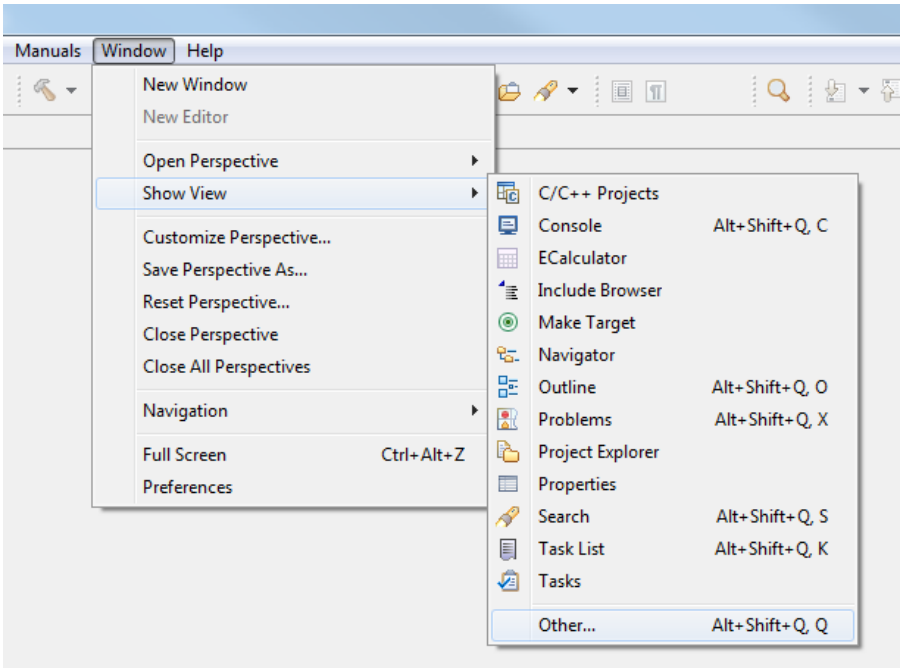


Figure 4 - Show View menu command

The **Show View** dialog box is now opened. Double click on any docking view to open it and get access to additional features.

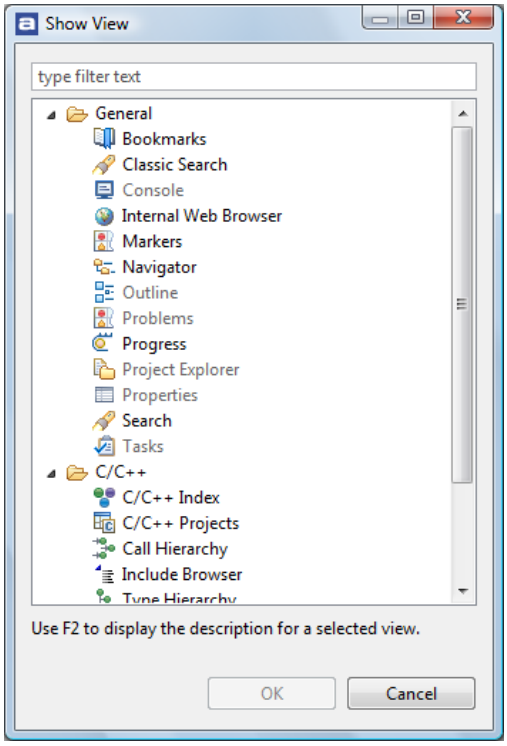


Figure 5 - Show View dialog box

STARTING THE PROGRAM

After installing ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™*** on your computer, start the program by performing the following steps (on Microsoft® Windows® Vista® and Windows 7®):

1. Open the Microsoft® Windows® **Start menu**
2. Click on **All Programs**
3. Open the **Atollic** folder
4. Open the **TrueSTUDIO® *for STMicroelectronics® STM32™*** product folder
5. Click on the **Atollic TrueSTUDIO®** product name

The program is then started and query for the **Workspace** location (all projects in **Atollic TrueSTUDIO® *for STMicroelectronics® STM32™*** are stored in workspaces).

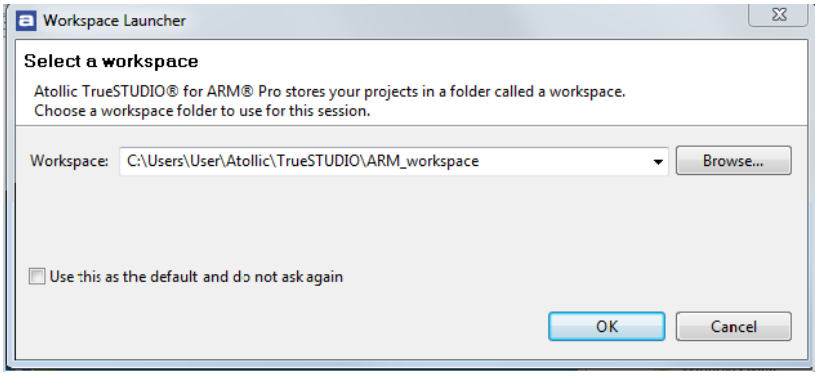


Figure 6 - Workspace launcher

Select the folder that will contain your projects and click on the **OK** button.

You need to have write access to your home directory to be able to start ***Atollic TrueSTUDIO®.***

The **Information Center** is displayed. Browse the information using the hypertext links as appropriate.

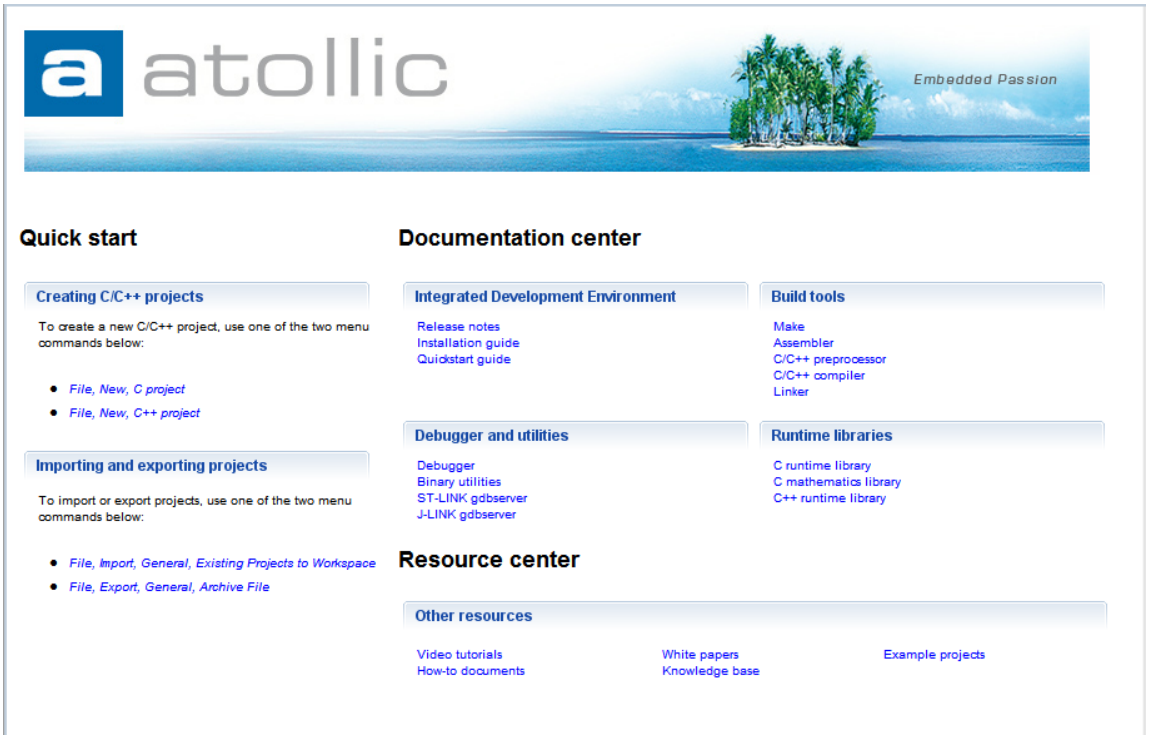


Figure 7- Information Center

When you want to start using ***Atollic TrueSTUDIO®***, click on the **Start using TrueSTUDIO** link. The **Welcome** window is removed, but can be opened again later using the **Help, Welcome** menu command.

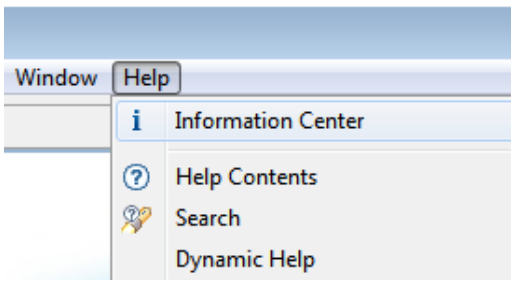


Figure 8 – Information Center menu command

CREATING A NEW PROJECT

Atollic TrueSTUDIO® for STMicroelectronics® STM32™ *supports* both managed and unmanaged projects. Managed projects are completely handled by the IDE and can be configured using GUI settings, whereas unmanaged projects require a makefile that has to be maintained manually.

To create a new managed mode C project, perform the following steps:

1. Select the **File, New, C Project** menu command to start the ***Atollic TrueSTUDIO®*** project wizard.

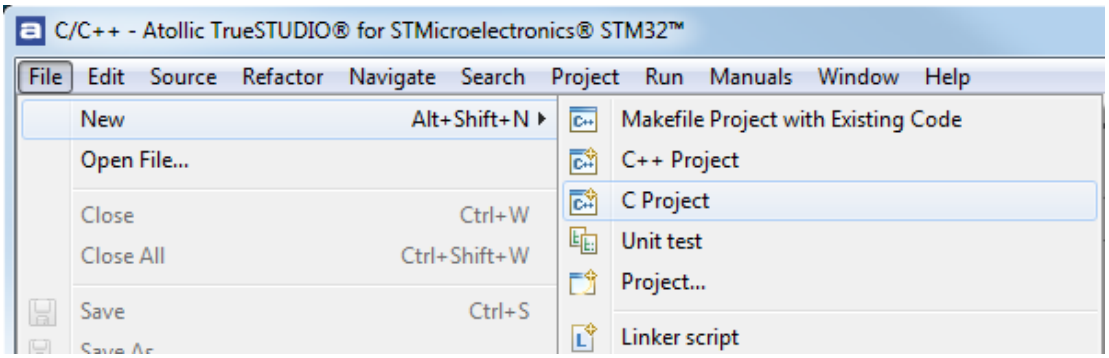


Figure 9 - Starting the project wizard

2. The **C Project** configuration page is displayed.

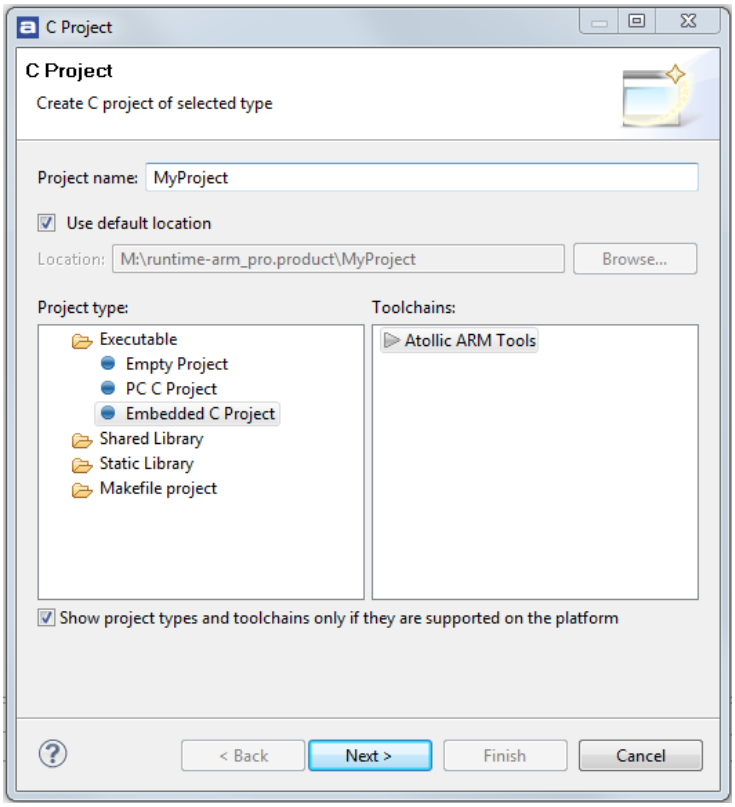


Figure 10 - C project

Enter a **Project name** (such as "MyProject") and select **Embedded C Project** as the **Project type**, and **Atollic ARM Tools** as the **Toolchain**. Then click the **Next** button to display the **TrueSTUDIO® Build Settings** page.

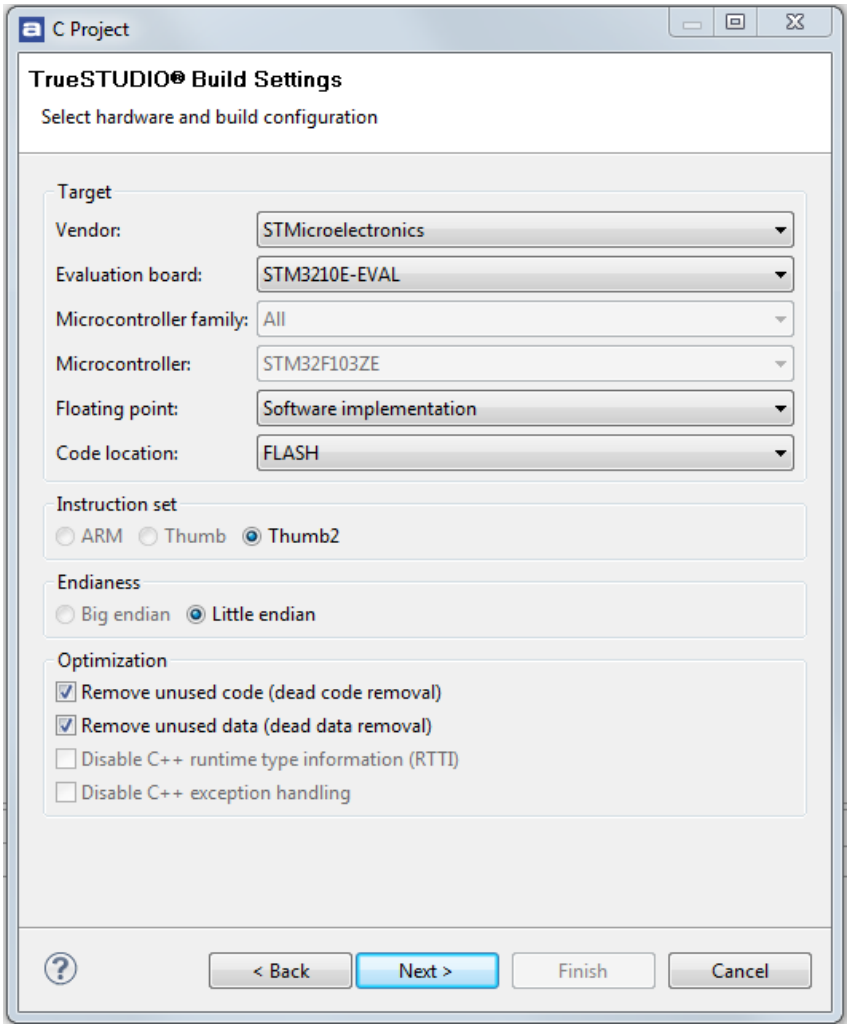


Figure 11 - TrueSTUDIO® Build Settings

3. In the **TrueSTUDIO® Build Settings** page, configure the hardware settings according to your Evaluation board or custom board design.

   Some microcontrollers have as an alternative hardware implementation of floating points.

   Please note that your evaluation board may have hardware switches for configuration of RAM or FLASH mode. This setting *must* be the same in both the project wizard and on the board.

   Finally, click the **Next** button to display the **TrueSTUDIO® Debug Settings** page.

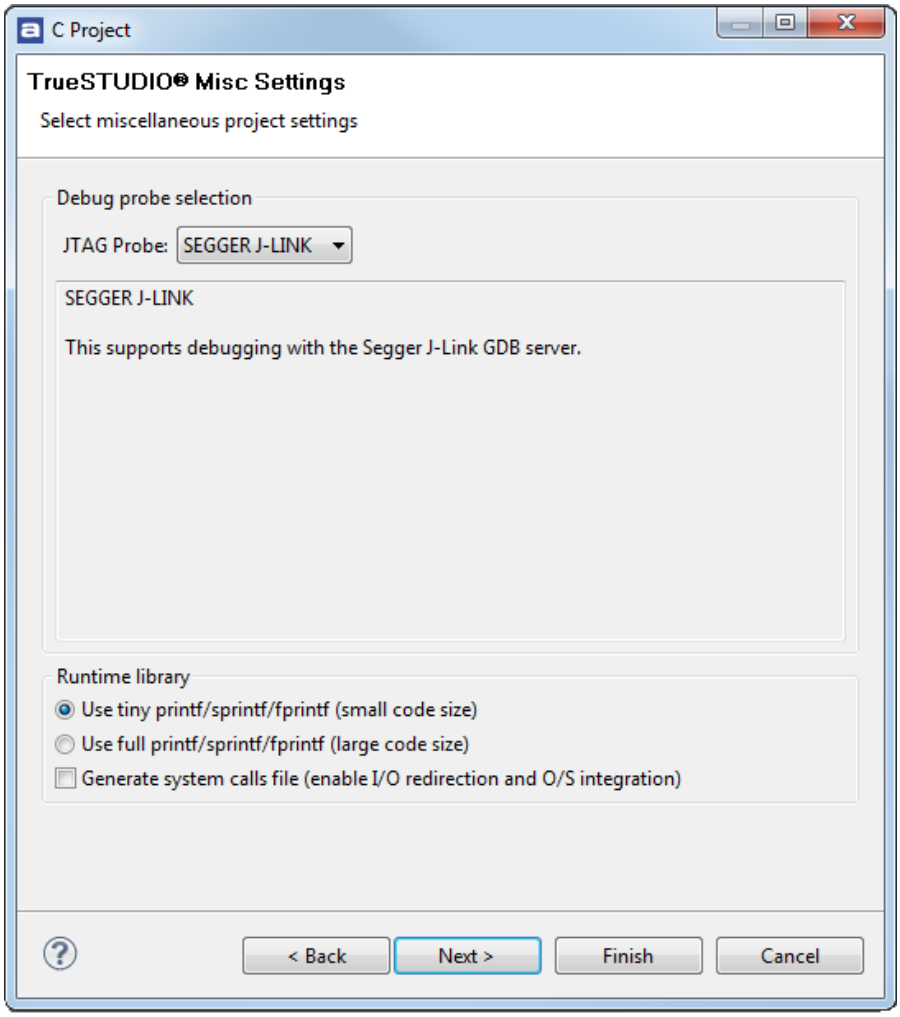


Figure 12 - TrueSTUDIO® Miscellaneous project settings

4. Select the JTAG probe you are using when debugging.

   You can also select the desired runtime library configuration here. If you have a limited amount of memory, the tiny printf-setting is recommended.

   Click the **Next** button to display the **Select Configurations** page.

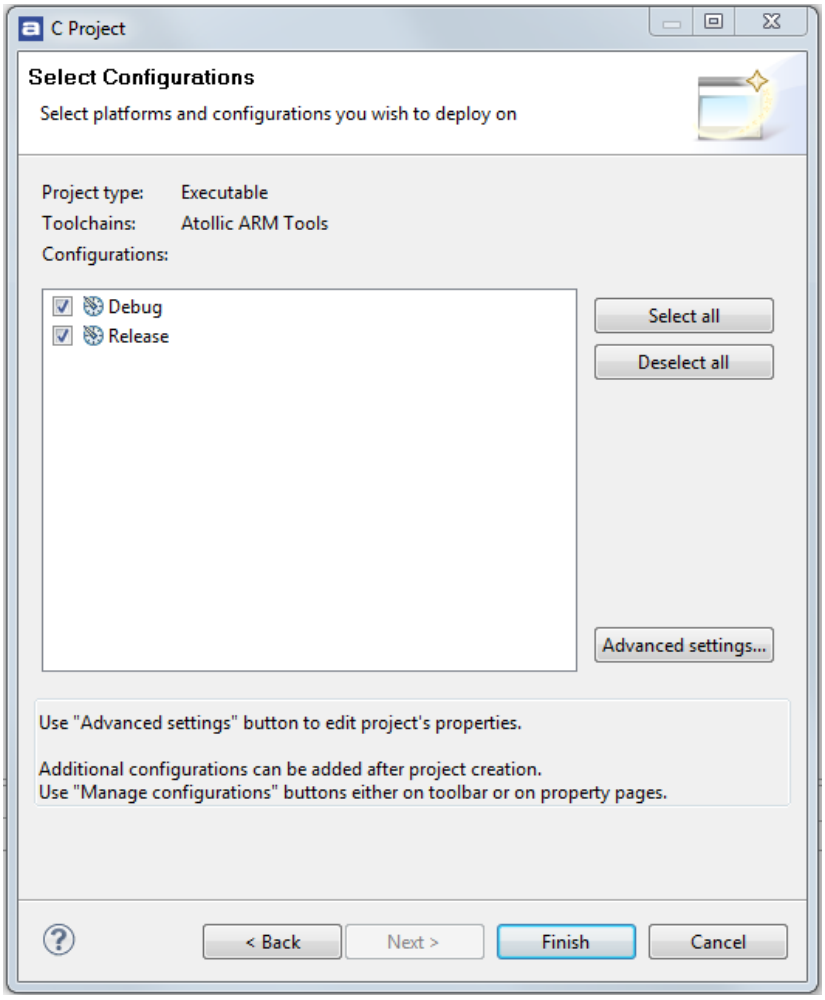


Figure 13 - Select Configurations

5. In the **Select Configurations** page, click on the **Finish** button to generate a new C project. The Advanced settings can be used to set additional properties such as Task Repository for the project.

6. A new managed mode C project is now created. ***Atollic TrueSTUDIO®*** generates target specific sample files in the project folder to simplify development.

7. Expand the project folder (such as “MyProject” in the example above) and the **src**-folder in the **Project Explorer** docking view.

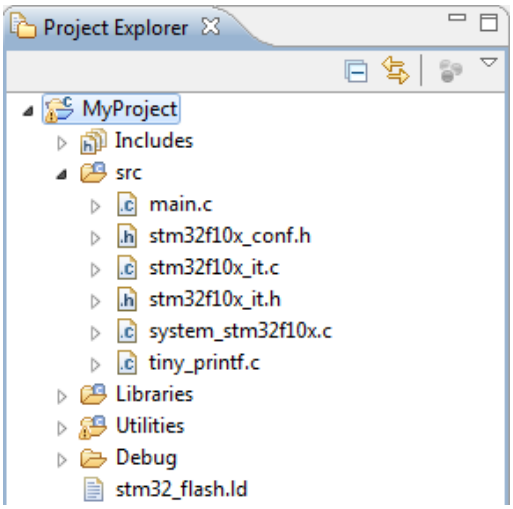


Figure 14 - Project Explorer view

8. Double click on the **main.c** filename in the **Project Explorer** tree to open the file in the editor.

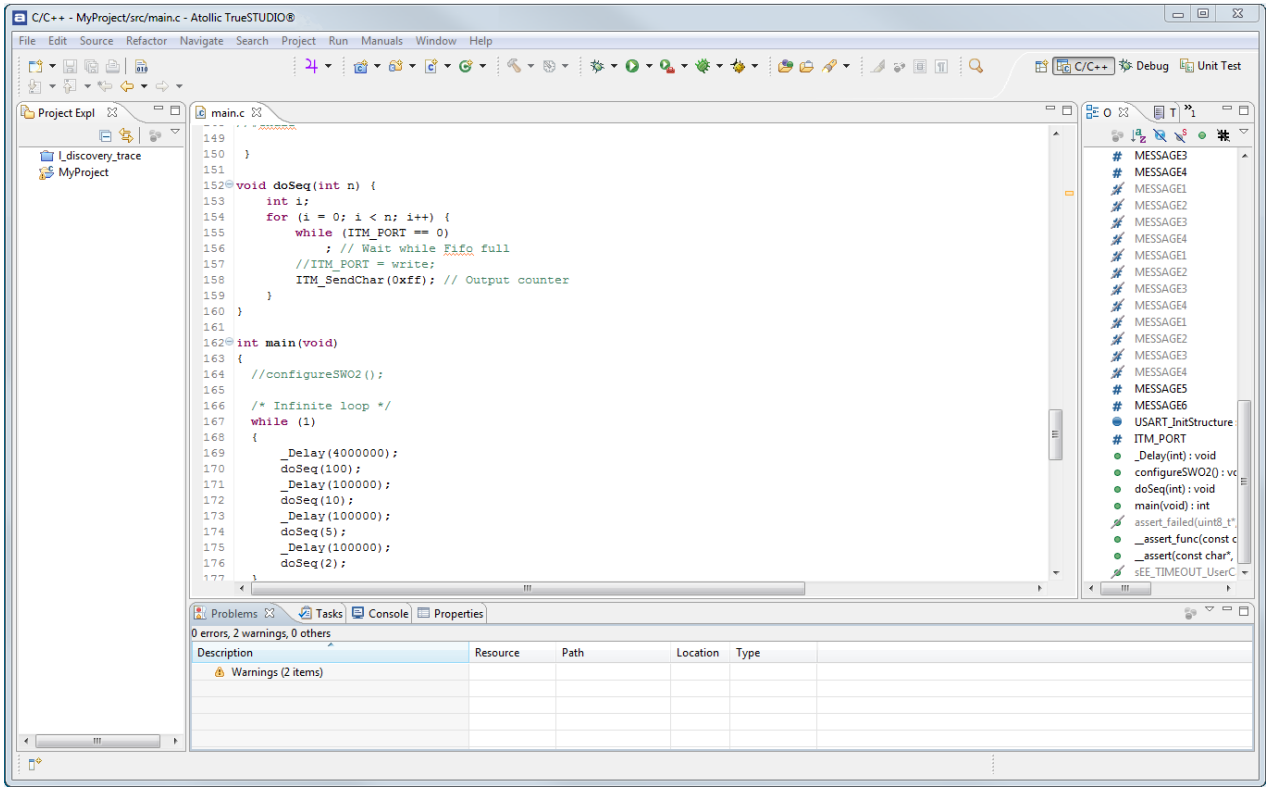


Figure 15 - Editing

IMPORTING AN EXAMPLE PROJECT

You can download example-project from STMicroelectronics® packed in a standard format for a no-hassle works "out of the box experience. To import and use them, follow these steps:

Download and perhaps unzip the package containing the ***TrueSTUDIO®*** example projects. It is recommended that the files be extracted to a folder "near" the root. The reason for this is that STMicroelectronics® use very long path lengths. If the files are installed to a folder that is too deep in your directory structure, you may run into violations of Windows path length character limitations.

Start ***Atollic TrueSTUDIO®*** and select a workspace. If the example-project includes a workspace, you should use that workspace. Otherwise you can use an existing workspace or create a new.

Import one or more examples using the import wizard:

- Select **File** -> **Import** -> **Existing Projects into Workspace**
- Select "**Select root directory**" and browse to the directory containing the ***TrueSTUDIO®*** example projects.
- Select which example projects to import and click "**finish**"

Make sure that the option "**Copy projects into workspace**" in unchecked!

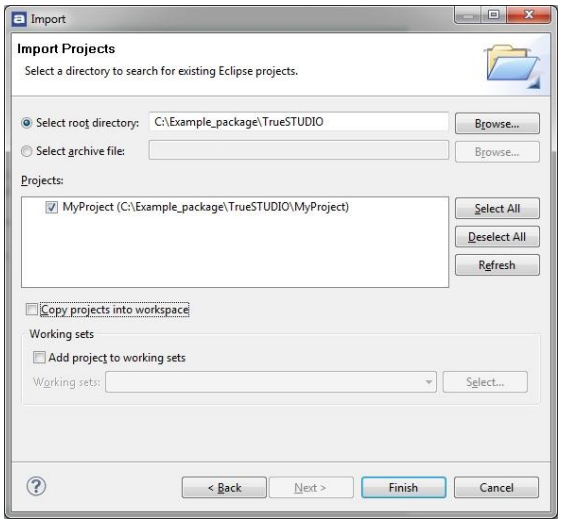


Figure 16 – Importing example project

Please note that the projects files and directories can physically reside outside the currently active workspace. In this case the `C:\Example_package\TrueSTUDIO directory`.

CONFIGURING THE PROJECT

Managed mode projects can be configured using dialog boxes (unmanaged mode projects require a manually maintained makefile).

Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Lite provides a simplified configuration GUI, with limited GUI options to control the command line tool options. Developers must set command line flags (such as -Os, -Wall, etc) manually, but it can be done from inside the GUI without any need to resort to makefiles.

Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Professional on the other hand provides extensive GUI controls for configuration of command line tool options using a simple point-and-click mechanism.

To configure a managed mode project, perform the following steps:

1. Select your project in **Project Explorer** docking view to the left.
2. Select the **Project, Properties** menu command.

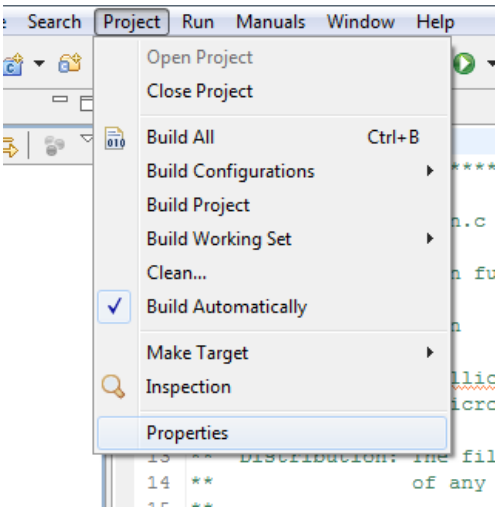


Figure 17 - Project properties menu command

3. The project **Properties** dialog box is displayed.

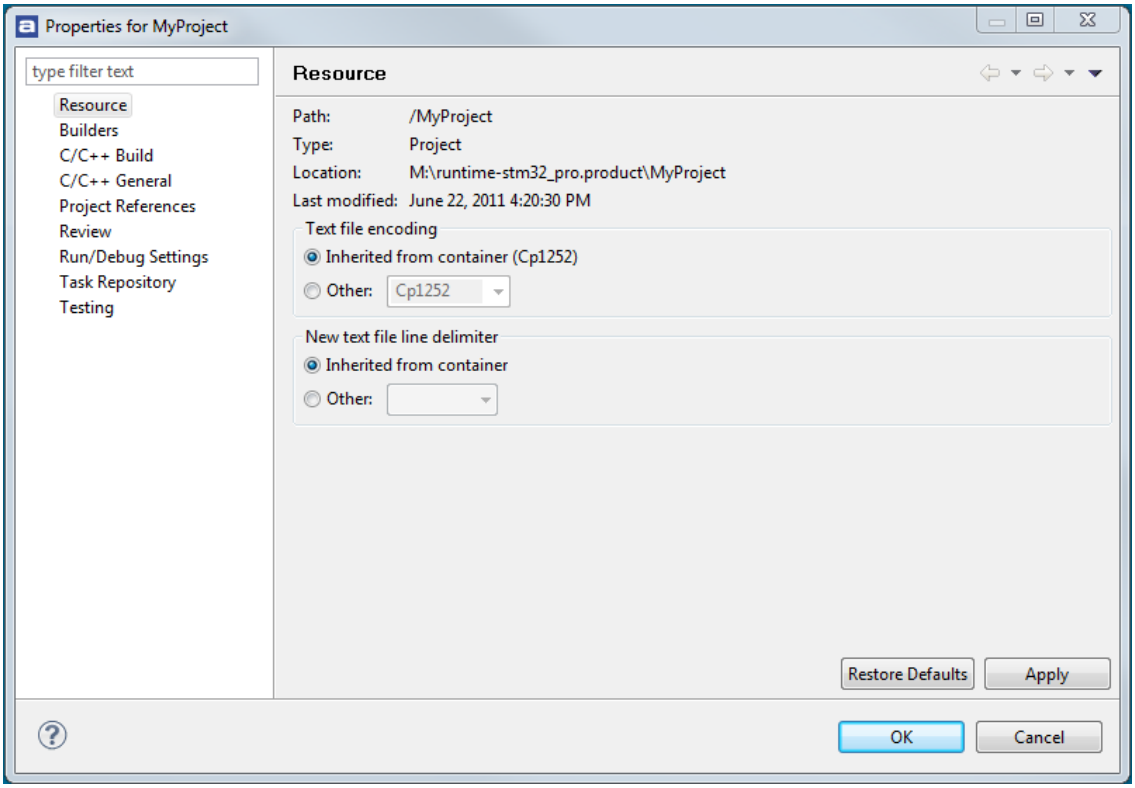


Figure 18 - Project properties dialog box

4. Expand the **C/C++ Build** item in the tree in the left column. Then select the **Settings** item to display the build **Settings** panel.

   Please note that in the ***Lite*** version, most of these settings are grayed-out (they are all available in the ***Professional*** version).

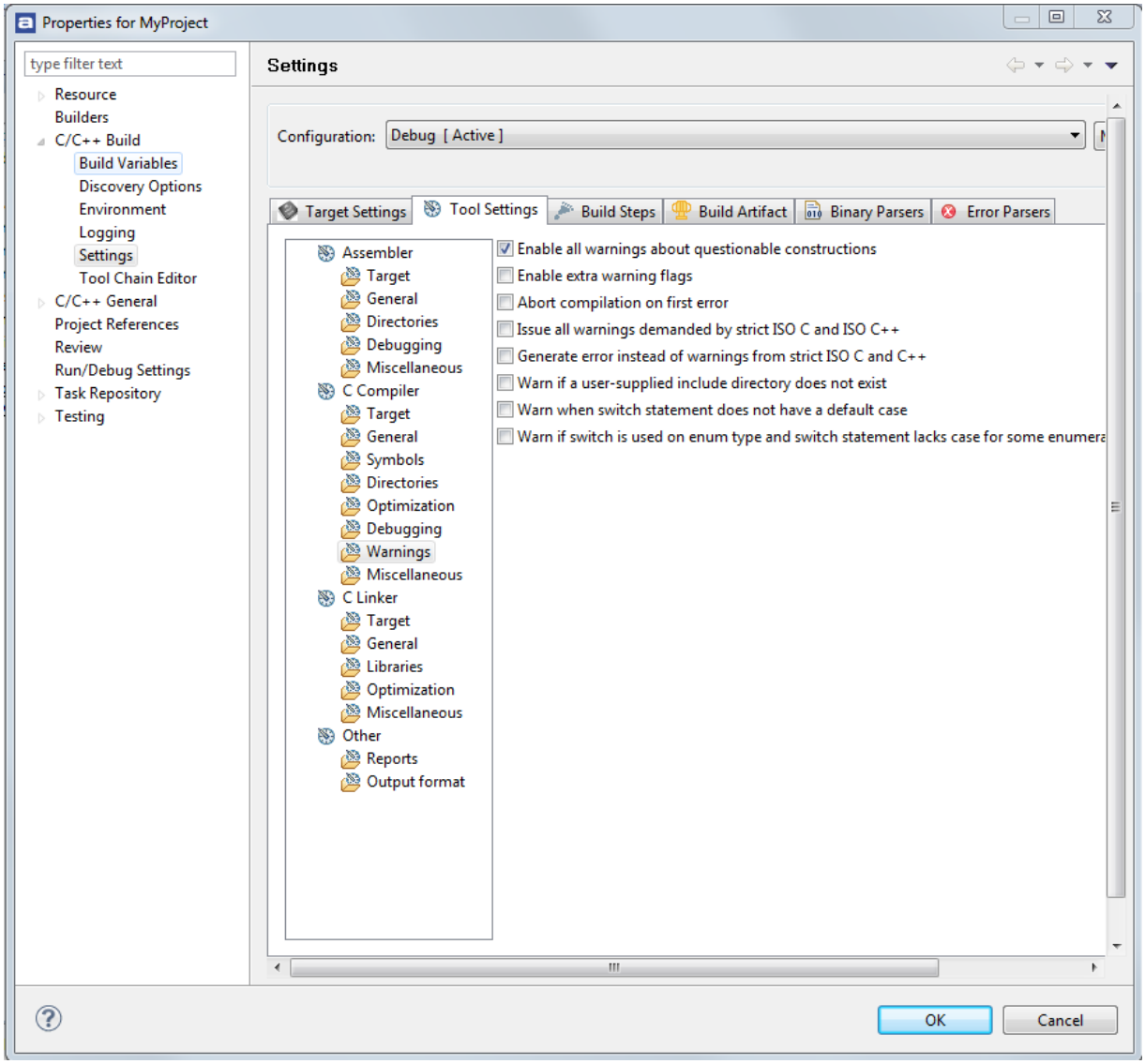


Figure 19 - Project properties dialog box

5. For ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Professional*** select panels as desired and configure the command line tool options using the GUI controls.

   Advanced users may want to enter command line options manually, and this can be done in the **Miscellaneous** panel for any tool.

   In the ***Lite*** version you must enter the command line options manually using the **Miscellaneous** panels.

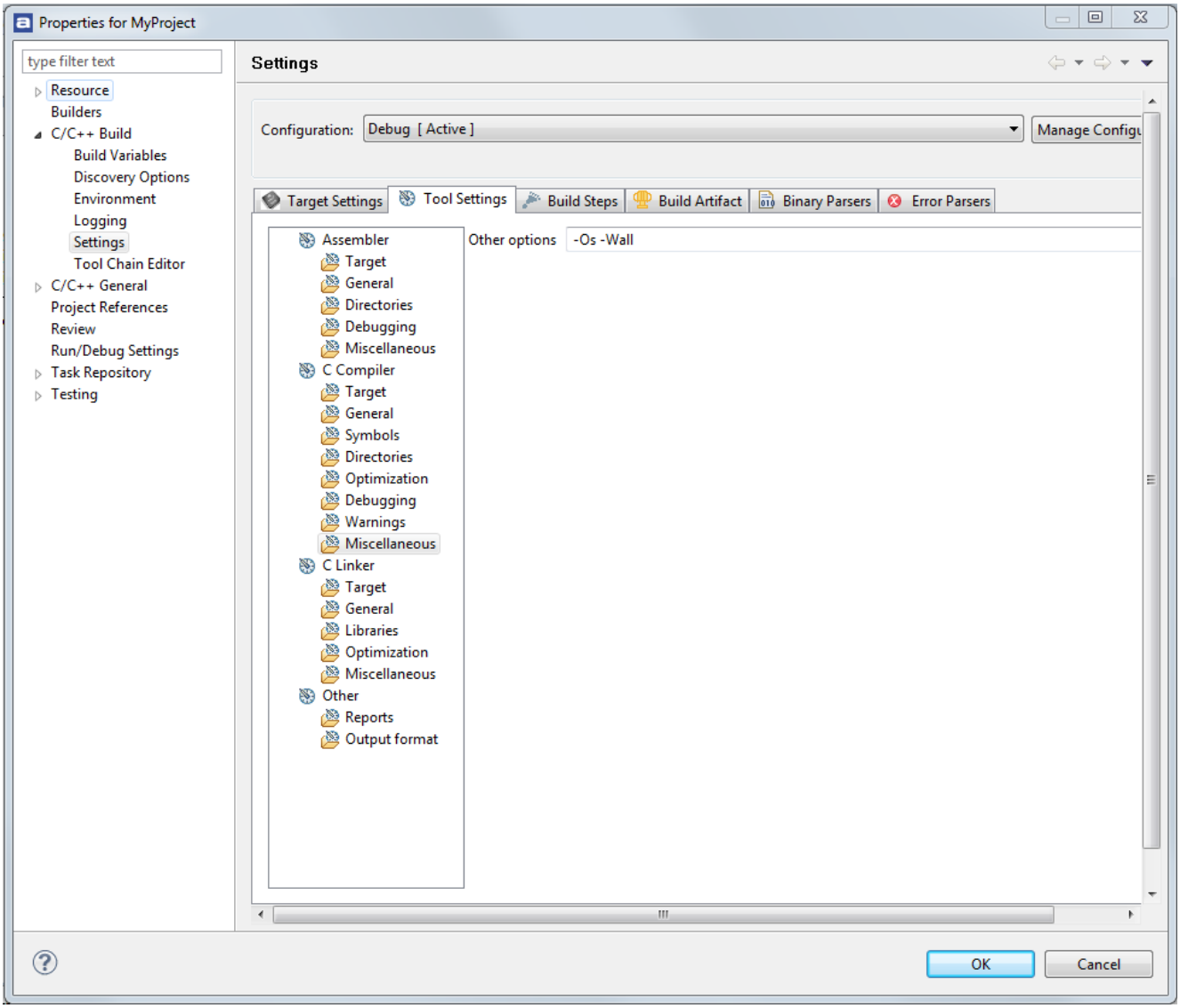


Figure 20 - Project properties dialog box

6. Some project settings are relevant for both managed mode projects and unmanaged mode projects. For instance the selected microcontroller or evaluation board may affect both the options to the compiler during a managed mode build and also how additional **TrueSTUDIO®** components, for instance the SFR-Viewer and debugger, will behave.

   Project settings relevant for both managed mode projects and unmanaged mode projects are collected under the **Target Settings** tag.

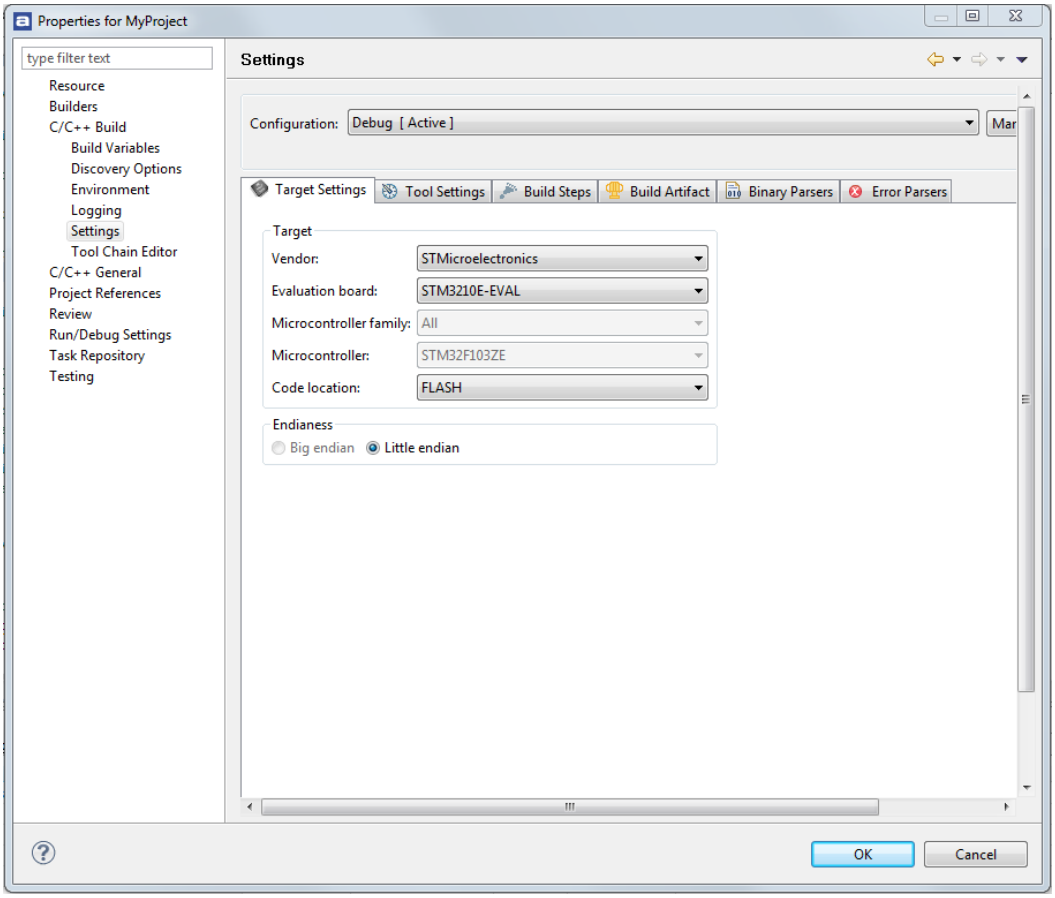


Figure 21 - Project properties dialog box

7. When the configuration is completed, click the **OK** button to accept the new settings.

BUILDING THE PROJECT

By default, ***Atollic TrueSTUDIO®*** builds the project automatically whenever any file in the build dependency is updated. This feature can be toggled using the **Project, Build Automatically** menu command. As automatic building is switched on by default, new projects created by the project wizard are built automatically when the projects are created.

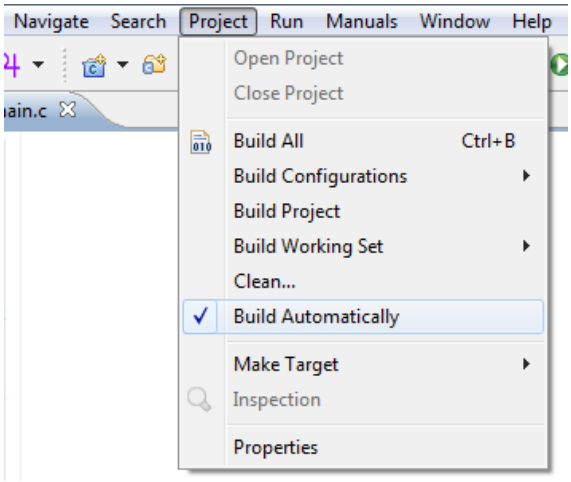


Figure 22 - Build automatically menu command

BUILD

To manually trigger a build, click on the **Build** toolbar button. Only the files that need to be recompiled will be rebuilt.

Figure 23 - Build toolbar button

REBUILD ALL

To force a "rebuild all", perform the following steps:

1. Open the **Console** view by clicking on its tab title. This will ensure you can see the build process.

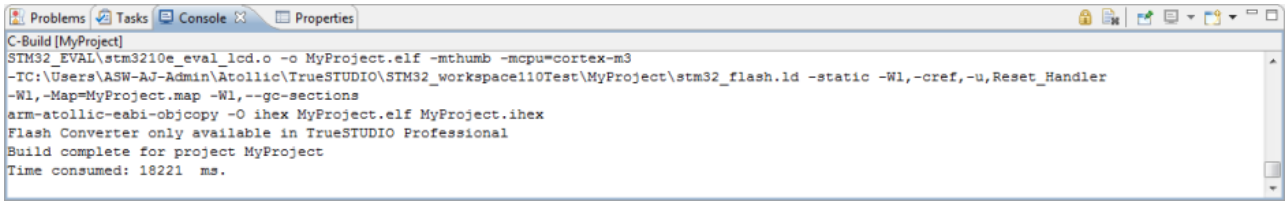


Figure 24 - Build console

2. Select the **Project, Clean…** menu command. This will delete the object files and application binary file from the last rebuild and thus trigger a complete rebuild of the project (if automatic build mode is still switched on).

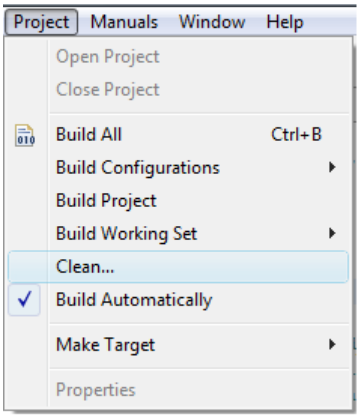


Figure 25 - Clean project

3. A dialog box with some options is displayed. Click on the **OK** button without any changes.

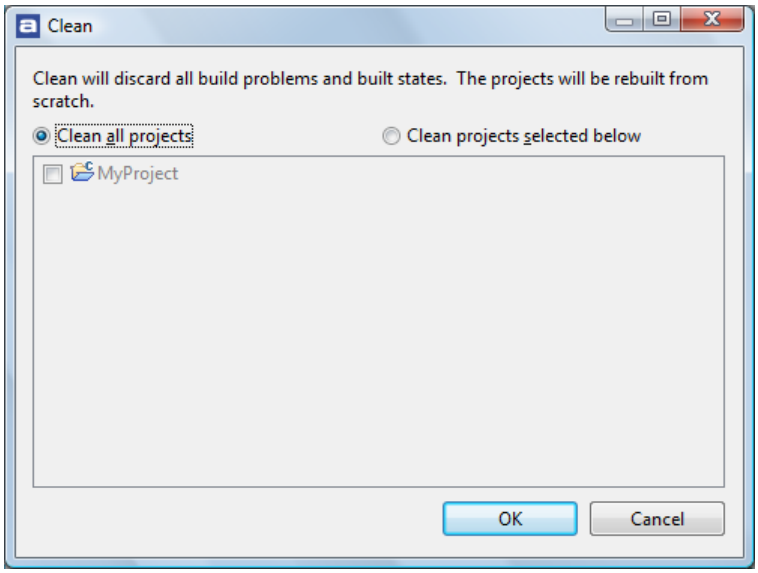


Figure 26 - Clean project dialog box

4. If **Build automatically** is enabled, a rebuild is started and the assembler, compiler and linker output is displayed in the **Console** view. If it is not, trigger a build from the **Project** menu or the **Build** toolbar button. A build is then started and the build output is displayed in the **Console** view.

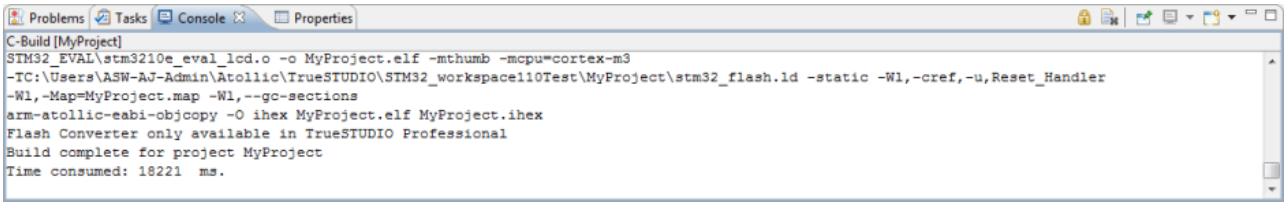


Figure 27 - Build console

DEBUGGING

Atollic TrueSTUDIO® for STMicroelectronics® STM32™ *includes* a very powerful graphical debugger based on the GDB command line debugger. ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™*** also bundle GDB servers for some of the supported JTAG probes, including the ST-LINK JTAG probe.

Atollic TrueSTUDIO® auto-start and auto-stop the gdbserver as needed, thus creating a seamless integration of debug servers.

To prepare for debugging using an ST-LINK JTAG probe connected to your electronic board, perform the following steps:

1. Verify that the RAM/FLASH switch on the board (if available) is set in the right mode. It must match the ***Atollic TrueSTUDIO®*** project configuration.
2. Find out if your board supports JTAG-mode or SWD-mode debug connectors. This information might be needed later.
3. Connect the JTAG cable between the JTAG dongle and the electronic board.
4. Connect the USB cable between the PC and the JTAG dongle.
5. Make sure the electronic board has proper power supply.

Once the steps above are performed, a debug session in ***Atollic TrueSTUDIO®*** can be started.

STARTING THE DEBUGGER

Perform the following steps to start the debugger:

1. Click on the **Debug** toolbar button (the insect icon) or the **F11** key to start the debug session.

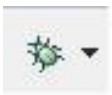


Figure 28 - Start the debug session

Alternatively, start the debug session with a right-mouse-click on the project name in the **Project Explorer** docking view and select **Debug As**, **Embedded C/C++ Debugging.**

2. The first time debugging is started for a project, ***Atollic TrueSTUDIO®*** display a dialog box that enable developers to confirm the debug configuration before launching the debug session. After the first debug session is started, this dialog box will not be displayed any more.

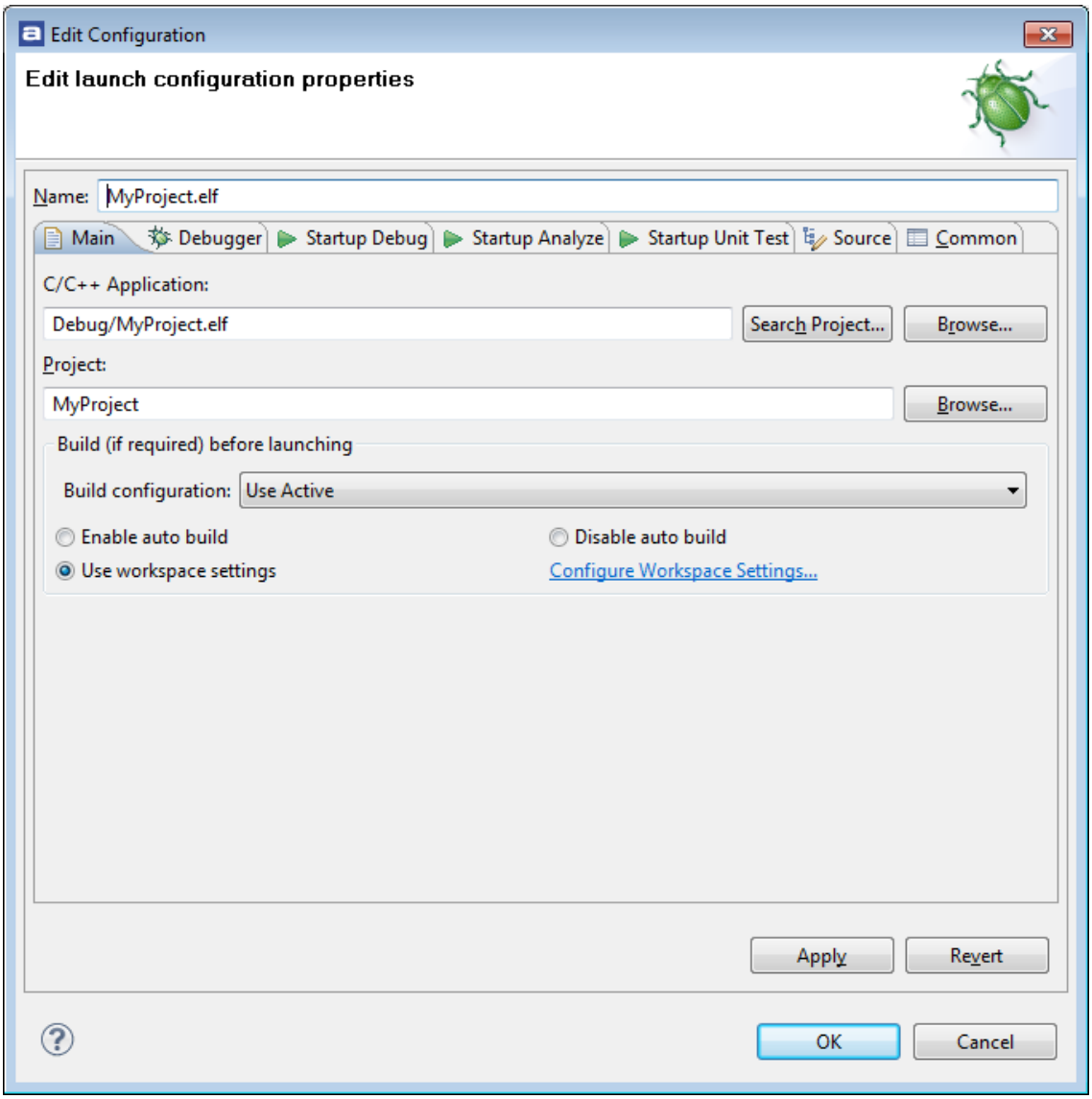


Figure 29 - Debug configuration dialog box

The debug-configurations can also be reach by right-clicking on the project and select **Debug As**, **Debug Configurations…**

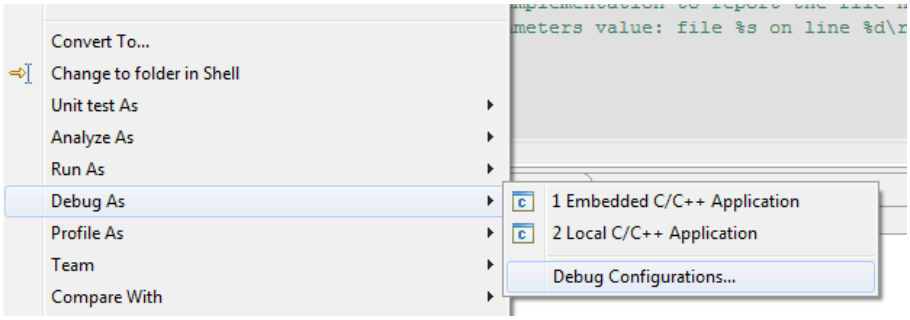


Figure 30 – Open debug configurations

3. The **Main** tab contains information on what project and executable to debug. The settings in the **Main** tab do normally not have to be changed. In this tutorial, do not make any changes in the **Main** tab, and click on the **Debugger** tab to display it.

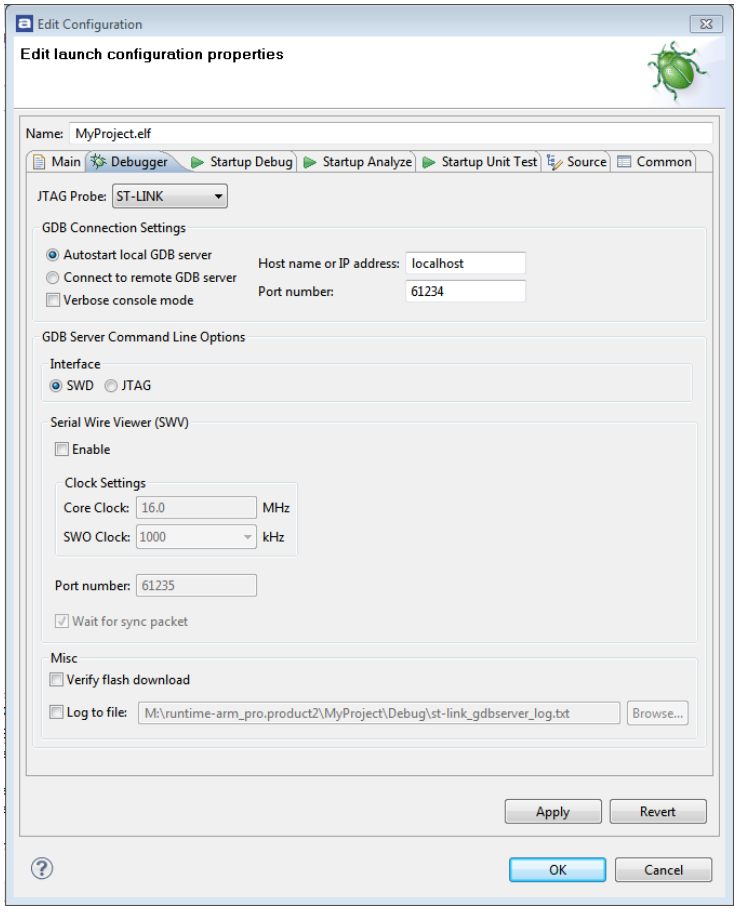


Figure 31 - Debug configuration dialog box

4. The **Debugger** tab contains information on what JTAG probe to use, its configuration, and how to start it.

   Select the JTAG probe you want to use.

   If you want to use Serial Wire Viewer (SWV), you have to select the **SWD** interface.

   For the ST-Link-probe the SWV-settings has the option **Wait for sync packet**. Enabling this will ensure more accurate data being receive, but can also lead to more packages being lost. Especially if a lot of data is being sent.

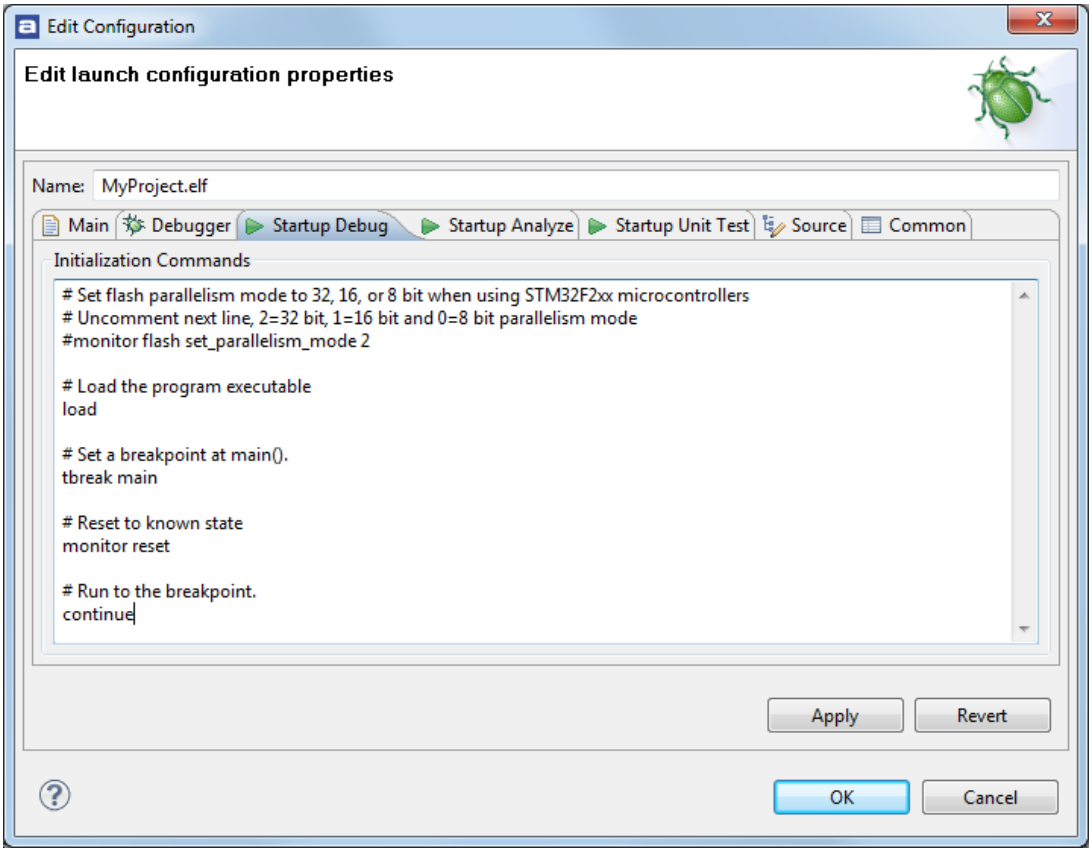


Figure 32 - Debug configuration dialog box

5. The **Startup Debug** tab contains an initialization script that is sent to the GDB debugger upon debugger start. This script can contain any GDB or gdbserver commands that work with your application, JTAG probe and board. The **Startup Debug** tab is also where gdb-script programs are defined. In this tutorial, do not make any changes in the **Startup Debug** tab. Click on the **OK** button to start the debug session.

   The **Startup Analyze** and **Startup Unit Test** tabs contain gdb-scripts used by the ***Atollic TrueANALYZER®*** and ***Atollic TrueVERIFIER®*** products.

6. ***Atollic TrueSTUDIO®*** launches the debugger, and switches to the **Debug** perspective, which provides a number of docking views and windows suitable for debugging.

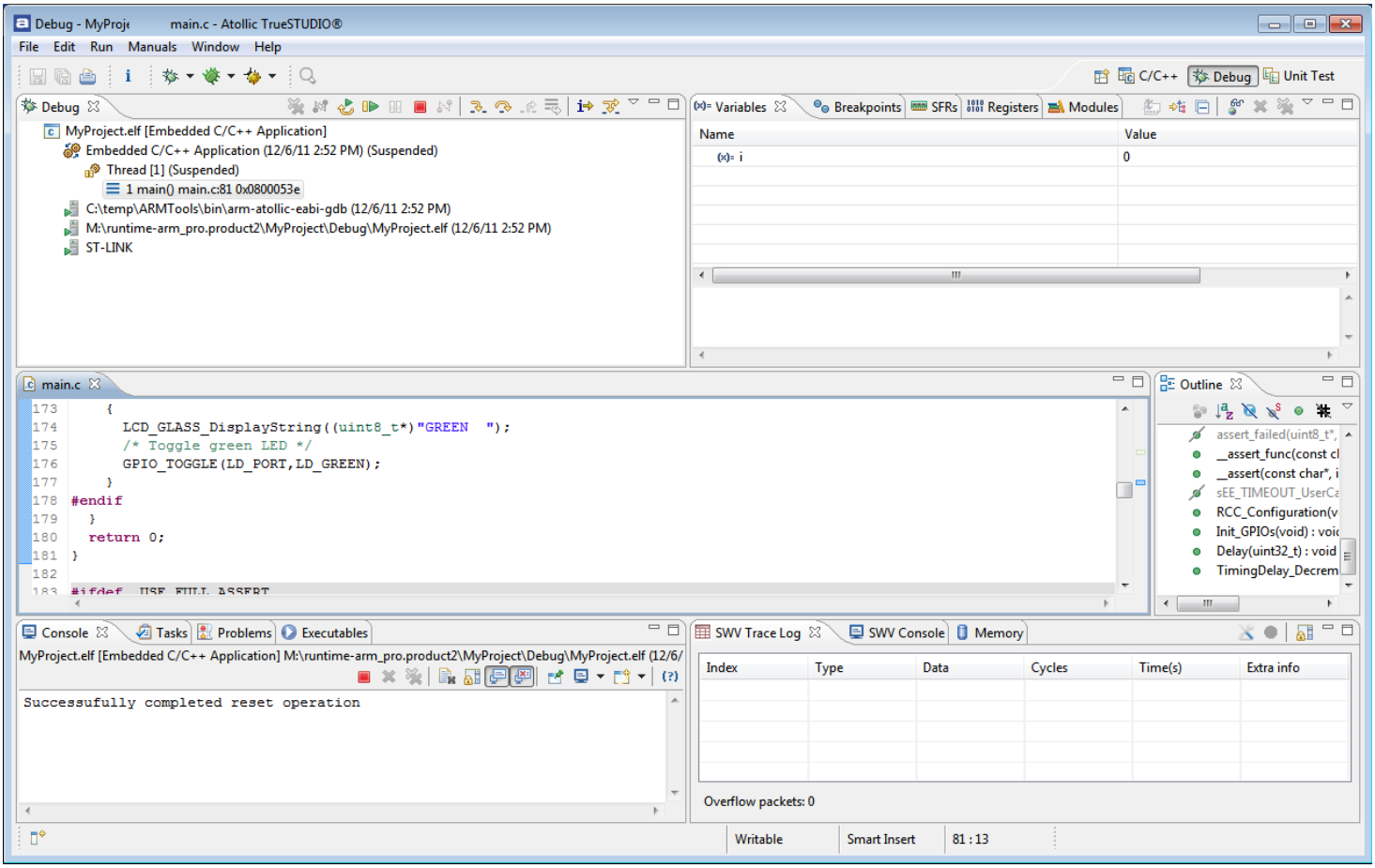


Figure 33 – Debug perspective

7. **Important Note:** Almost all developers start with the Debug configuration for building, downloading and investigation the behavior of software. There are two important properties of the debug build:

   - Complete symbolic information is emitted by the toolchain to help navigate through the information in the high level code during the debug process.
   - The lowest level of optimization is normally used.

   When code is found to behave as specified, a Release build with no symbolic debug information and high optimization level is usually built. After switching from Debug to Release, the target can be programmed by launching a

debugging session. It is important to be cautious when doing so, as unexpected results may occur. The ***Atollic TrueSTUDIO®*** philosophy of resolving which executable image to program into the part must be considered carefully in this case.

List making and history keeping are important driving factors in determining the practical behavior of ***Atollic TrueSTUDIO®*** development environment.

It is possible to create multiple debug session configurations. This is done by clicking on the down arrow next to the debug icon and selecting debug configurations to bring up the list of existing configurations. By right clicking on an existing configuration, you enable the options to create a new one, duplicate the existing one or delete it. The easiest way to create a new configuration is duplicate an existing one, edit the configuration settings in the dialogue box, and then rename it. You may then toggle among the names of the debug session configurations in the list, if needed to launch the most suitable session for the task at hand.

If the user does not explicitly choose a debug session configuration from an existing list, then ***Atollic TrueSTUDIO®*** by default will launch the last debug session configuration that was used. If a developer has been debugging a build created in the Debug build configuration, then switched to the Release build configuration, and launches a debug session by clicking on the debugging icon, ***Atollic TrueSTUDIO®*** will fetch the last debugging session configuration, which specifies that the Debug build image is to be programmed into the part. ***Atollic TrueSTUDIO®*** has no provision to use a debug session configuration that fetches the image built from the Release configuration and programs that image into the part. This behavior is different from some toolchains that automatically reconfigure their debug launch mechanisms to use the built image from the current build configuration. In ***TrueSTUDIO®***, the user must explicitly create a debug session configuration that uses the build image that is appropriate after switching among build configurations.

For example, the location of the build image built in the Debug configuration for a Project Wizard generated project in the debug session configuration dialogue box is shown here:

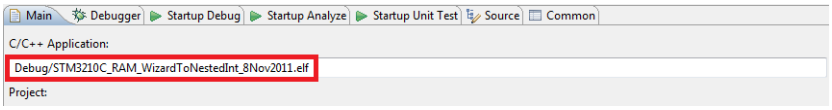


Figure 34 – File to debug

The Project Wizard puts Debug and Release folders in the project folder within the workspace. To make this fetch the image built using the Release configuration, the user may use the Browse button to locate the image, or simply change **Debug** in the above path to **Release** and then rename the debug session configuration and switch to it for use.

If desired, other properties of the new debug session configuration can be edited to achieve a desired behavior. For example, setting the temporary breakpoint at the first line of main() could be commented out in the dialogue box under the **Startup Debug-**tab. This would result in being able to simple program the part and start execution (load and go) functionality, which is probably the most desirable use case when using a Release build.

8. The **Debug** perspective and other perspectives in ***Atollic TrueSTUDIO®*** can be enhanced with lots of practical toolbar buttons and menus by selecting **Windows**, **Customize Perspective…**

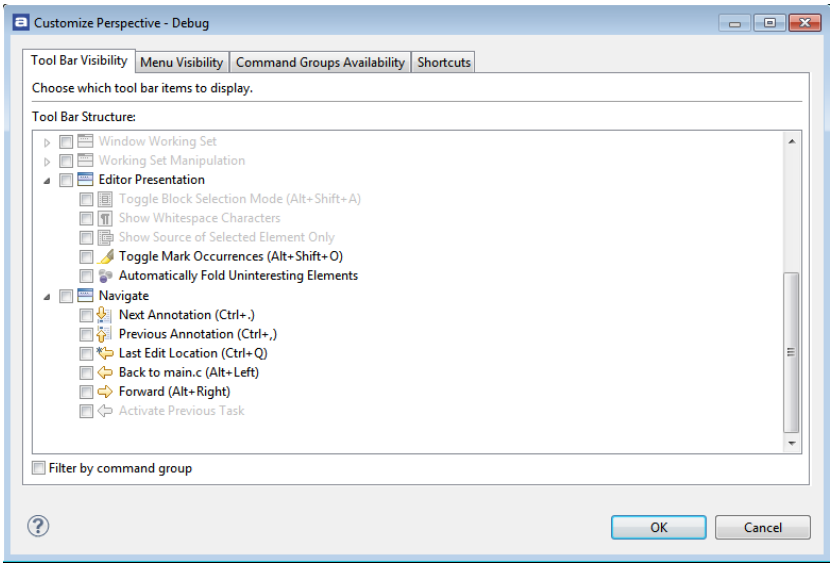


Figure 35 – Customize Perspective

DEBUGGING

Once the debug session has been started, ***Atollic TrueSTUDIO®*** switch to the **Debug** perspective, sets a breakpoint at main(), resets the processor and executes the startup code until execution stops at the first executable code inside main().

The **Debug** perspective is now active, with the next program line to execute being highlighted in the source code window.

A number of execution control functions are now available from the **Run** menu.

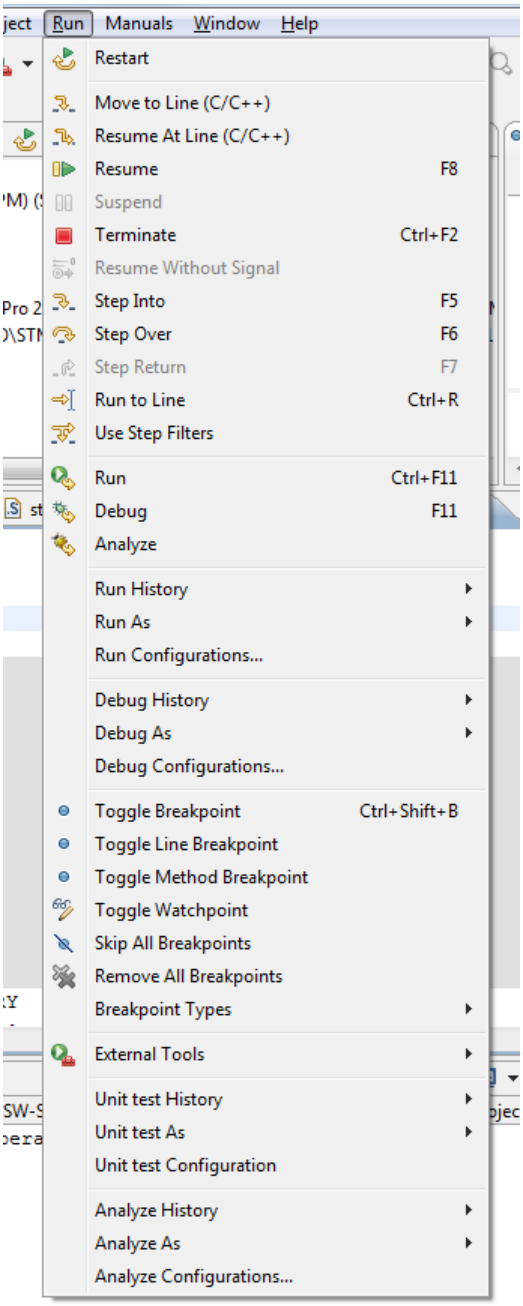


Figure 36 - Run menu

Alternatively, use the corresponding execution control commands in the **Debug** view toolbar.

Figure 37 - Run control toolbar

A commonly used task that is not available from the **Run** menu is to switch between C/C++ level stepping in the C/C++ source code window, and assembler level instruction stepping in the **Disassembly** view.

Click on the instruction stepping button to activate assembler mode instruction stepping in the **Disassembly** view. Click it once more to return to C/C++ level stepping in the C/C++ source code editor.

A standard code breakpoint at a program line can easily be inserted by a right-mouse-click in the left margin of the C/C++ source code editor. A context menu will be opened.

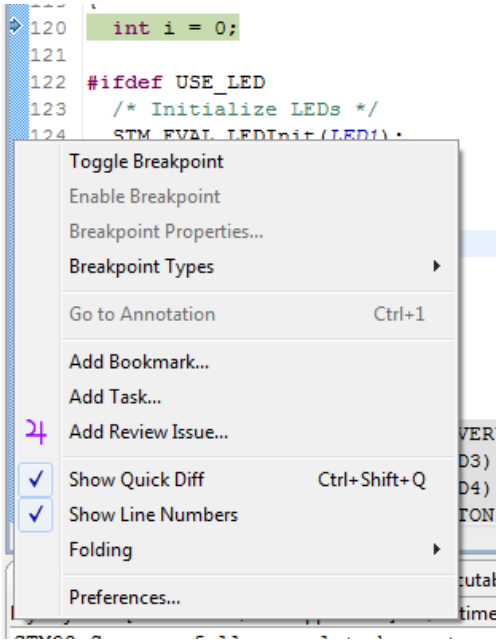


Figure 38 - Toggle breakpoint

Select the **Toggle Breakpoint** menu command to set or remove a breakpoint from the corresponding program line.

SFR's can be reached and manipulated from the SFR's view. To open it, select **Window, Show View, Other..**. menu command.

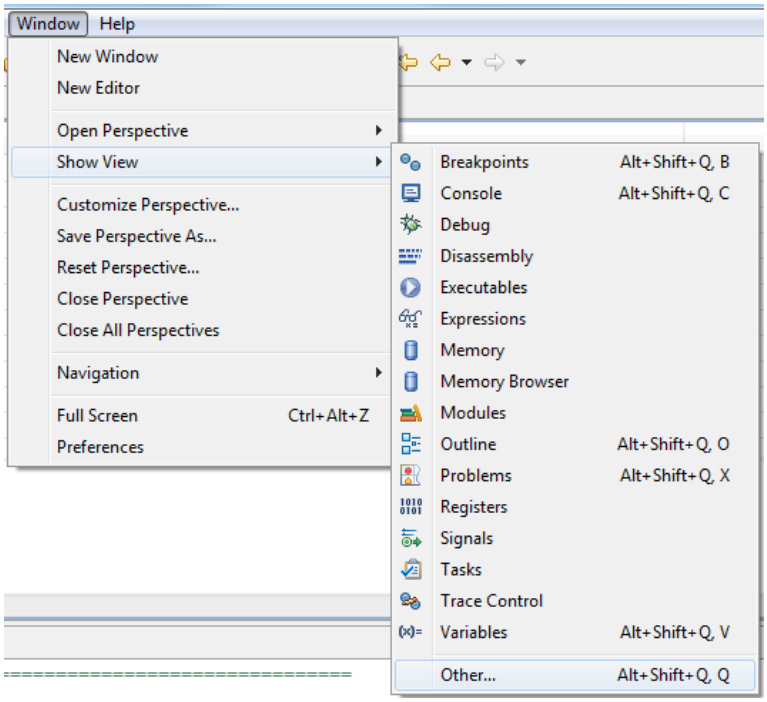


Figure 39 - Select other view

Then open up Debug and select SFR's.

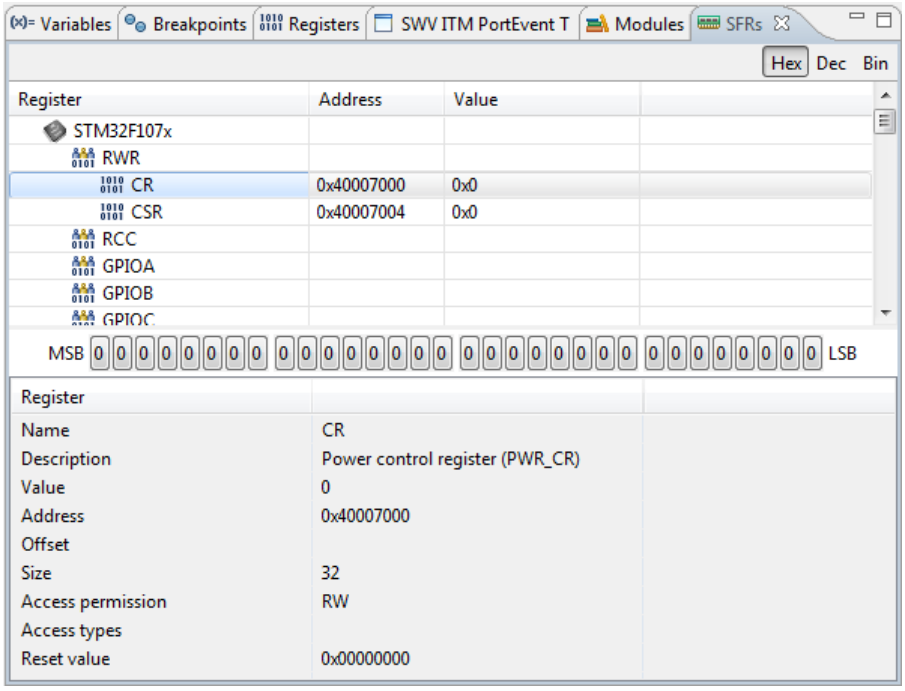


Figure 40 - SFR view

USING THE SERIAL WIRE VIEWER

SERIAL WIRE VIEWER OVERVIEW

To use system analysis and real-time tracing in ARM® processors, a number of different technologies interact; Serial Wire Viewer (SWV), Serial Wire Debug (SWD) and Serial Wire Output (SWO). These technologies are part of the ARM® Coresight™ debugger technology and will be explained below.

SERIAL WIRE DEBUG (SWD)

Serial Wire Debug (SWD) is a debug port similar to JTAG, and provides the same debug capabilities (run, stop on breakpoints, single-step) but with fewer pins. It replaces the JTAG connector with a 2-pin interface (one clock pin and one bi-directional data pin). The SWD port itself does not provide for real-time tracing.

SERIAL WIRE OUTPUT (SWO)

The Serial Wire Output (SWO) pin can be used in combination with SWD and is used by the processor to emit real-time trace data, thus extending the two SWD pins with a third pin. The combination of the two SWD pins and the SWO pin enables Serial Wire Viewer (SWV) real-time tracing in compatible ARM® processors. Please note that the SWO is just one pin and it is easy to set a configuration that produces more data than the SWO are able to send.

SERIAL WIRE VIEWER (SWV)

Serial Wire Viewer is a real-time trace technology that uses the Serial Wire Debugger (SWD) port and the Serial Wire Output (SWO) pin. Serial Wire Viewer provides advanced system analysis and real-time tracing without the need to halt the processor to extract the debug information.

Serial Wire Viewer (SWV) provides the following types of target information:

- Event notification on data reading and writing
- Event notification on exception entry and exit
- Event counters
- Timestamp and CPU cycle information

Based on this trace data, modern debuggers can provide developers with advanced debugger capabilities.

INSTRUMENTATION TRACE MACROCELL (ITM)

The Instrumentation Trace Macrocell (ITM) enables applications to write arbitrary data to the SWO pin, which can then be interpreted and visualized in the debugger in various ways. For example, ITM can be used to redirect `printf()` output to a console view in the debugger. The standard is to use port 0 for this.

The ITM port has 32 channels, and by writing different types of data to different ITM channels, the debugger can interpret or visualize the data on various channels differently.

Writing a byte to the ITM port only takes one write cycle, thus taking almost no execution time from the application logic.

STARTING SWV-TRACING

To be able to use the Serial Wire Viewer (SWV) you need a JTAG-probe that supports SWV. Older JTAG-probes, such as ST-LINK V1, doesn't.

Also your gdbserver must also support SWV. ST-LINK has to be 1.4.0 or later and SEGGER J-LINK has to be of version 4.32.A or later. If you have an older version installed you must upgrade to the ones included.

Not all boards support SWD-mode. To be able to use SWV, your board has to support SWD and have SWO enabled.

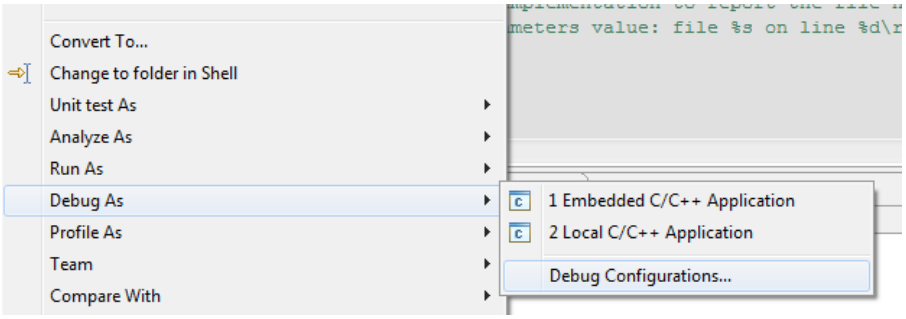


Figure 41 – Open debug configurations

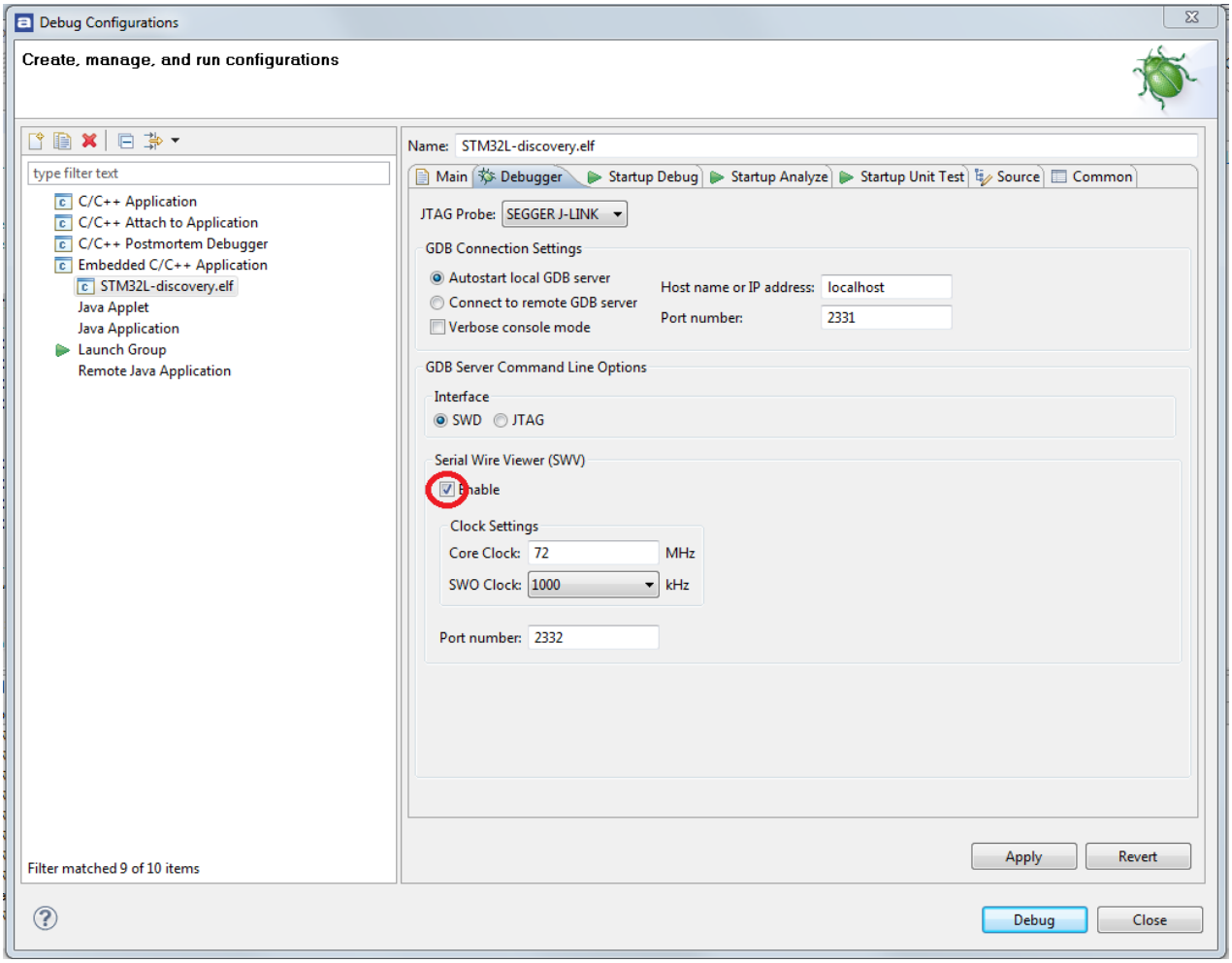


Figure 42 – Change debug configuration for SWV

1. Open ***Atollic TrueSTUDIO®*** debug configuration dialog by right-clicking on your project and select **Debug AS**, **Debug Configuration ….** You need to enable SWV by selecting the SWD-interface and by enabling the **SWV-checkbox**.
   You will have to enter the target's **Core clock** speed you have set in your application and the desired **SWO clock** speed. The latter is depending on your JTAG Probe and should be a multiple of the Core clock.

2. Switch to debug-perspective by starting a debug-session as described above. You need a running debug-session to be able to configure and starting Serial Wire Viewer, thus for SWV to work you cannot just switch to the debug-perspective without actually debugging.

3. Pause the debugging by clicking the yellow **Pause**-button.

4. Open your first SWV-view. We suggest the **SWV Data Trace**.

   Do that by first selecting **Window**, **Show View**, **Other…**

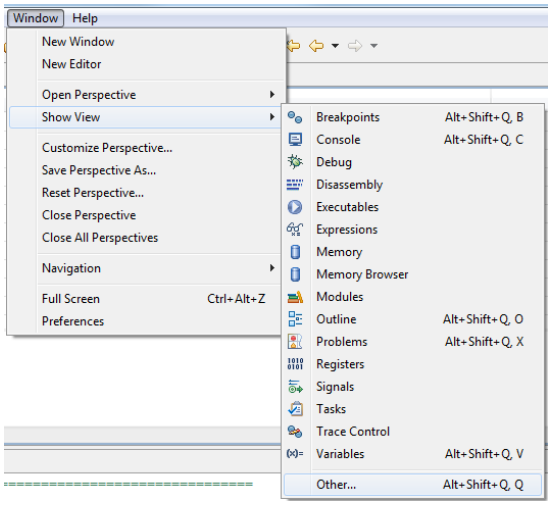


Figure 43 – Open the other views

In the new dialog you open up **Debug** and then you select the **SWV Data Trace**.

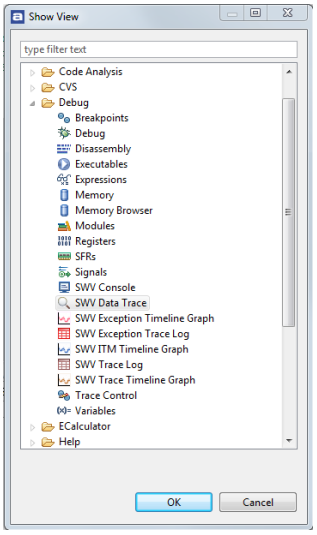


Figure 44 – Select SWV Data Trace

5. Open the SWV-setting panel by clicking on the **SWV-setting**-button in the **SWV Data Trace**-view.

Figure 45 – The SWV-configuration button

6. Configure what and how you want to trace.

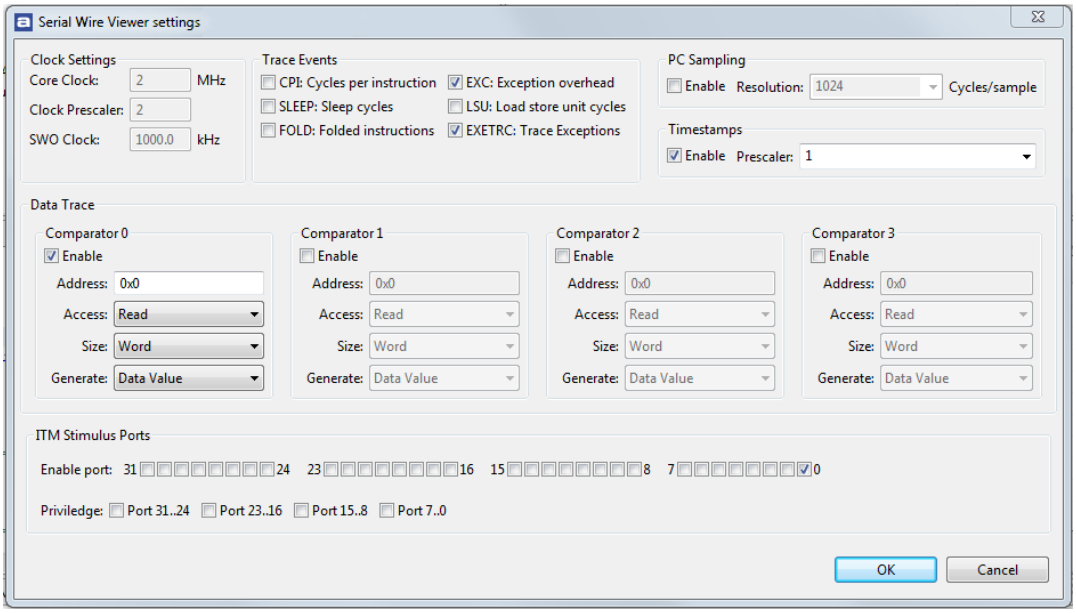


Figure 46 – The SWV-setting dialog

It might be tempting to trace everything as much as possible. However, the SWV works best when tracing a limited set of things. Almost all ARM® products can read and write data faster than the SWO-pin are able to process. That will result in overflow-packages and probably also corrupt data. For best and most reliable result you should not trace for more data then you really need.

When you detect overflow-packages, it is an indication that you have configured SWV to trace more data than SWO can process. You need to decrease the amount of data traced.

If you want to use any of the timeline-views in ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™*** you must enable the Timestamps. The prescaler is set to 1 and should only be changed if you have problem with too many SWV-packages.

You can trace up to four different symbols or areas of the memory, as for an example the value for a global variable. To do that, you have to enable one comparator and enter the name of the variable or the memory-address you want to trace. The value of the traced variables can be displayed both in the **Data trace** view and the **Data Trace Timeline graph**.

7. Save your configuration in ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™*** by clicking the **OK**-button. The configuration is saved with your other debug-configurations and will be used until changed.

8. The views that display SWV-information are:

   -  **SWV Trace Log** - Lists all incoming SWV-packages in a spreadsheet
   -  **SWV Trace Timeline Graph** – A graph displaying how all SWV-packages arrives over time.
   -  **SWV Exception Trace Log** – The same as SWV Trace Log but restricted to Exception-events.
   -  **SWV Exception Timeline Graph** – A graph displaying the exceptions distribution over time. Remember that each exception sends up to three SWV-packages.
   -  **SWV Console** - Prints readable text output from the target application. Typically this is done with printf() that redirects its output to ITM channel 0.
   -  **SWV ITM Timeline Graph** – A graph displaying how ITM-packages are distributed over time.
   -  **SWV Data Trace** – Tracks up to four different symbols or areas in the memory. Can be the name of a global variable.
   -  **SWV Data Trace Timeline Graph** – A graphical display for the values of the variables or memory areas in the **SWV Data Trace**.
   -  **SWV Statistical Profiling –** Statistics of what functions are used the most according to the PC Sampling. This is useful when you are optimizing your code.

   You can have more than one SWV-view open at the same time for simultaneous tracking of events.

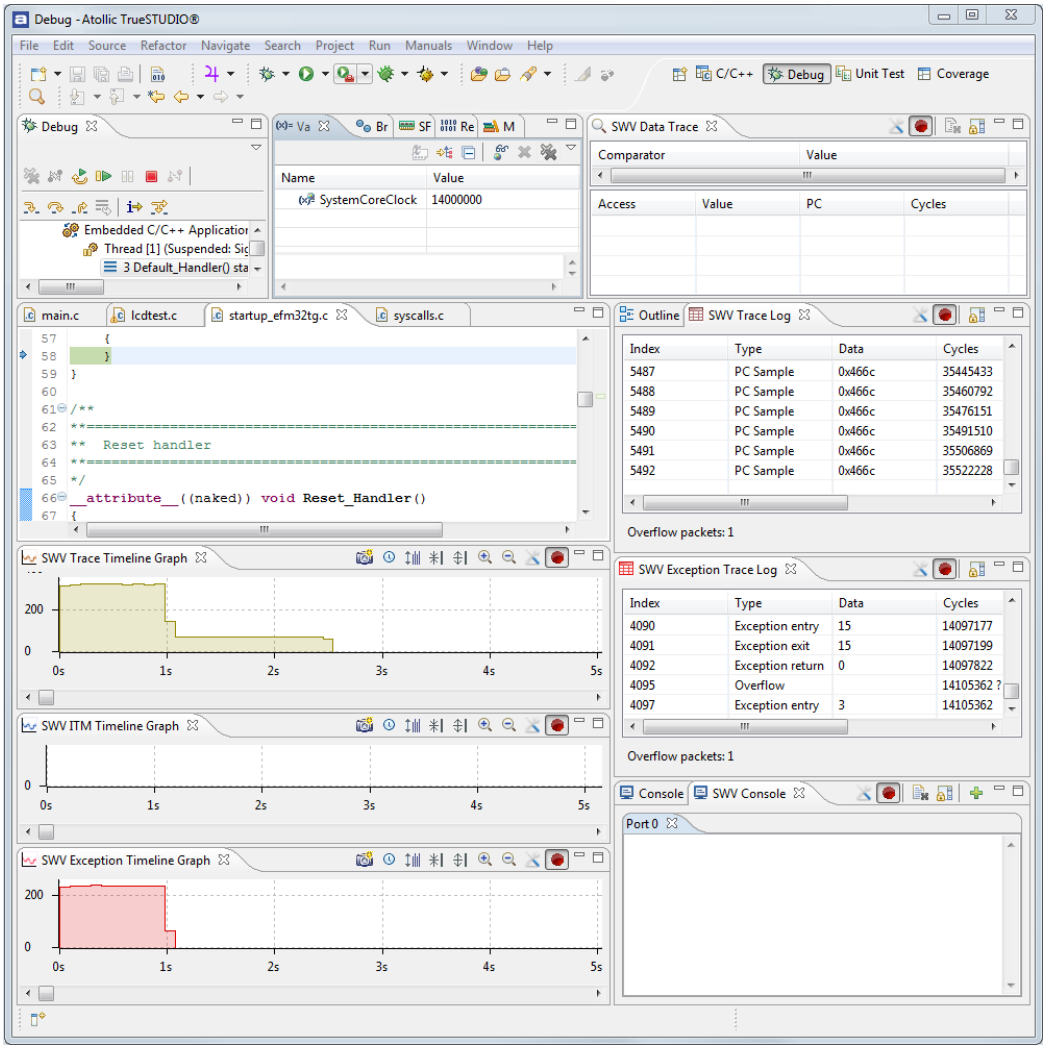


Figure 47 – Many SWV-Views can be displayed at the same time

9. Send the SWV configuration to your board and start SWV-tracing by pressing the **Start/Stop Trace**-button. Your board will not send any SWV-packages before it is configured to do so. You must resend the configuration to the board each time the configuration-registers are resetted.

Figure 48 – The Start/Stop Trace-button sends the configuration to the board

Please note that you can't configure the tracing when it is running. You have to pause debugging and stop tracing first and then you can select the configuration button. Each new configuration has to be sent to the board before it takes effect.

The configuration is sent to the board when the **Start/Stop Trace**-button is pressed.

10. Start the debugging again by pressing the green play-button.

Figure 49 –Start tracing

11. You should now see packages arriving to the **SWV Trace log**.

THE TIMELINE GRAPHS

All the timeline graphs, except the **Data Trace Timeline**, have some common features. The Data Trace is displaying distinct values for variables during the time of execution and has a bit different logic.

- You can save the graph as an image by clicking the camera icon.
- They show the time in seconds by default.
- They assume debugging is paused when starting. If you open the graph during tracing, you need to pause debugging before they will display any data.
- There is a limit for how much you can zoom in when the debugging is running. You can go into more details when debugging is paused.
- You can zoom in by double-clicking on the left mouse-button and zoom out by double-clicking on the right button.
- The tooltip shows the number of packages in each bar. Except for **Trace Timeline graph**, the content of the bars with fewer than 50 packages are showed in details.

STATISTICAL PROFILING

This is a way to check what functions are used the most during the execution of your program. It is not a code analyzer and it will display only statistical information about what code that is executed. This is a technical limitation of the SWV-protocol. Other products from Atollic, such as ***TrueANALYZER®***, are more suited for such needs.

1. Configure SWV to send Program Counter samples. With the configured Core clock-cycle intervals, SWV will report to ***TrueSTUDIO®*** what value the Program Counter has. We recommend that you to begin with configure your

PC-sampling to a high cycle interval. Thus you ensure that you will not overflow the interface. Later on you can test with a lower value.

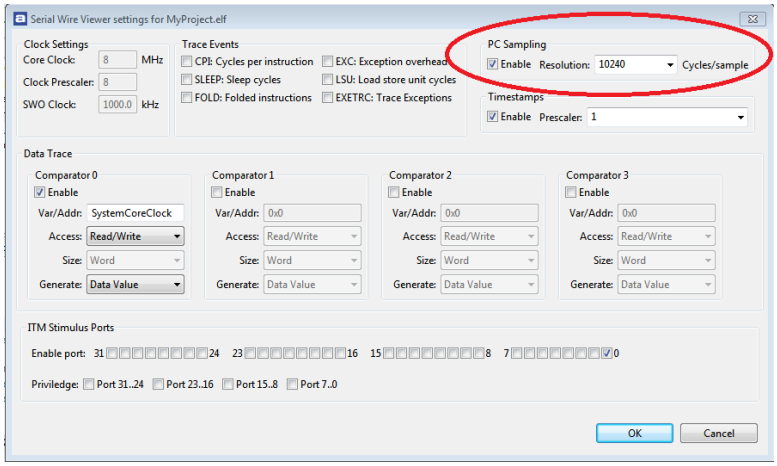


Figure 50 –Statistical profiling configuration

2. Open the Statistical Profiling view by selecting **Window, Show View, Other…** and then open the **Statistical Profiling** view. It will still be empty, since no data has been collected.

3. Push the red Start/Stop trace-button to send the configuration to the board.

4. When you start debugging, ***TrueSTUDIO®*** starts collecting statistics about what functions is reported by SWV.

5. Pause the debugging. The collected data are displayed in the view. The longer you run your debugging, the more statistics will be collected.

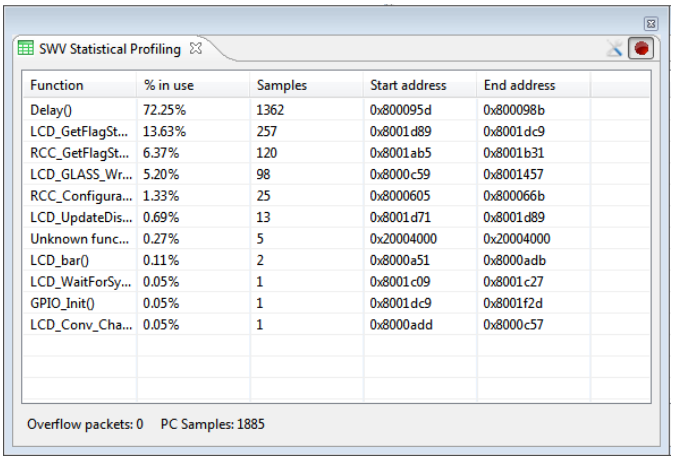


Figure 51 – Statistical profiling view

CONFIGURE PRINTF()

1. To be able to use `printf()` to send ITM-packages you will have to configure `syscalls.c`. If you didn't generate a syscalls-file at project-generation you will have to do the following steps:

   - In the **Project explorer**, Right click on the project and select New->Other...
   - Expand **System calls**.
   - Select "**Minimal System Calls Implementation**" and click next.
   - Click **Browse...** and select the src folder as new file container and click **OK**.
   - Click on **Finish** and verify that `syscalls.c` is added to your project.

2. Replace the `_write()` –function with the following code:

```
int _write(int file, char *ptr, int len)
{
  /* Implement your write code here, this is used by
puts and printf for example */
  int i=0;
  for(i=0 ; i<len ; i++)
    ITM_SendChar((*ptr++));
  return len;
}
```

CHANGE THE TRACE BUFFER SIZE

The incoming SWV-packages are saved in the Serial Wire Viewer Trace buffer. It has by default a max-size for 2 000 000 packages. If you want to trace more packages you have to increase that limit.

Select **Widows, Preferences**. In the dialog select **Run/Debug, Embedded C/C++ Application** and then **Serial Wire Viewer**.

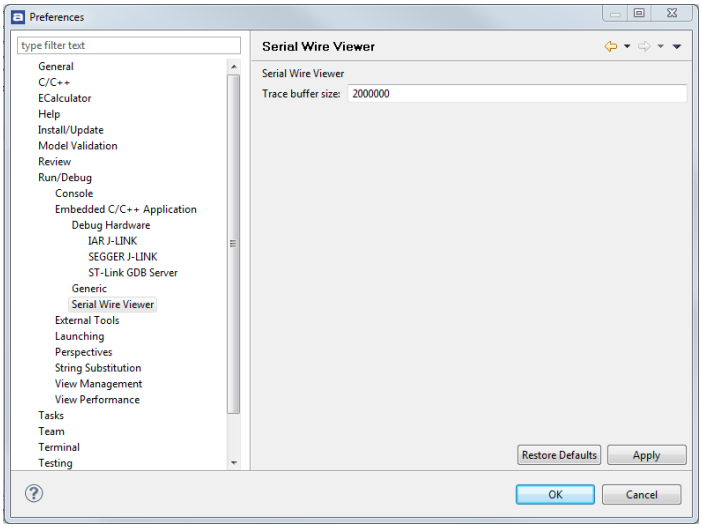


Figure 52 – Serial Wire Viewer preferences

The buffer is stored in the heap. The allocated heap can be displayed if you select **Windows**, **Preferences** and **General** and then enable "**Show heap status**". The current heap and allocated memory will now be displayed in the lower, right corner.

There is an upper limit for how much memory ***Atollic TrueSTUDIO®*** can allocate. Sometimes you might want to change it so that you can store more information during a debug-session.
· Go to your ***Atollic TrueSTUDIO®*** installation directory and then go into the *'ide'* folder.

· Edit the *TrueSTUDIO.ini* file and change the *-Xmx512m* parameter to the size in megabytes that you prefer.

· Save the file and try launching ***Atollic TrueSTUDIO®*** again.

COMMON REASONS WHY SWV ISN'T TRACING

- You have set the wrong **Core Clock** for the target. It is very important to select the right **Core Clock**.
  If you don't know the speed of the target's Core Clock, you can sometime find it by setting a breakpoint in the program-loop and then open the Variable-View. Then right click and select **Add Global Variables..,** and select **SystemCoreClock**. In other cases it can be found in the startup code and is called something like `SYSCLK`.

- SWV isn't enabled in your debug configuration.

- You haven't sent your configuration to the board.

- Some manufacturer, such as Energy Micro, has SWO disabled by default. In that case you have to enable it with a function-call, such as `DBG_SWOEnable()`.
- You are sending to much data to the SWO. Reduce the amount of things enabled for tracing.
- The probe, the gdbserver, the board or something else isn't supporting SWV.

STOPPING THE DEBUGGER

When the debug session is completed, the running application must be stopped.

1. Stop the target application by selecting the **Run, Terminate** menu command, or by clicking on the **Terminate** toolbar button in the **Debug** view.

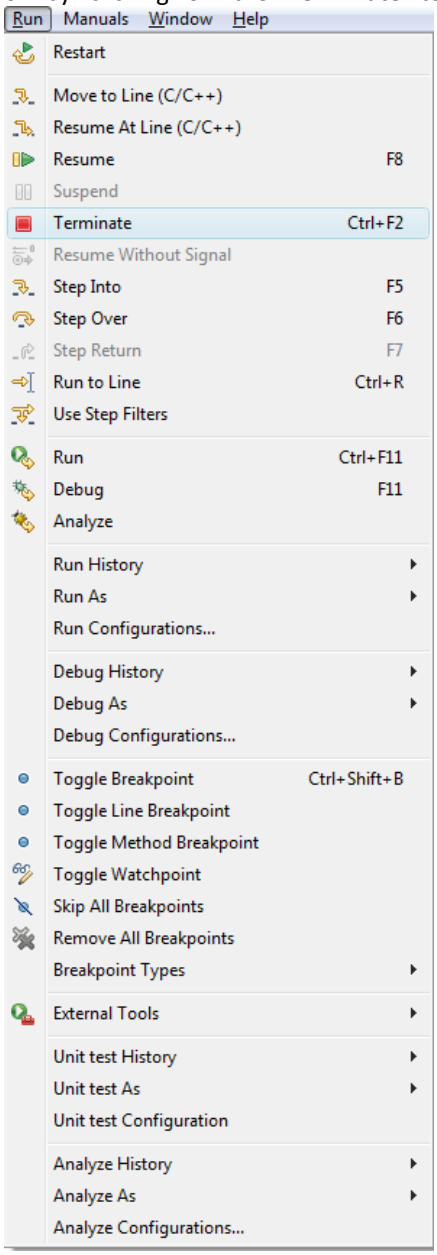


Figure 53 - The Terminate menu command

2. ***Atollic TrueSTUDIO®*** now automatically switch to the **C/C++ editing** perspective

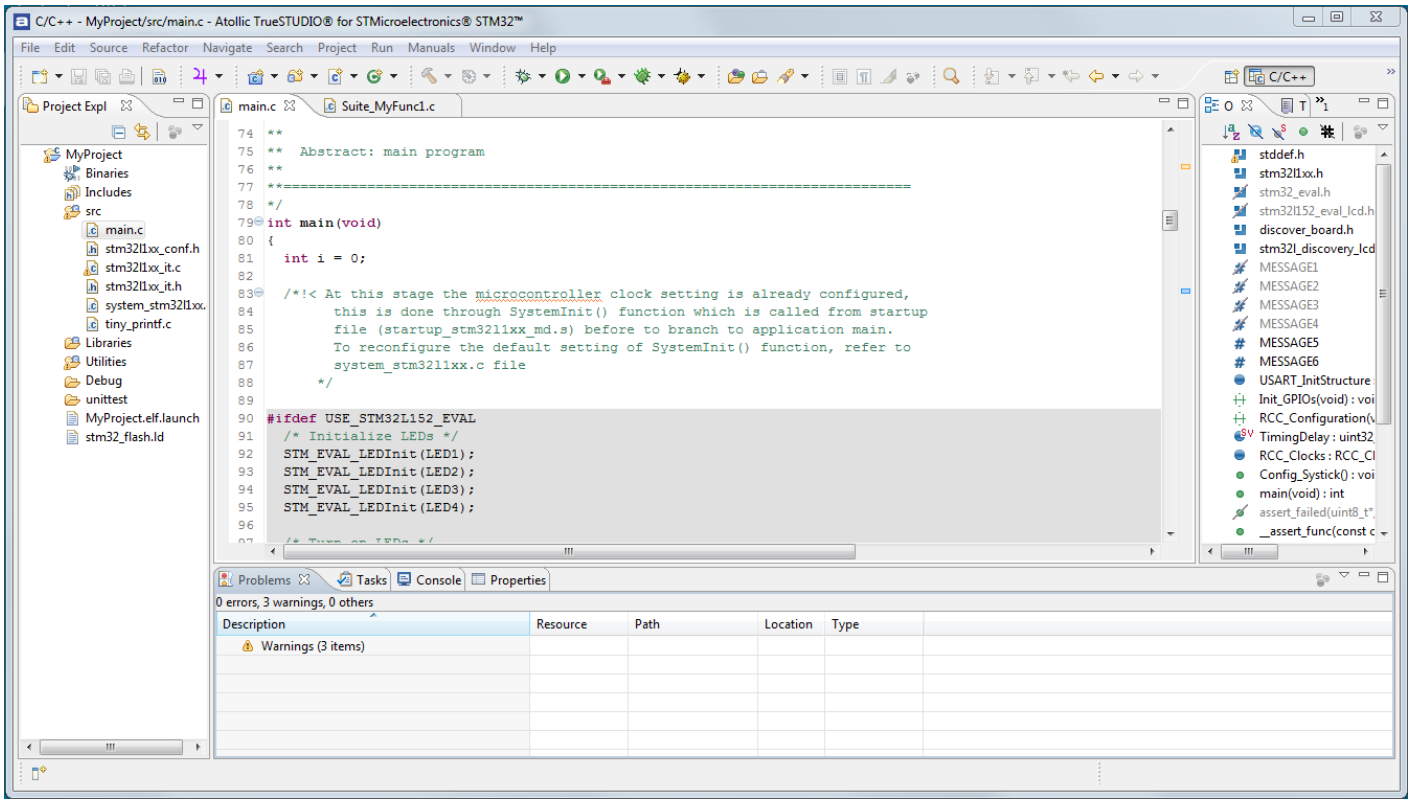


Figure 54 - C/C++ perspective

Section 2. STATIC CODE INSPECTION

This section provides information on how to use ***Atollic TrueINSPECTOR®***, which is an add-on product that can be purchased separately. A demo is included in all ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™*** *installations***.**

This section covers information on the following topics:

- Selecting coding standard rules
- Running a code inspection
- Analyzing the code inspection results

INTRODUCTION

Atollic TrueSTUDIO® for STMicroelectronics® STM32™ includes a demo of ***Atollic TrueINSPECTOR®***, a professional tool for static source code inspection that helps you to find potential bugs automatically. By using ***Atollic TrueINSPECTOR®***, you can easily improve the quality of your software product.

Atollic TrueINSPECTOR® performs static source code inspection and generates software metrics. The source code is validated against a database of formal coding standards, and coding constructs that are known to be error-prone are detected automatically, thus reducing the number of errors. This in turn reduces development/debugging/test time, reduces development cost, and improves the quality of your software product.

MISRA (The Motor Industry Software Reliability Association) was established as a collaboration between various vendors in the automotive industry, with the purpose to promote best practice in developing safety-critical systems in road vehicles and other types of embedded systems.

MISRA-C:2004 is a coding standard for the C programming language, developed by MISRA. The purpose is to identify a subset of the C language that improves safety, portability, reliability and maintainability. MISRA-C:2004 contains 141 coding rules, which limit the flexibility of how the source code can be written. By following the MISRA-C coding standard, you ensure that unsafe or unreliable coding constructs are not used in your software product, thus improving software quality and reducing the time spent on debugging.

Atollic TrueINSPECTOR® performs MISRA-C:2004 checking, automatically verifying source code compliance, and pointing out any code lines that break any of the coding standard rules. The analysis results are presented in textual form as well as in easy-to-understand graphical charts.

Developers can configure which rules to enable or disable at specific code inspection sessions, and reported rule violations are directly connected to the corresponding lines in the C/C++ editor. For each violation, ***Atollic TrueINSPECTOR®*** gives an example of code that triggers the violation, and provides an example of the recommended coding style that solves the reported problem.

Atollic TrueINSPECTOR® also generates software metrics, including cyclomatic values of code complexity. With a better understanding of what parts of your code is too complex, these sections can be rewritten using a less complex coding style. Avoiding complex code sections is a good way to improve maintainability and reducing the risk of errors, thus reducing development time and increasing product quality.

Atollic TrueINSPECTOR® generates various types of reports that can be exported in Microsoft® Word®, Microsoft® Excel®, Microsoft® PowerPoint®, HTML and PDF formats.

A fully working one-week license of ***Atollic TrueINSPECTOR®*** is included for commercial customers of ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Professional***, and a demo version is bundled with ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Lite***

SELECTING CODING STANDARD RULES

Before a code inspection session is started, you might want to configure what coding standard rules to use.

1. Select the **File, Properties** menu command to open the **Properties** dialog box, and drill down to **the Testing, Rule Setting** panel:

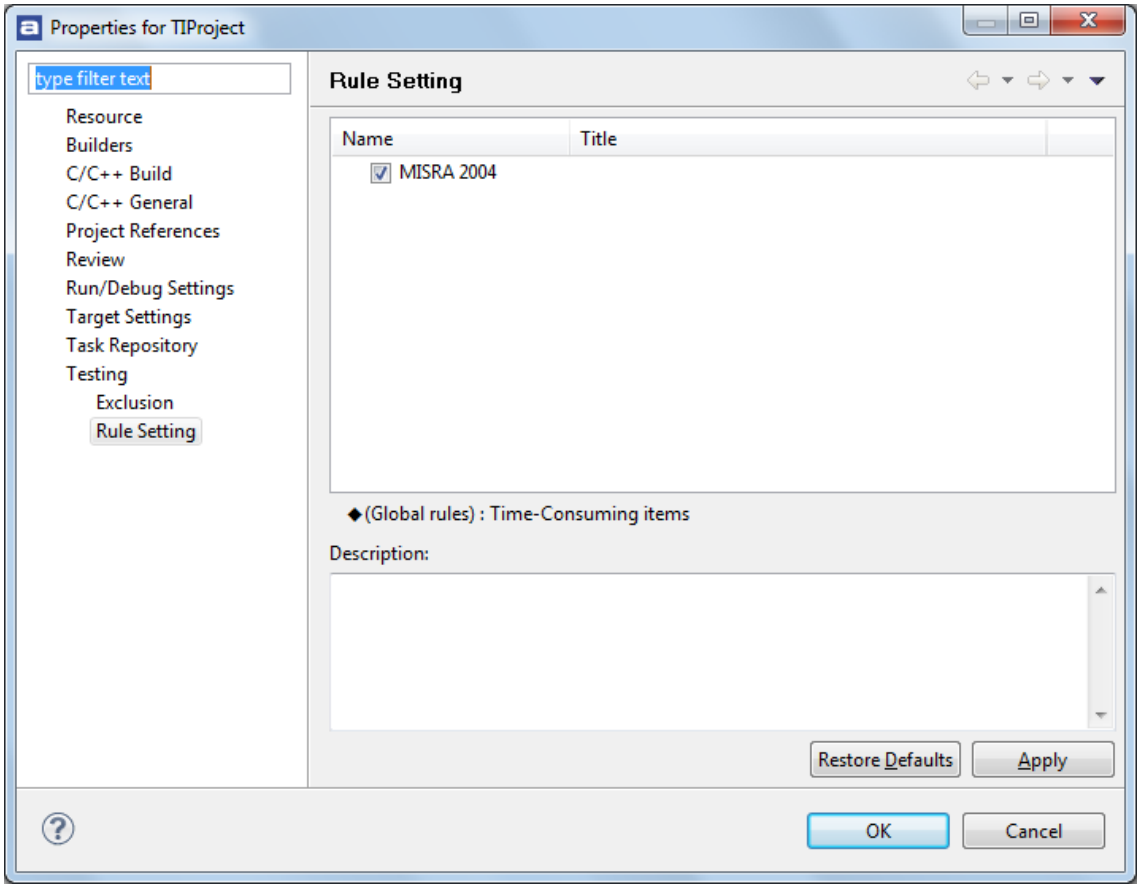


Figure 55 - The Rule setting dialog box

2. Expand the MISRA 2004 node, and select one of the rules:

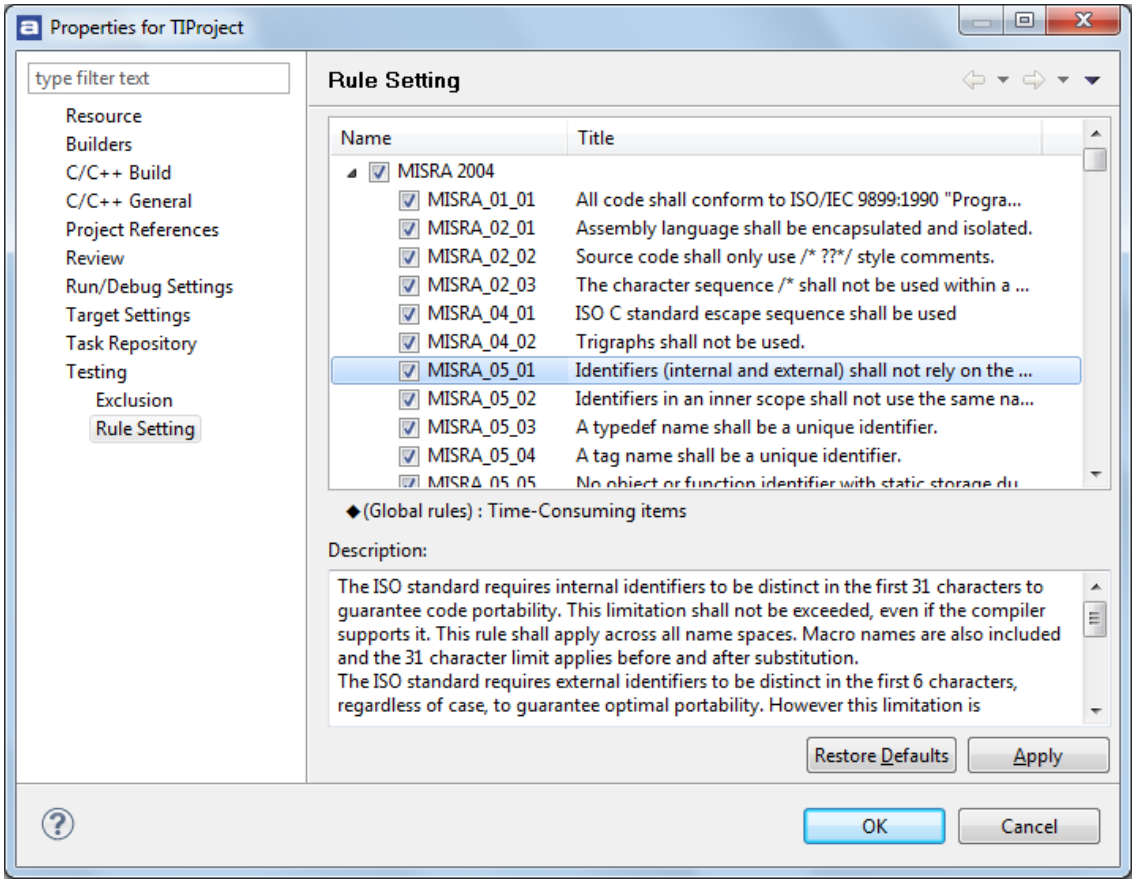


Figure 56 - Selecting rules

The **Description** field explains the purpose of the selected coding standard rule, and checking for any rule can be enabled or disabled as appropriate using the corresponding checkboxes. Click the **OK** button to leave the dialog box without doing any changes.

RUNNING A CODE INSPECTION

Once the coding standard rules to use have been configured, a source code inspection session can be started:

1. Select the **Project, Inspection** menu command:

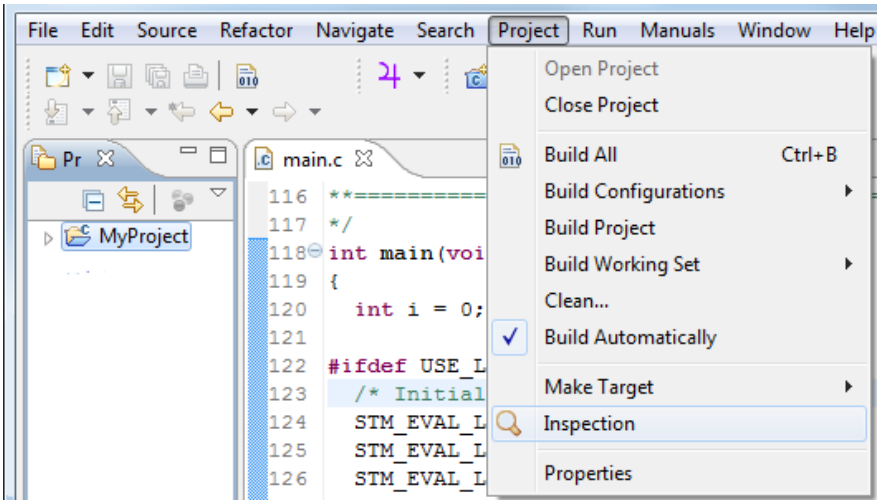


Figure 57 - Starting a code inspection session

2. ***Atollic TrueINSPECTOR®*** then starts to analyze the source code, and when completed, asks you for permission to switch to the **Inspection** perspective.

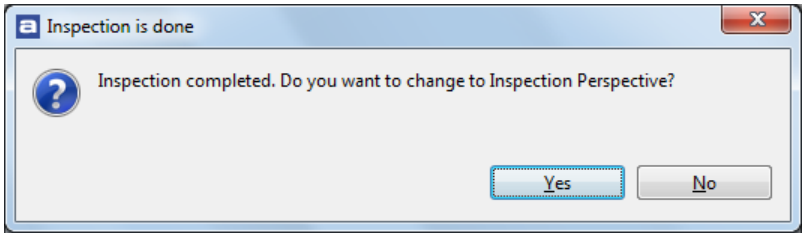


Figure 58 - The Perspective swap message box

3. Click the **Yes** button to switch to the **Inspection** perspective.

The **Inspection** perspective is now displayed, with several new docking views specific to source code inspection:

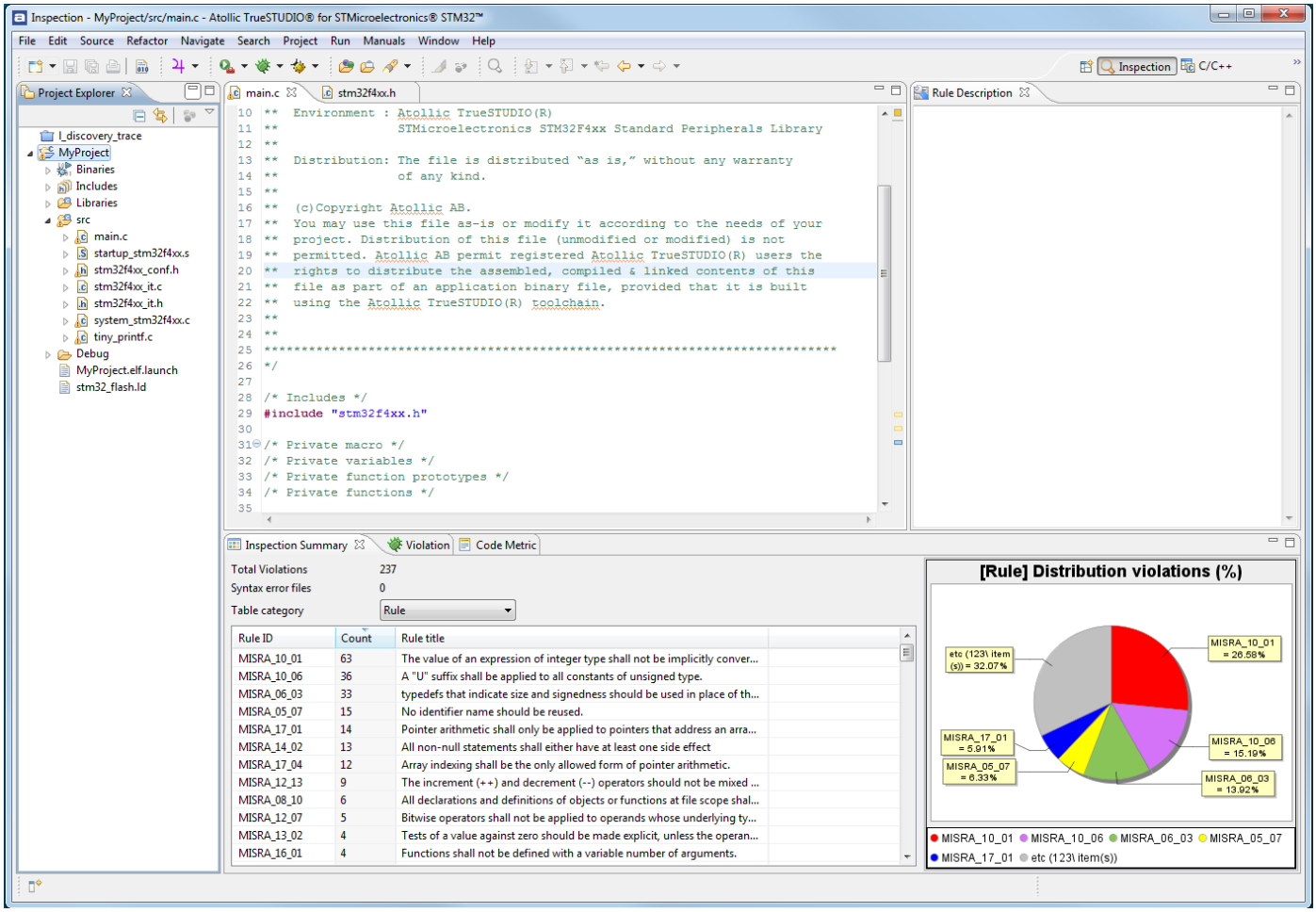


Figure 59 - The Inspection perspective

ANALYZING THE INSPECTION RESULTS

Once a code inspection session has been performed, a number of docking views display the inspection results:

- The **Inspection summary** view
- The **Violation** view
- The **Code Metric** view
- The **Rule Description** view

THE INSPECTION SUMMARY VIEW

The **Inspection Summary** view gives an overview of the inspection results, including a sorted text list and a graphical pie chart for the most important results. By selecting different values for the **Table category** drop-down list, the view can display other information too.

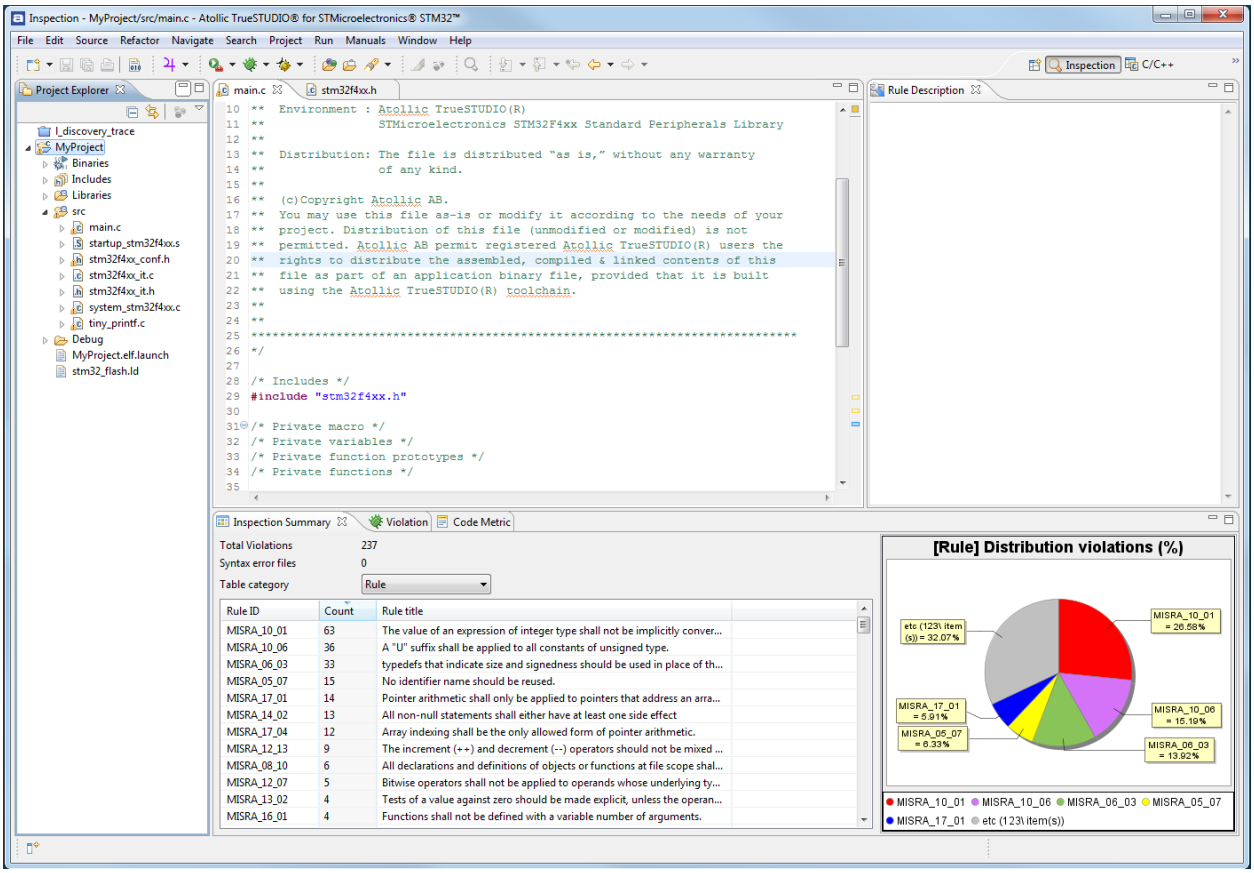


Figure 60 - The Inspection Summary view

THE RULE DESCRIPTION VIEW

By clicking on any rule in various other views, the **Rule Description** view will explain the details of that rule, including bad coding practice that triggers the violation and a recommendation for alternative, better, coding practice. The **Rule Description** view is thus a very important tool to learn and improve the coding style over time.

Select a rule in the **Inspection Summary** view list to activate the **Rule Description** view:

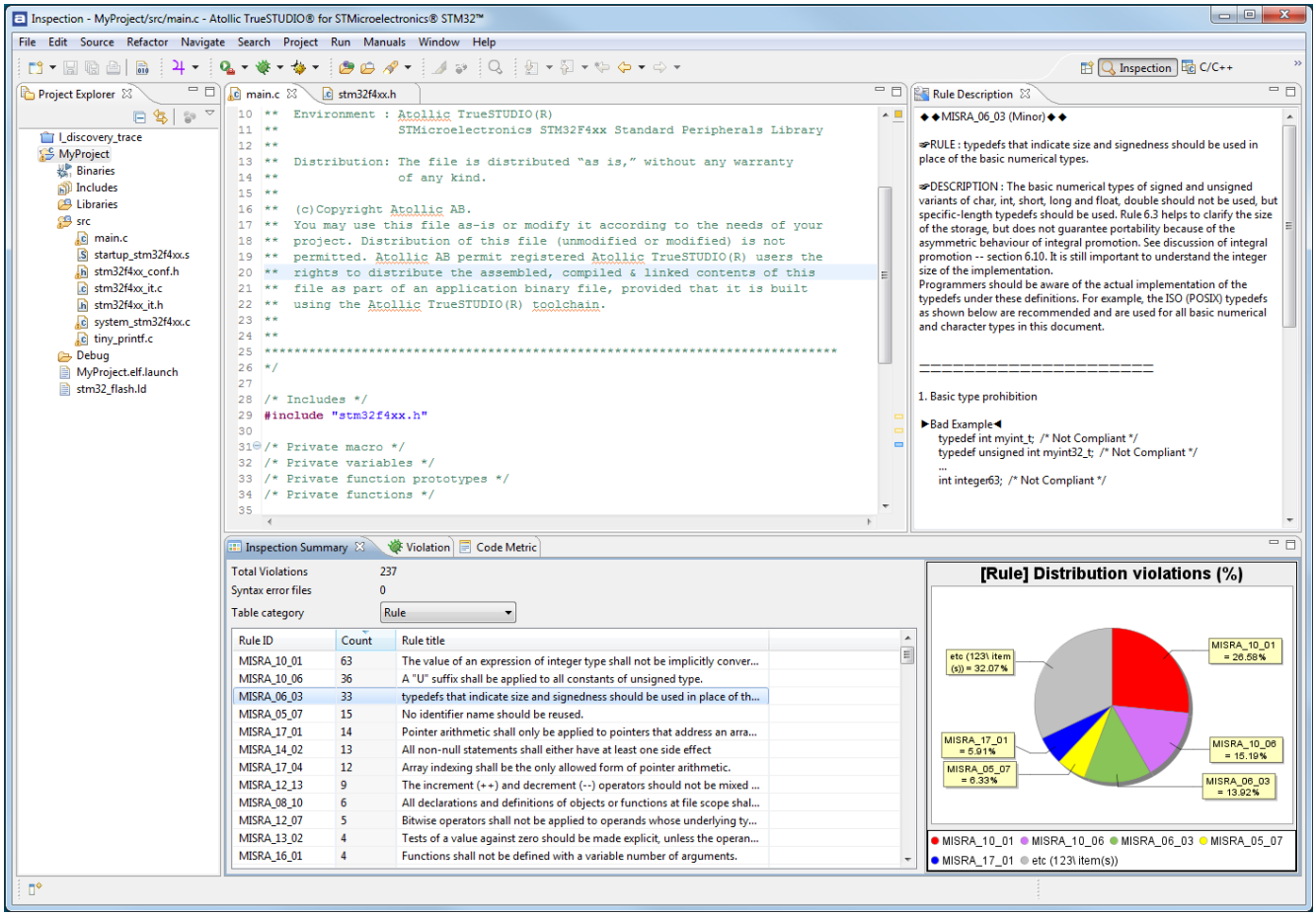


Figure 61 - The Rule Description view

VIOLATION VIEW

The **Violation** view lists the actual violations (connected to specific file:line instances) sorted by rule number:

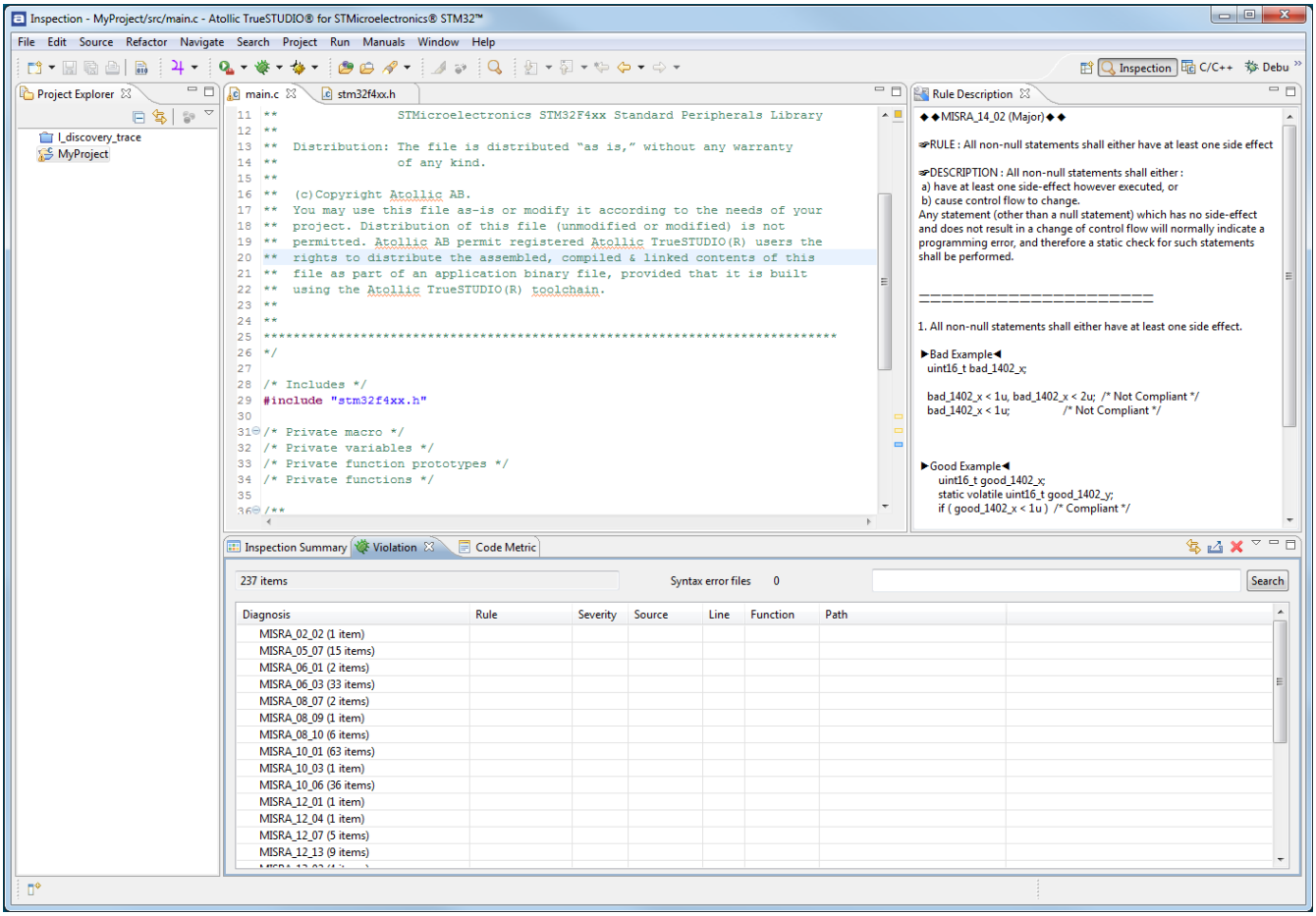


Figure 62 - The Violation view

Expand the desired rule number to see all violations of that rule type:

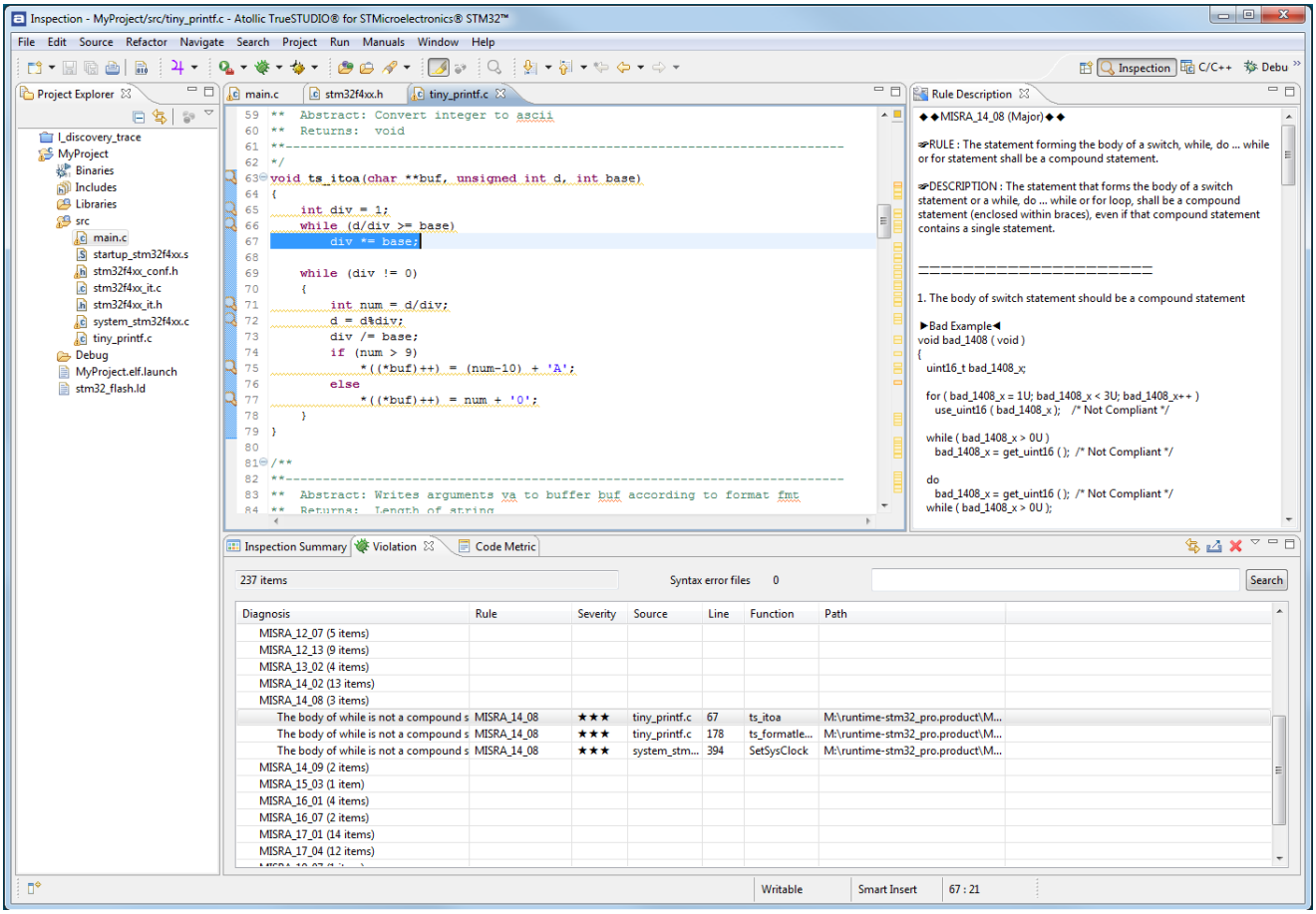


Figure 63 - Individual violations

By double-clicking on any violation, the corresponding file:line is opened automatically in the editor. Then click on the **Generate report** toolbar button:

A dialog box is now displayed. Enter the name of the report(s), the desired output directory, and the file formats of choice:

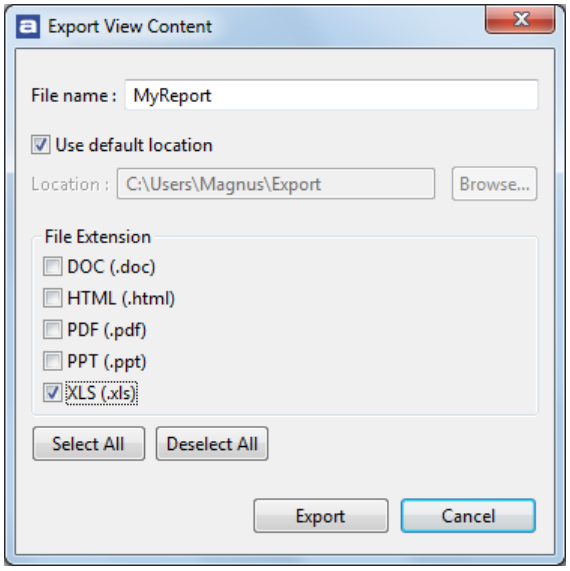


Figure 64 - The Report generator dialog box

Click on the **Export** button to generate the reports. Once the report(s) are exported, ***Atollic TrueINSPECTOR®*** gives you the option to open the folder where the reports were generated:

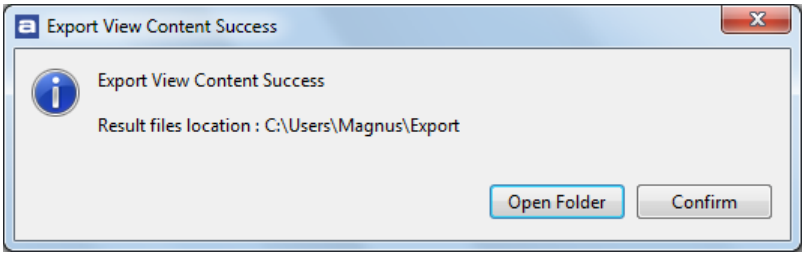
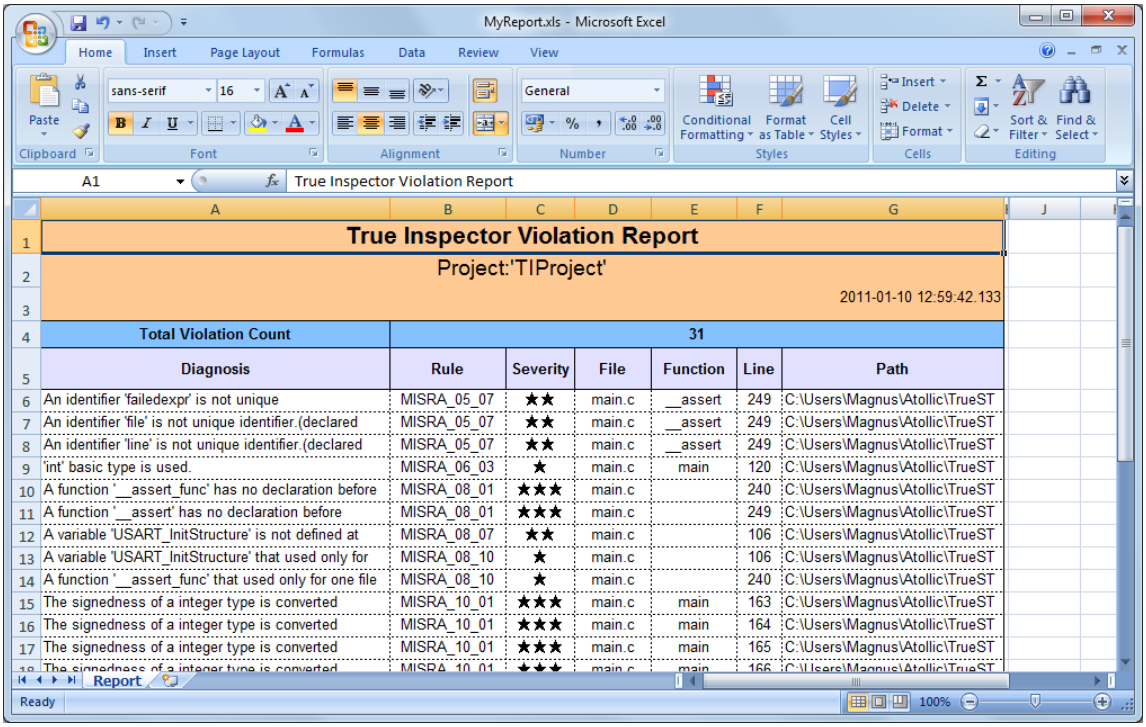


Figure 65 - The Open Folder message box

Click the **Open Folder** button to view the report files in the export folder. Double-click to open any of the generated reports to open it in a suitable viewer (such as Adobe® Acrobat® Reader or Microsoft® Excel®):



True Inspector Violation Report

Project:'TIProject'

2011-01-10 12:59:42.133

Total Violation Count						31
Diagnosis	**Rule**	**Severity**	**File**	**Function**	**Line**	**Path**
An identifier 'failedexpr' is not unique	MISRA_05_07	★★	main.c	__assert	249	C:\Users\Magnus\Atollic\TrueST
An identifier 'file' is not unique identifier.(declared	MISRA_05_07	★★	main.c	__assert	249	C:\Users\Magnus\Atollic\TrueST
An identifier 'line' is not unique identifier (declared	MISRA_05_07	★★	main.c	__assert	249	C:\Users\Magnus\Atollic\TrueST
'int' basic type is used.	MISRA_06_03	★	main.c	main	120	C:\Users\Magnus\Atollic\TrueST
A function '__assert_func' has no declaration before	MISRA_08_01	★★★	main.c		240	C:\Users\Magnus\Atollic\TrueST
A function '__assert' has no declaration before	MISRA_08_01	★★★	main.c		249	C:\Users\Magnus\Atollic\TrueST
A variable 'USART_InitStructure' is not defined at	MISRA_08_07	★★	main.c		106	C:\Users\Magnus\Atollic\TrueST
A variable 'USART_InitStructure' that used only for	MISRA_08_10	★	main.c		106	C:\Users\Magnus\Atollic\TrueST
A function '__assert_func' that used only for one file	MISRA_08_10	★	main.c		240	C:\Users\Magnus\Atollic\TrueST
The signedness of a integer type is converted	MISRA_10_01	★★★	main.c	main	163	C:\Users\Magnus\Atollic\TrueST
The signedness of a integer type is converted	MISRA_10_01	★★★	main.c	main	164	C:\Users\Magnus\Atollic\TrueST
The signedness of a integer type is converted	MISRA_10_01	★★★	main.c	main	165	C:\Users\Magnus\Atollic\TrueST

Figure 66 - View the Generated report

THE CODE METRICS VIEW

In addition to validating the source code against a formal coding standard, ***Atollic TrueINSPECTOR®*** also provides source code metrics and code complexity measurements.

The **Code Metric** view display statistics on the source code. The default mode is to display information on module level:

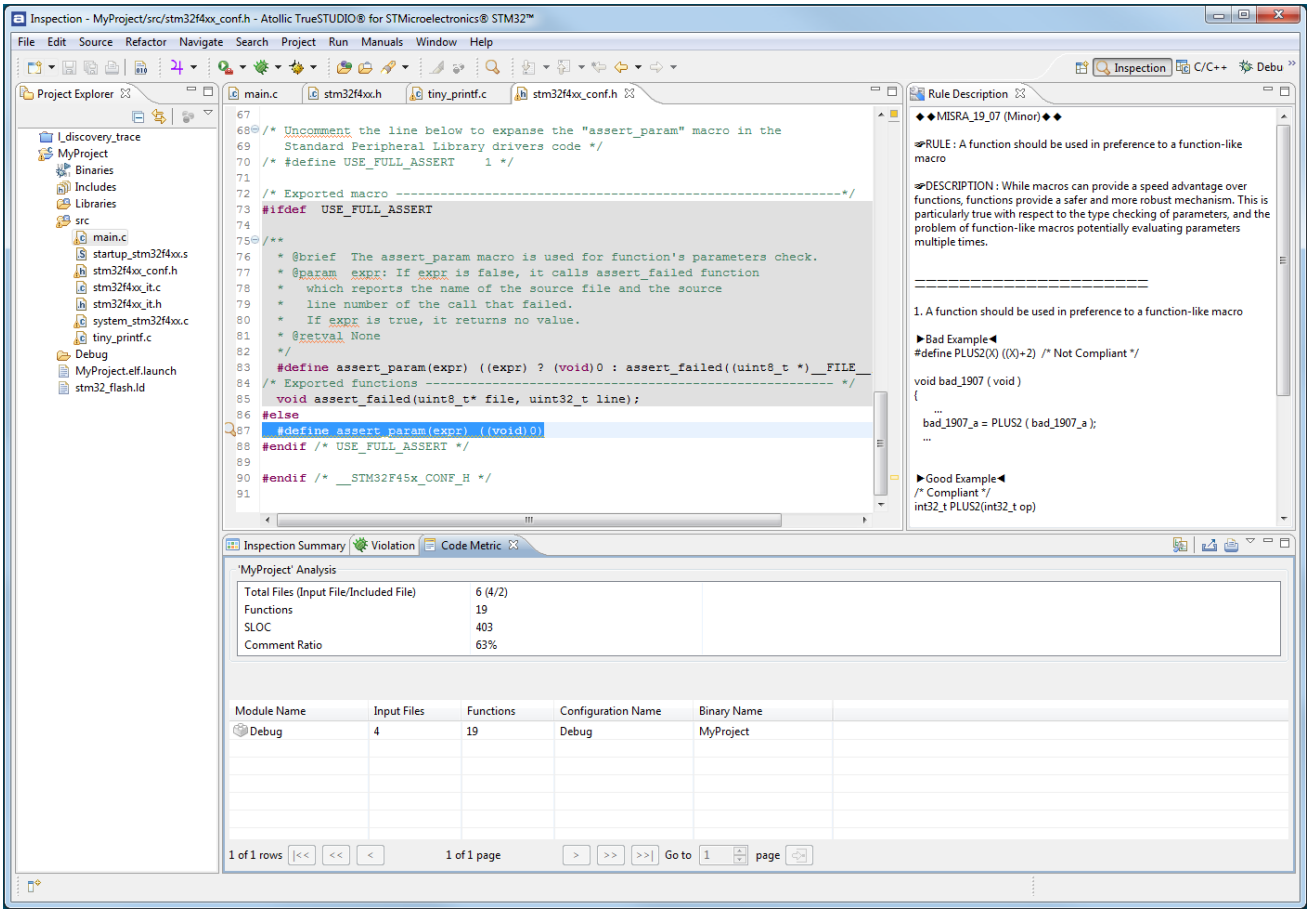


Figure 67 – The Code Metric view (module mode)

Click on the **Metric mode** toolbar button in the Code Metric view to switch to function mode:

Atollic TrueINSPECTOR® now displays detailed information for each file:

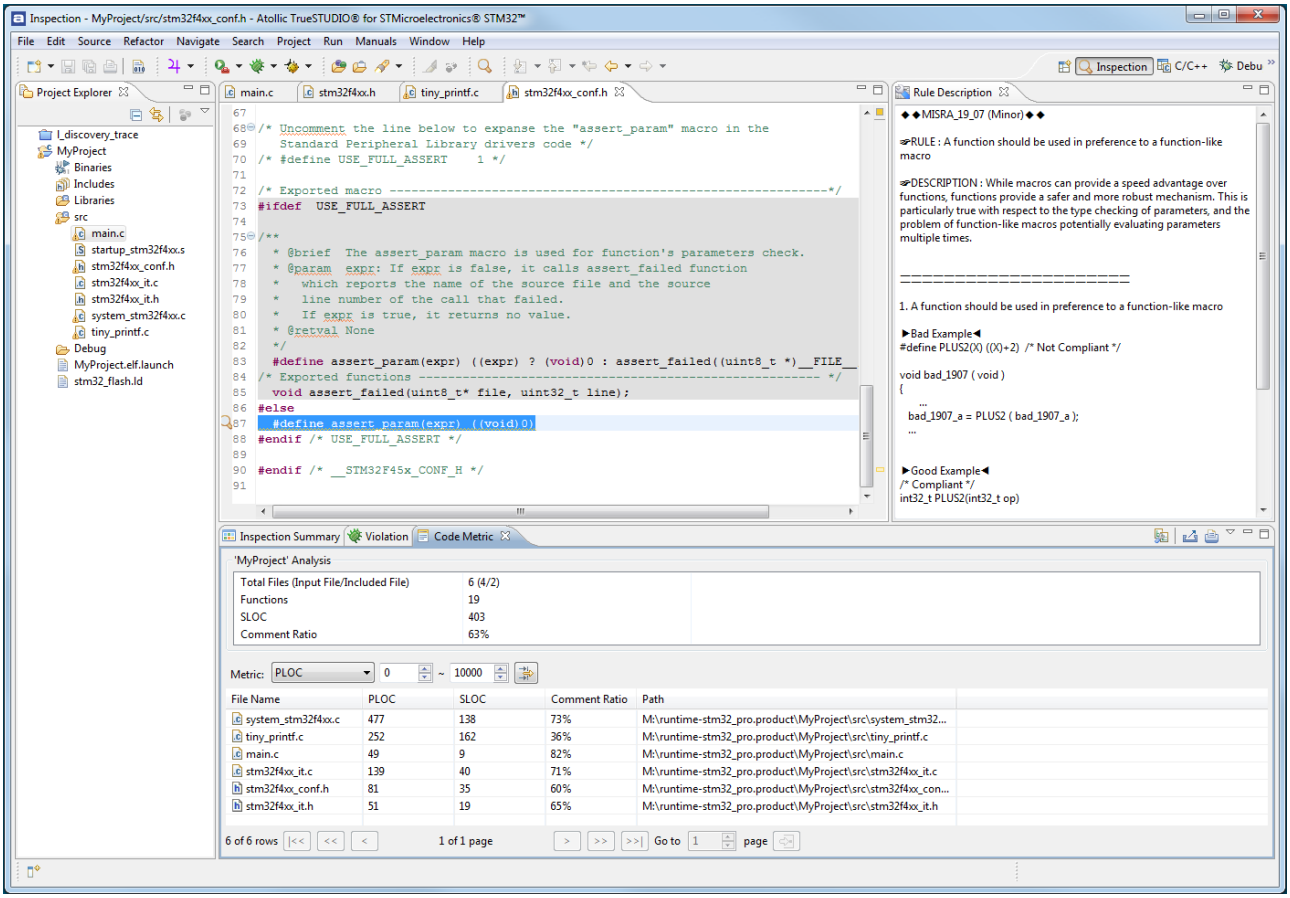


Figure 68 - The Code Metric view (file mode)

Click on the **Metric mode** toolbar button one more time to switch to function mode:

Atollic TrueINSPECTOR® now displays detailed information for each function:

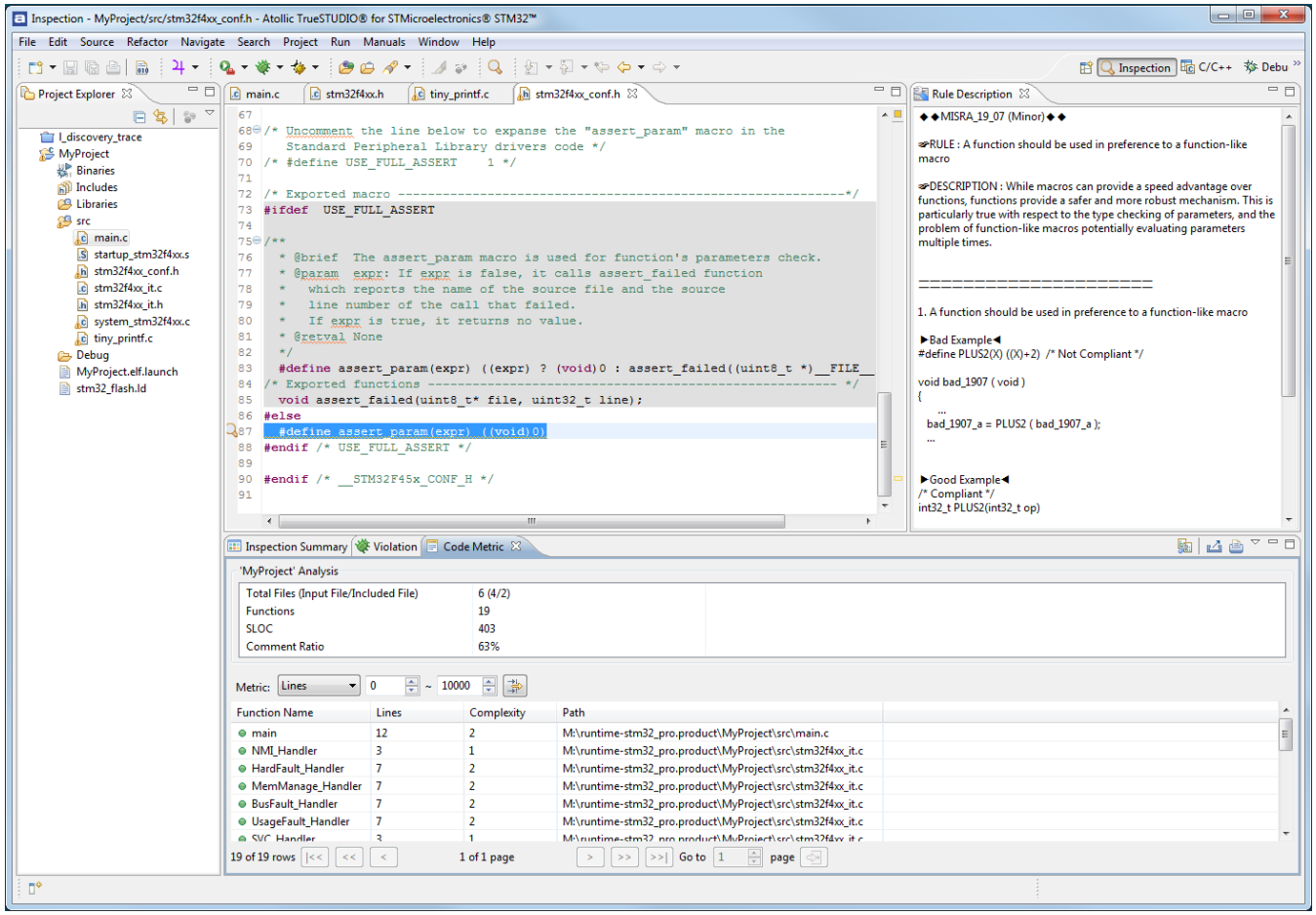


Figure 69 - The Code Metric view (function mode)

Notice the cyclomatic value of code **Complexity** is listed for each function too. It is recommended to refactor or simplify the implementation of functions with a very high complexity value.

Atollic recommends rewriting functions with a high complexity level in order to reduce risk of programming errors and improving maintainability.

Finally, click on the **Generate report** toolbar button:

A dialog box is now displayed. Enter the name of the report(s), the desired output directory, and the file formats of choice:

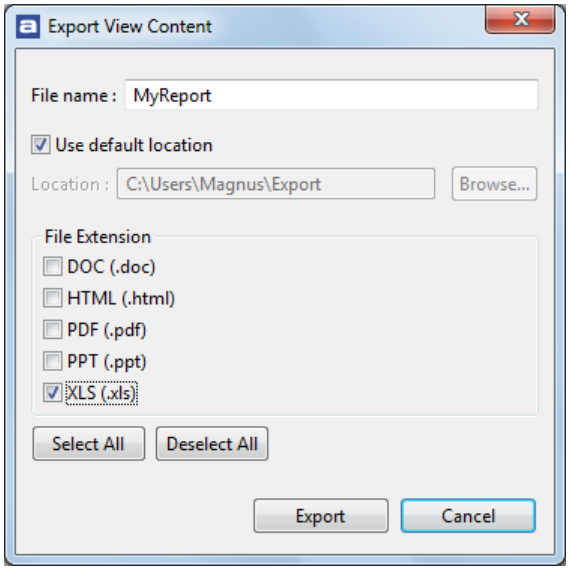


Figure 70 - The Report generator dialog box

Click on the **Export** button to generate the reports. Once the report(s) are exported, ***Atollic TrueINSPECTOR®*** gives you the option to open the folder where the reports were generated:

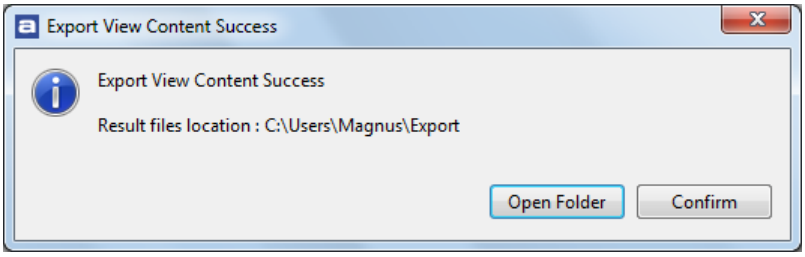


Figure 71 - The Open Folder message box

Click the **Open Folder** button to view the report files in the export folder. Double-click to open any of the generated reports to open it in a suitable viewer (such as Adobe® Acrobat® Reader or Microsoft® Excel®).

SUMMARY

By using the source code inspection features of ***Atollic TrueINSPECTOR®***, developers get many benefits that can improve software quality, in particular validation against industry best-practice coding standards and source code metrics including code complexity measurements.

Atollic strongly recommend you deploy ***Atollic TrueINSPECTOR®*** in your project, thus improving your software quality to a completely new level!

SECTION 3. CODE COVERAGE ANALYSIS

This section provides information on how to use ***Atollic TrueANALYZER®***, which is an add-on product that can be purchased separately. A demo is included in all ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™*** installations***.***

Commercial buyers of ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Professional*** get a one week fully working license of ***Atollic TrueANALYZER®***, while ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Lite*** users get a demo version.

This section covers:

- Different types of code coverage analysis
- Starting a code coverage analysis sessions
- Retrieving the result

INTRODUCTION

Atollic TrueANALYZER® provides advanced features for professional code coverage analysis up to the level of Modified condition/Decision coverage (MC/DC) as required by RTCA DO-178B for flight control system software.

A fully working one week demo-license is included in ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Professional.***

WHY PERFORM CODE COVERAGE ANALYSIS?

All embedded developers are used to debugging; the process used to understand the behavior of a program such that an error can be corrected. Debugging however, requires that an error has been detected in the first place. Programming errors that you do not know exist cannot be debugged or corrected.

Code coverage analysis is a powerful tool for finding more of the bugs that most likely do exist in your software – whether you are aware of them or not. As such, a code coverage analysis tool can help you find more programming errors, which enables you to release a software product of better quality.

More technically, code coverage analysis finds areas in your program that is not covered by your test cases, enabling you to create additional tests that cover otherwise untested parts of your program. These additional tests can detect more bugs.

It is thus important to understand that code coverage helps you understand the quality of your test procedures, not the quality of the code itself (although that happens indirectly due to the better test procedures you put in place when you use code coverage analysis).

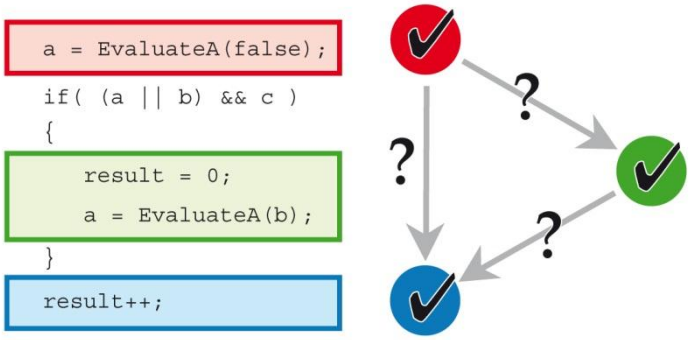
Atollic TrueANALYZER® automates these work tasks.

DIFFERENT TYPES OF ANALYSIS

There are many different types of code coverage analysis, some very basic and others that are very rigorous and complicated to perform without tool support. Different types of code coverage analysis can be classified formally:

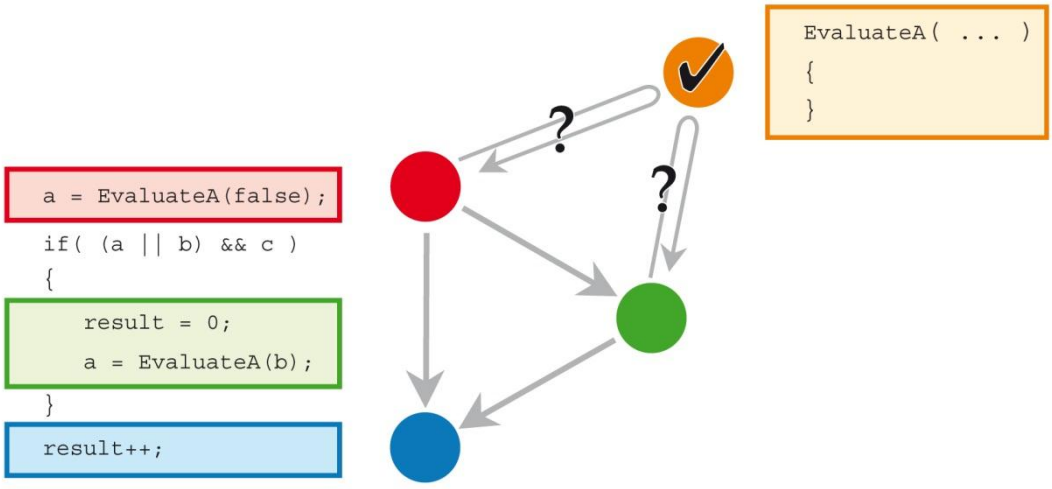
STATEMENT COVERAGE

This very basic type of code coverage analysis is sometimes included in standard debuggers. Statement coverage is a very basic type of code coverage analysis. It can only verify what blocks of code (=set of statements) have been executed, not under what circumstances it happened.



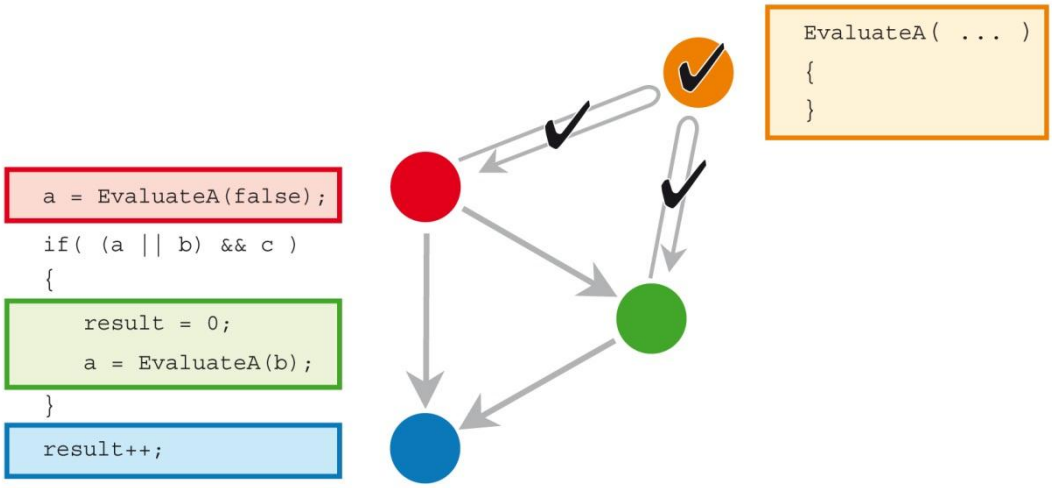
FUNCTION COVERAGE

This basic type of code coverage analysis is sometimes included in standard debuggers too. Function coverage can only report whether a function has been called or not, it does not say anything about what was executed inside it, or how or why the function was called. And it certainly does not test all the potential calls into the function.



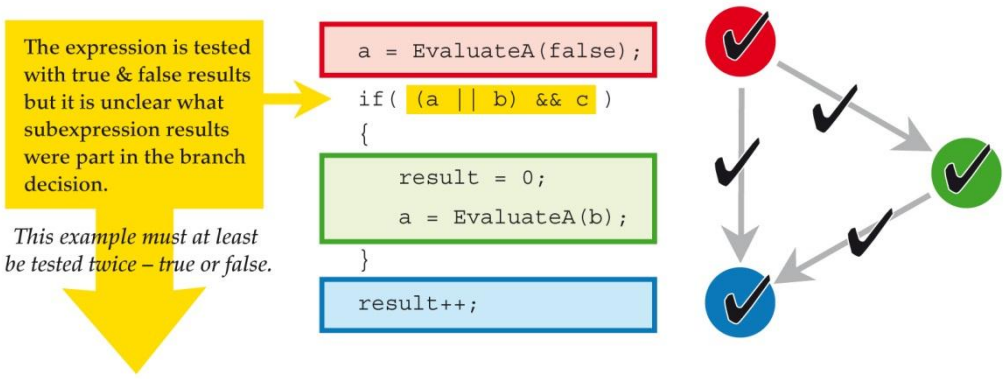
FUNCTION CALL COVERAGE

Function call coverage measures how many of the potential calls to a function have been made. For example, a function might be called from several hundred different places. To reach 100% Function call coverage, every call to the function must be executed at least one time.



BRANCH COVERAGE

Branch coverage is a more advanced type of code coverage analysis. Branch coverage ensures that all code blocks have been exercised and that all branch paths have been executed. It does not consider how a branch decision was taken in a complex branch expression.



MODIFIED CONDITION/DECISION COVERAGE

Modified condition/decision coverage (MC/DC) is a very advanced type of code coverage analysis, and basically builds on top of Condition/Decision coverage to create an even more rigorous code coverage analysis.

To fulfill MC/DC coverage for a code section, the following must apply:

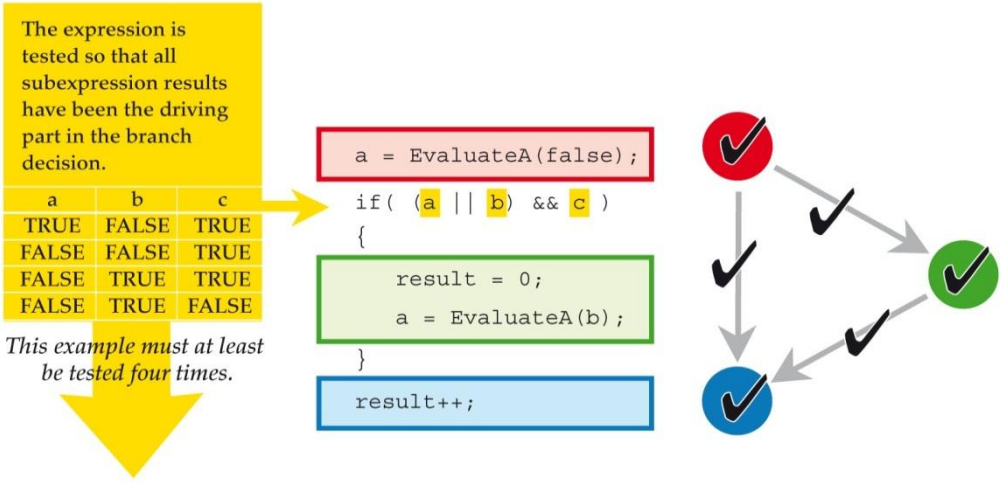
- Every entry and exit point must be invoked at least one time.
- Every condition in all decisions must take all possible outcomes at least one time.
- Every decision has taken all possible outcomes at least one time.
- Each condition in a decision is shown to independently affect the decision outcome.
- A condition is shown to independently affect a decision outcome by varying only that condition while holding all other conditions fixed.

Safety-critical software is often required to be tested such that it fulfills MC/DC-level code coverage analysis successfully.

RTCA DO-178B for example, requires MC/DC-level testing of airborne software of “Level A” criticality, which is the most safety-critical part of airborne software, such as the flight control or avionics system.

But any software project benefit from rigorous code coverage analysis:

- Companies with a good reputation want to avoid bad reputation on the market by releasing products of poor quality.
- Products that are very difficult or expensive to field-upgrade are good candidates for rigorous testing.
- Products that are produced in high volume ought to be tested formally.
- And of course, safety-critical or semi-safety-critical products should be tested formally.



TOOL SUPPORT

Atollic TrueANALYZER® is a very powerful tool for code coverage analysis. It performs the following types of code coverage:

- Statement coverage
- Function coverage
- Function call coverage
- Branch coverage
- Modified condition/decision coverage

To achieve such rigorous code coverage analysis, ***Atollic TrueANALYZER®*** performs the following tasks:

- Analyze the application source code
- Instrument & recompile a copy of the application source code
- Download the instrumented application to target using a JTAG probe
- Execute the instrumented application in the target board
- On request, upload latest analysis result to the PC for visualization

PERFORMING CODE COVERAGE ANALYSIS

Follow the instructions below to prepare for, and perform, a code coverage analysis session of your application on your target board.

PREPARATIONS

Before performing a code coverage analysis section, ***Atollic TrueANALYZER®*** needs to have a connection to the STM32 target board. This is commonly done using the same JTAG probe you use for debugging in ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™.***

To prepare for code coverage analysis using an ST-LINK or SEGGER JTAG probe connected to your electronic board, perform the following steps:

1. Verify that the RAM/FLASH switch on the board (if available) is set in the right mode (it must match the ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™*** *project* configuration).
2. Find out if your board supports JTAG-mode or SWD-mode debug connectors. The JTAG probe settings must be configured accordingly.
3. Connect the JTAG cable between the JTAG dongle and the electronic board.
4. Connect the USB cable between the PC and the JTAG dongle.
5. Make sure the electronic board has proper power supply.

Once the steps above are performed, a code coverage analysis session in ***Atollic TrueANALYZER®*** can be started.

STARTING CODE COVERAGE ANALYSIS

Starting a code coverage session is very easy. Make sure that you have a working project loaded in ***Atollic TrueSTUDIO®.*** I.e., the project shall be created, compile without problems, and it shall be possible to download the application to the target board using the JTAG probe and the ***Atollic TrueSTUDIO®*** debugger.

Unless the application can be debugged and executed successfully in target using the ***Atollic TrueSTUDIO®*** debugger and the associated JTAG probe, code coverage analysis will not work either. Make sure debugging works as expected before trying to start code coverage analysis sessions.

Provided that the pre-requisites outlined above are fulfilled, perform the following steps to start a code coverage analysis session:

1. Select the project by clicking on the Project name (root node) in the ***Atollic TrueSTUDIO®*** **Project Explorer** view.

2. In the **Run** menu, select **Analyze As** and **Embedded C/C++ Application**:

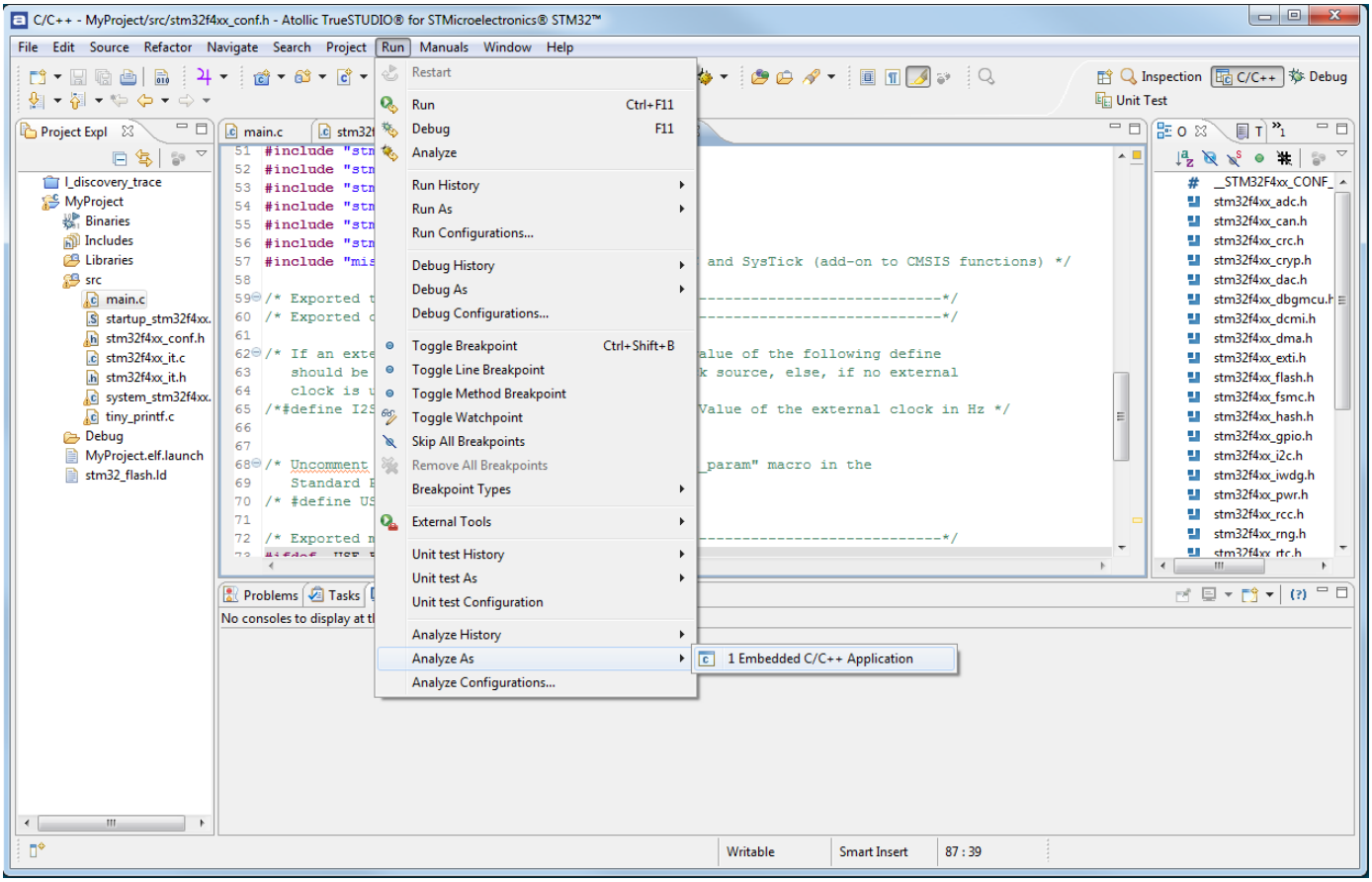
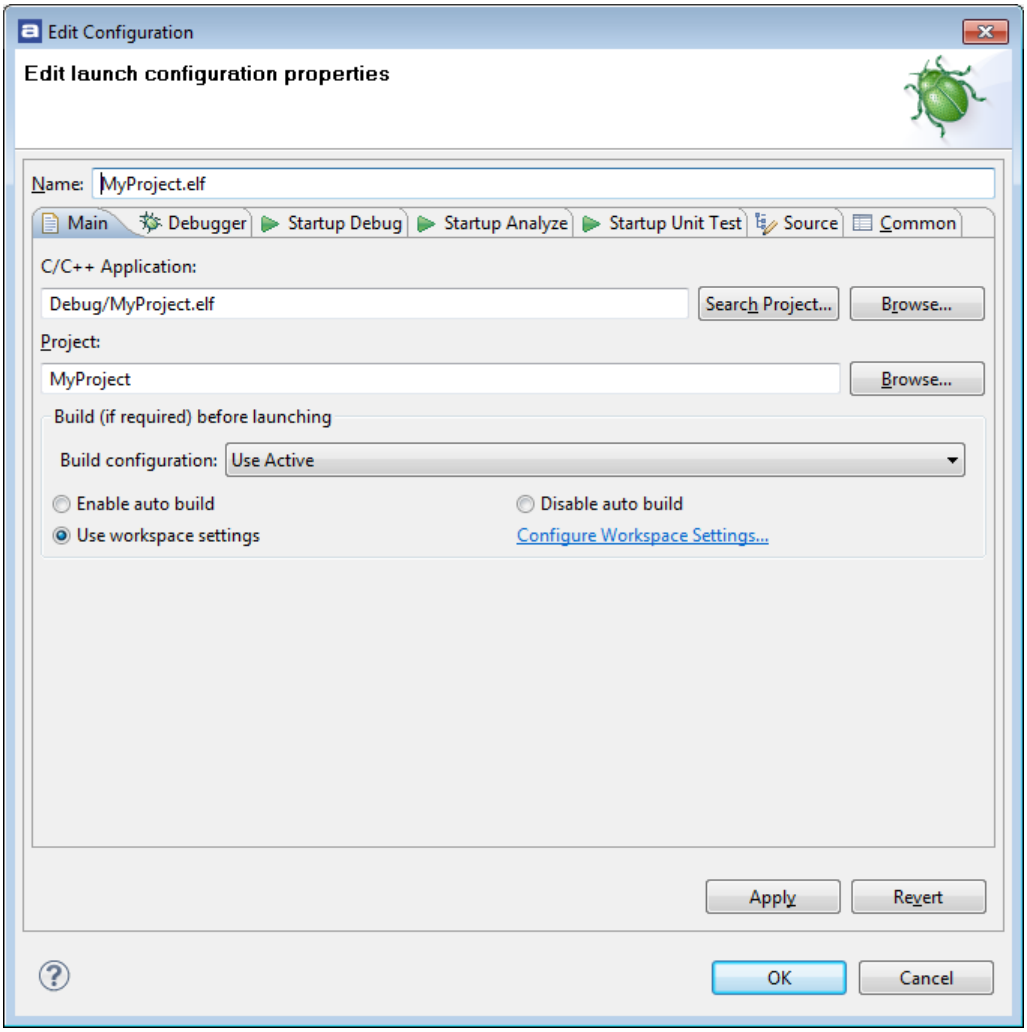
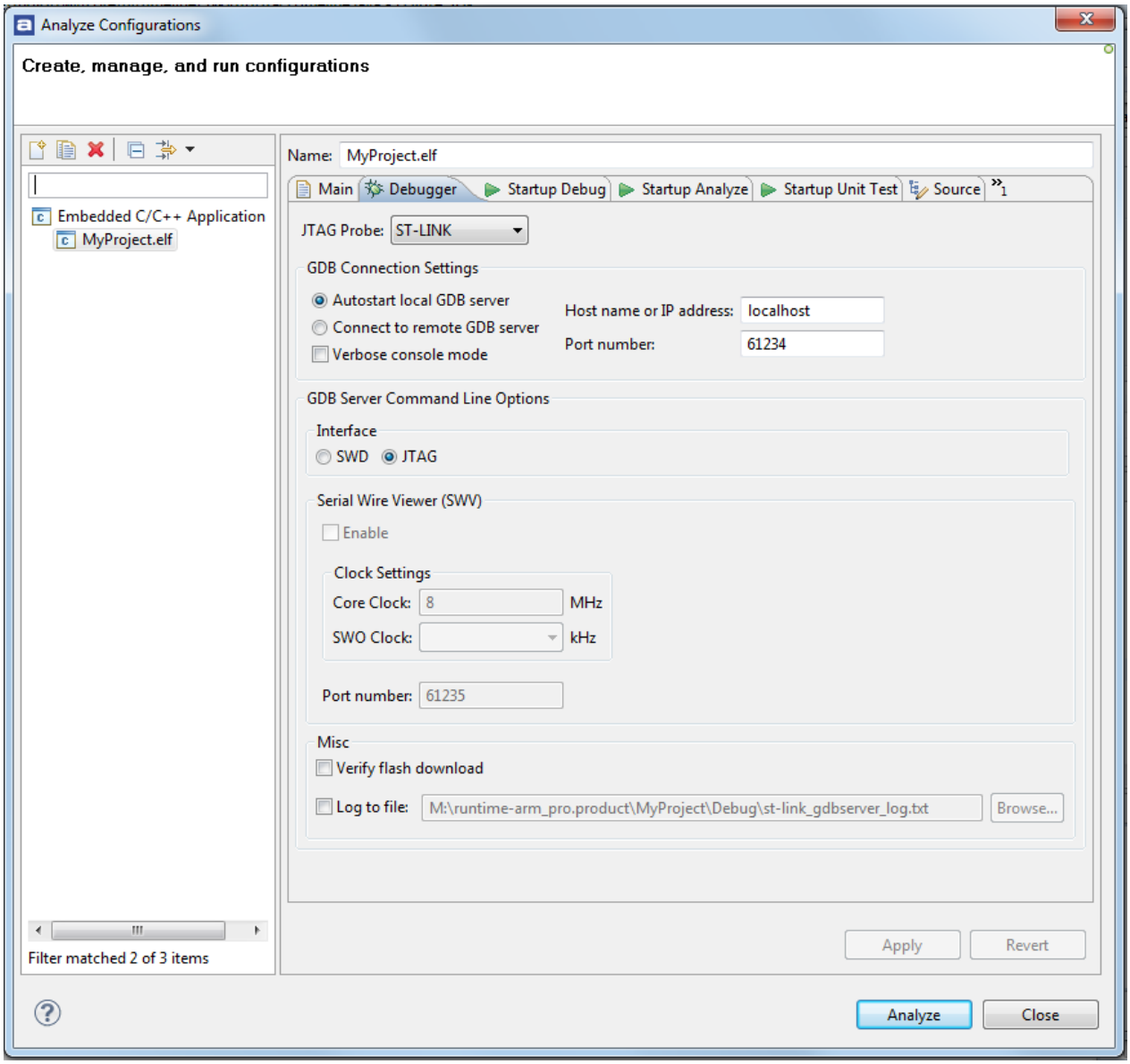


Figure 72 - Starting a code coverage analysis session

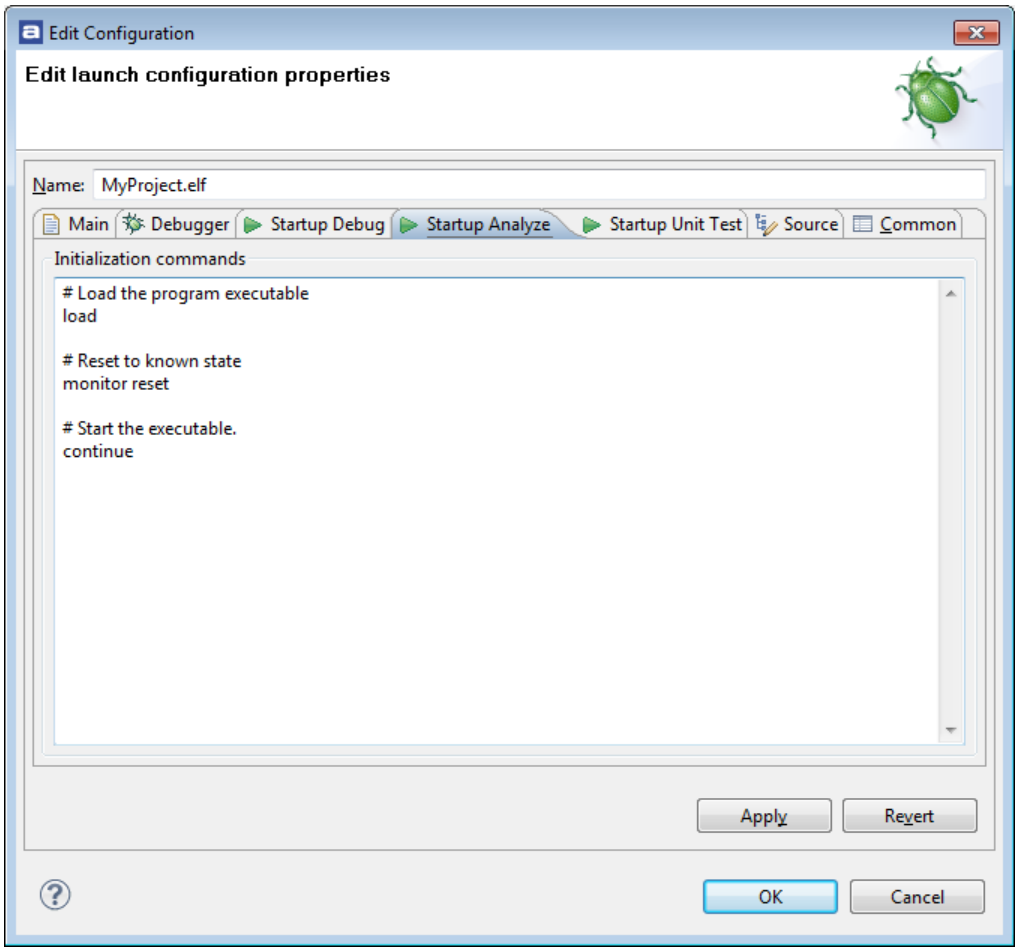
3. The launch configuration dialog box is displayed:



4. Select the **Debugger** tab and make sure the settings are correct for your board and JTAG probe:



5. Select the **Startup Analyze** tab. Board or JTAG probe specific debugger initializations can be made if necessary:



6. Click **OK** to start analysis of your source code, and instrumentation as well as re-compilation of a copy of your application. This may take several minutes:

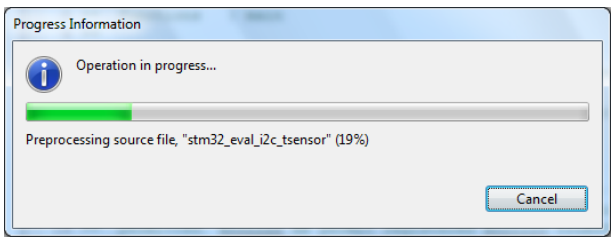


Figure 73 - Analysis, instrumentation and re-compilation

7. Once the analysis, instrumentation and re-compilation is completed, ***Atollic TrueANALYZER®*** ask you for permission to switch to the **Coverage** perspective:

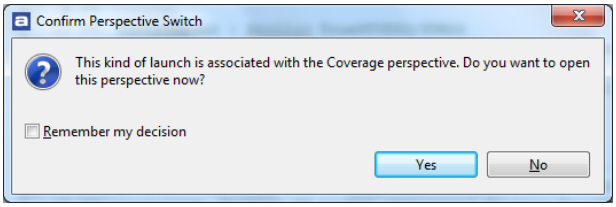


Figure 74 - Change to Code coverage perspective

8. Click **Yes** to switch into the **Coverage** perspective. In the **Analysis** view, the text **localhost:<port number>** is displayed in the tree. This tree item is the "handle" to target connection (JTAG probe connection to the target board). Initially this tree item also displays information on downloading the instrumented application to target, or other connection related information.

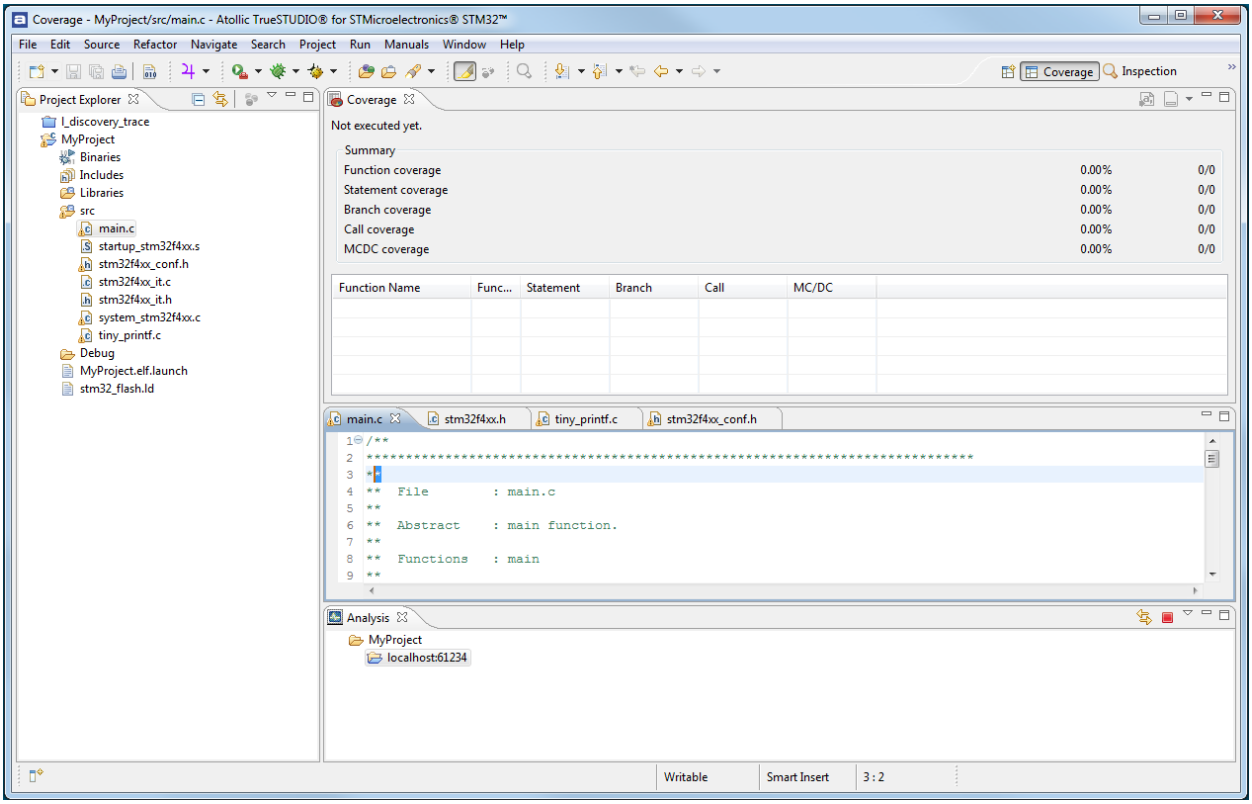


Figure 75 - Code coverage perspective with no analysis results

9. In the **Analysis** view, click on the text **localhost:<port number>** to select the hardware connection you want to control. Notice that the **Refresh** button in the **Analysis** view toolbar becomes enabled:

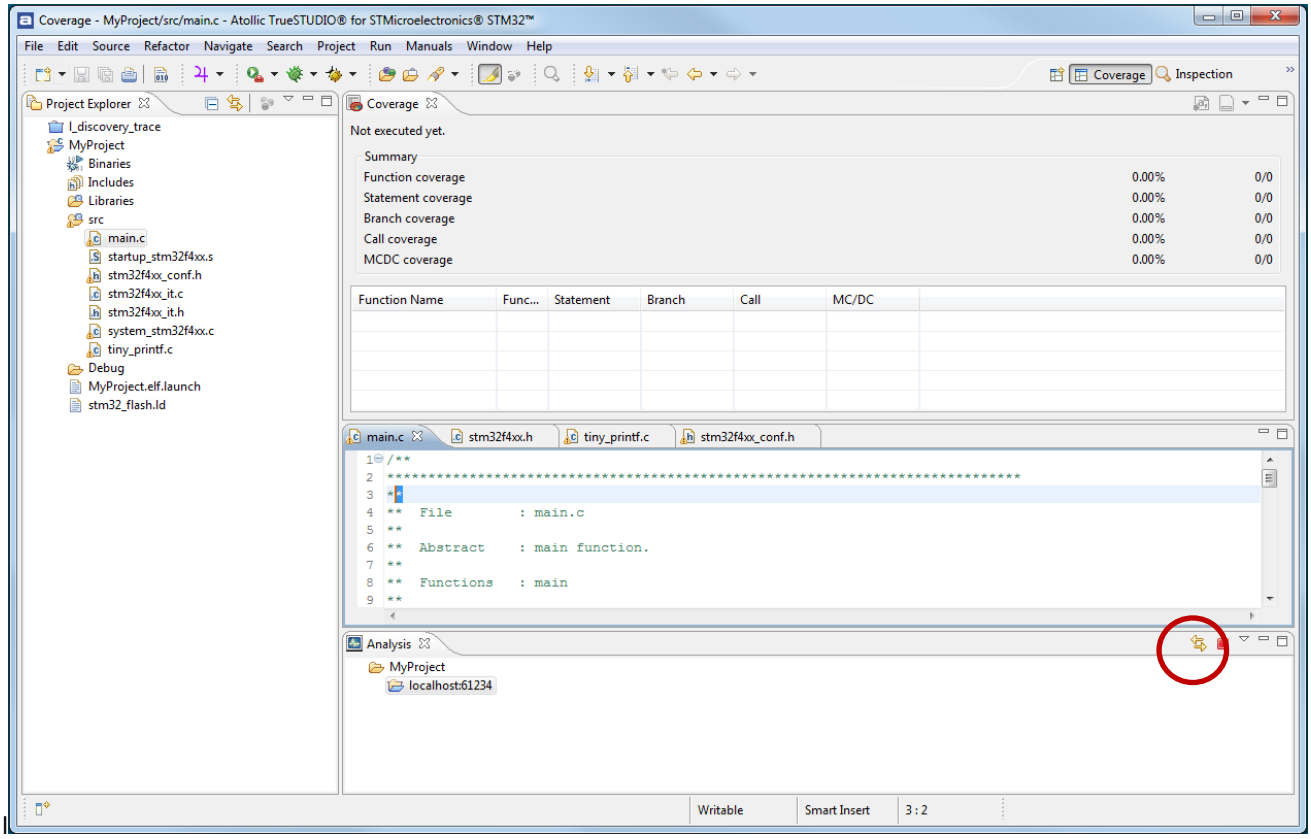


Figure 76 - Hardware connection selected

10. At any time during execution of your application in the target board, click on the **Refresh** button in the **Analysis** view toolbar to update new code coverage analysis data to ***Atollic TrueANALYZER®.*** The **Coverage** view shows an overall summary of the complete application, as well as analysis results per function. By double-clicking on a function name, the corresponding source code file is opened at the right code line, and color coding display which lines have been executed:

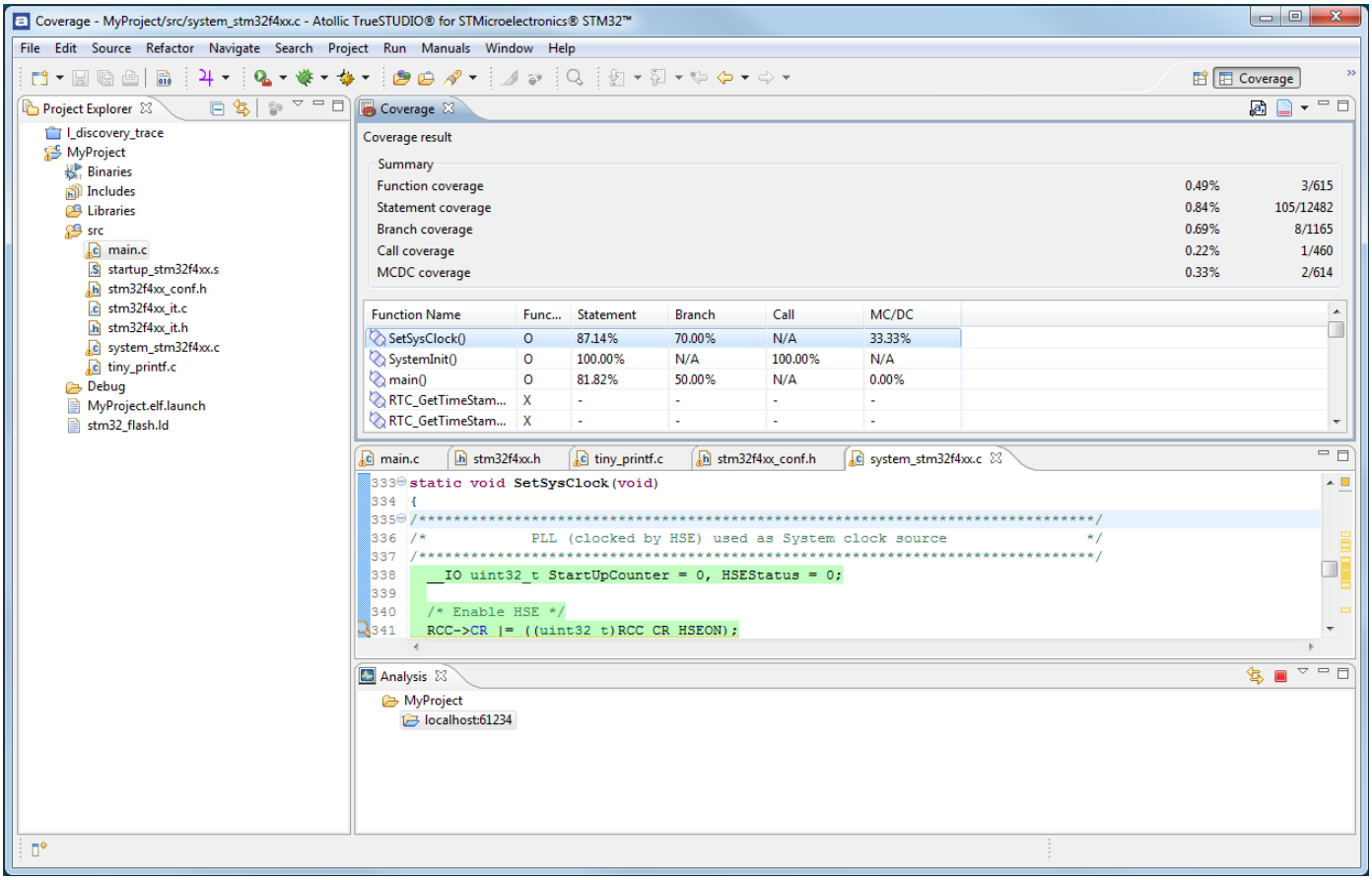


Figure 77 - Visualization of code coverage analysis results

11. You can interact with your embedded system (for example by pressing buttons, sending communication packages, or activating various sensors) as much as you like. Whenever you want to upload the latest analysis results, just click the **Refresh** button again and new test results will be displayed in the IDE.

12. To terminate the code coverage analysis session, make sure the **localhost:<port number>** text is selected in the **Analysis** view. Then click the **Terminate** button in the toolbar:

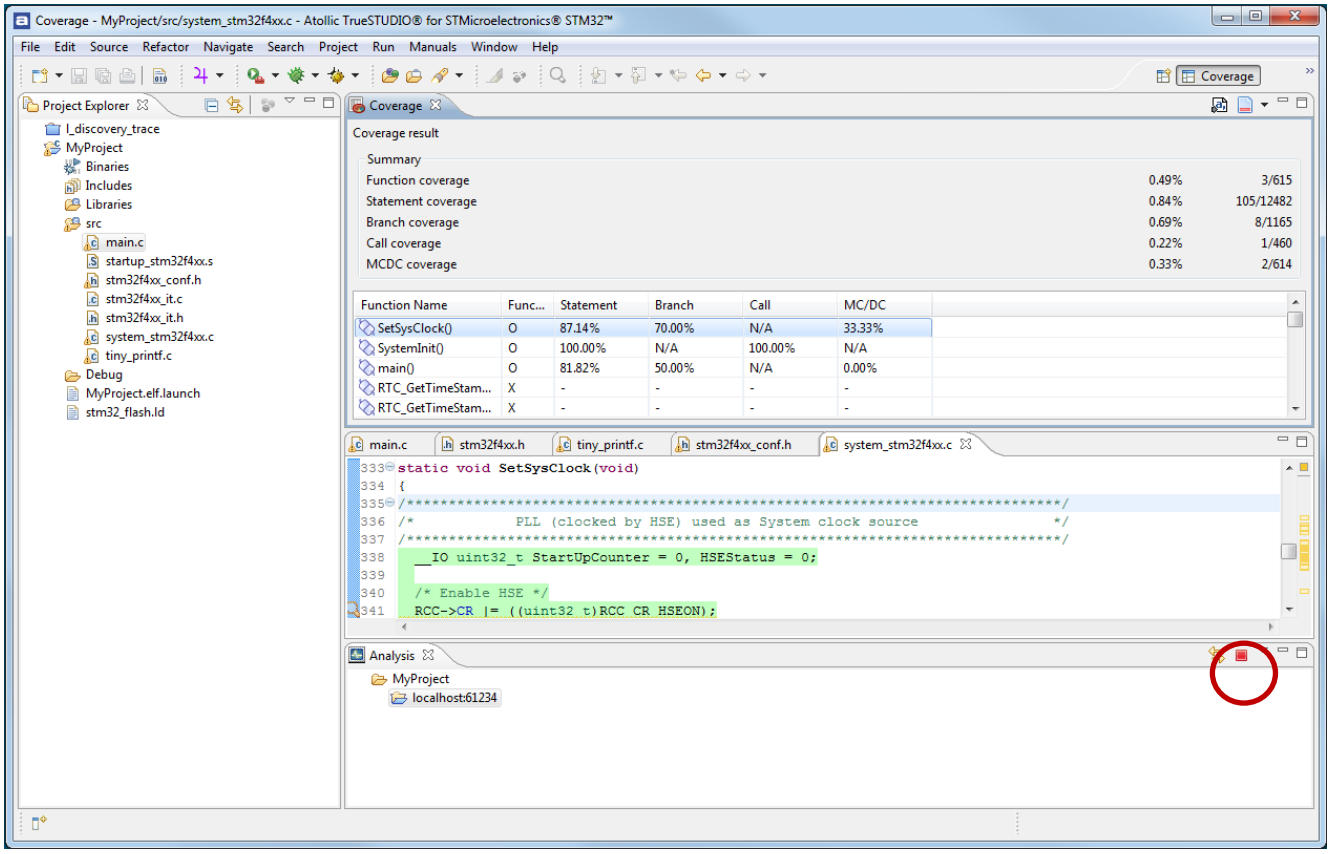


Figure 78 - Terminating an analysis session

Do not forget to terminate the code coverage analysis session, as the connection to the target board remains in use and will block future attempts to launch ***Atollic TrueSTUDIO®*** debugger sessions, ***Atollic TrueANALYZER®*** code coverage analysis sessions, or ***Atollic TrueVERIFIER®*** test sessions.

13. Execution is stopped in the target and ***Atollic TrueANALYZER®*** release the connection to the gdbserver. Now switch back to the C/C++ perspective using the **Window, Close Perspective** menu command.

SECTION 4. TEST AUTOMATION

This section provides information on how to use ***Atollic TrueVERIFIER®***, which is an add-on product that can be purchased separately. ***Atollic TrueVERIFIER®*** is fully integrated into ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™.***

Buyers of ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Professional*** get a one week fully-working license of ***Atollic TrueVERIFIER®***, while ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Lite*** users get a demo version.

This section covers:

- Creating unit tests
- Creating and managing test cases
- Running tests
- Analyzing test results

INTRODUCTION

Atollic TrueSTUDIO® for STMicroelectronics® STM32™ includes ***Atollic TrueVERIFIER®***, a deeply integrated module for professional software test automation. ***Atollic TrueVERIFIER®*** automates testing of your software, and is a very powerful tool for improving the quality of your software.

Atollic TrueVERIFIER® analyses the source code of the project, and generate unit tests automatically (in C source code) for the selected functions in the application. It compiles the unit tests automatically, and downloads them to the target board using the same JTAG probe being used for ***Atollic TrueSTUDIO®*** debugging.

Once downloaded to the target board, the unit tests are executed automatically with full execution-path monitoring. When the tests are completed, test results as well as rigorous test- and code coverage information is uploaded and presented in the IDE.

Test results for each test case can be of 3 types (success, failure, and error in the case the test could not be executed due to some unexpected problem).

As all test cases are executed with full execution-path monitoring, ***Atollic TrueVERIFIER®*** provides a stringently defined value for the quality of the test cases being run (in practice, how many of all the possible combinations of potential execution paths has been executed with the test suite).

Atollic TrueVERIFIER® provides test- and code coverage information for Block coverage, the more advanced Branch coverage, and even Modified condition/Decision coverage (MC/DC) as required by RTCA DO-178B for testing of flight control system software.

A fully working one-week license of ***Atollic TrueVERIFIER®*** is included for customers of ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Professional***, and a demo version is bundled with ***Atollic TrueSTUDIO® for STMicroelectronics® STM32™ Lite.***

THE TEST SCENARIO

For the purpose of this tutorial, we will be testing two C functions running on a STM3210E-EVAL evaluation board from STMicroelectronics®. The board is connected to the PC using an ST-LINK JTAG probe.

The functions to test are inserted into `main.c` and are defined as follows:

```
int MyFunc1( int x )
{
  if( x < -10 || x > 65 )
  {
         return 1;
  }
  else
         return 0;
}

int MyFunc2( char ch )
{
  return ch - 1;
}
```

EXCLUDING FILES & FOLDERS FROM TESTING

Before starting to generate unit tests, you may want to exclude certain files and folders from unit testing – in particular, you may not want to test 3rd party libraries (such as RTOS, device drivers or middleware you have purchased) and you cannot test hardware interrupt handlers.

To exclude files and folders from taking part in testing, select the **File, Properties** menu command. Drill down to **the Testing, Exclusion** panel:

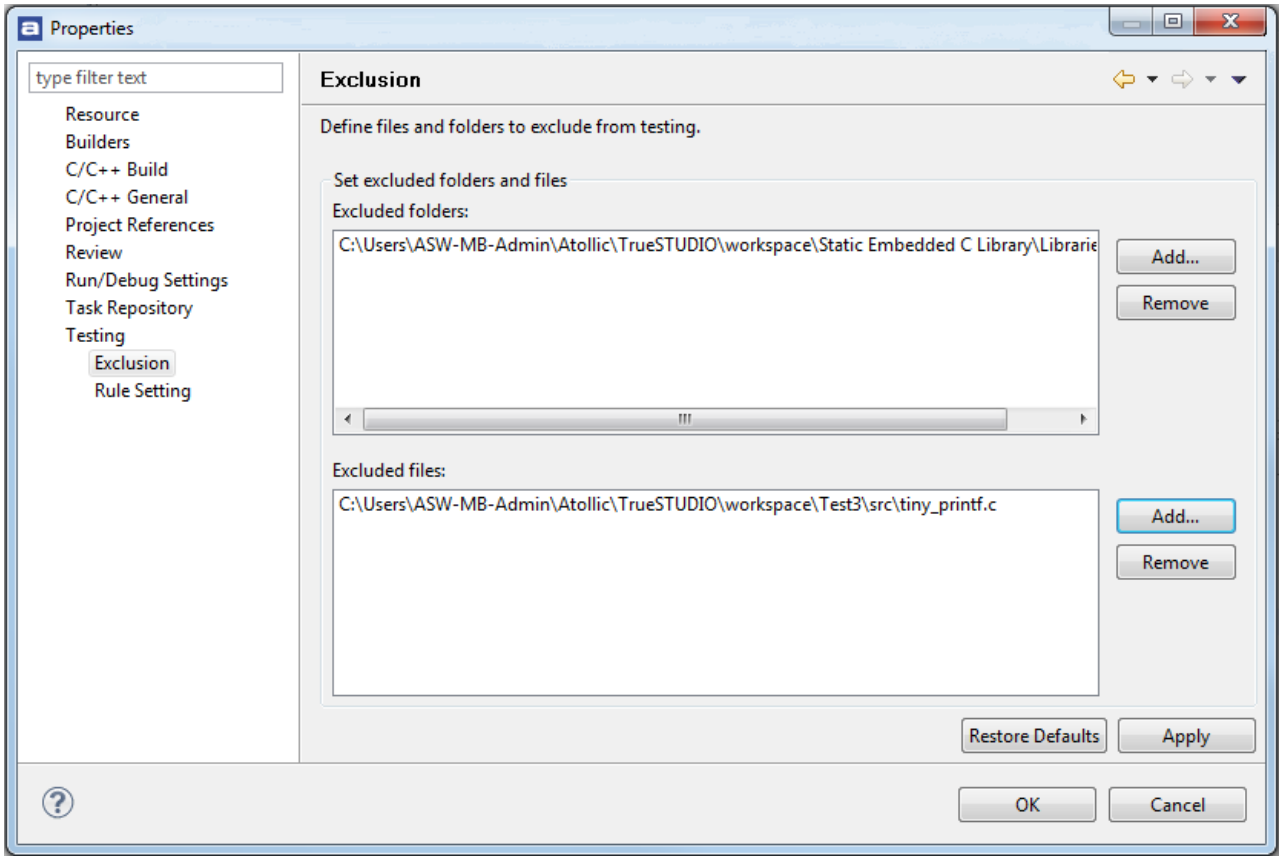


Figure 79 - The file and folder exclusion dialog box

Exclude files and folders as appropriately for your project, and close the dialog box by clicking the **OK** or **Cancel** button.

PREPARING THE PROJECT FOR UNIT TESTING

To enable the project for testing, make sure the project root item is selected in **the Project explorer** view. If it isn't already, click on the Project root name in the **Project Explorer** view:

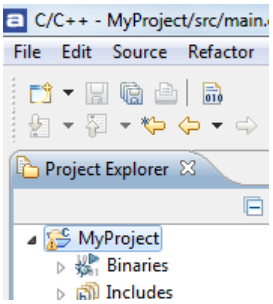


Figure 80 - Selecting the project root item

Then right-click on the project root item to open the context menu, and select the **New, Unit test** menu command.

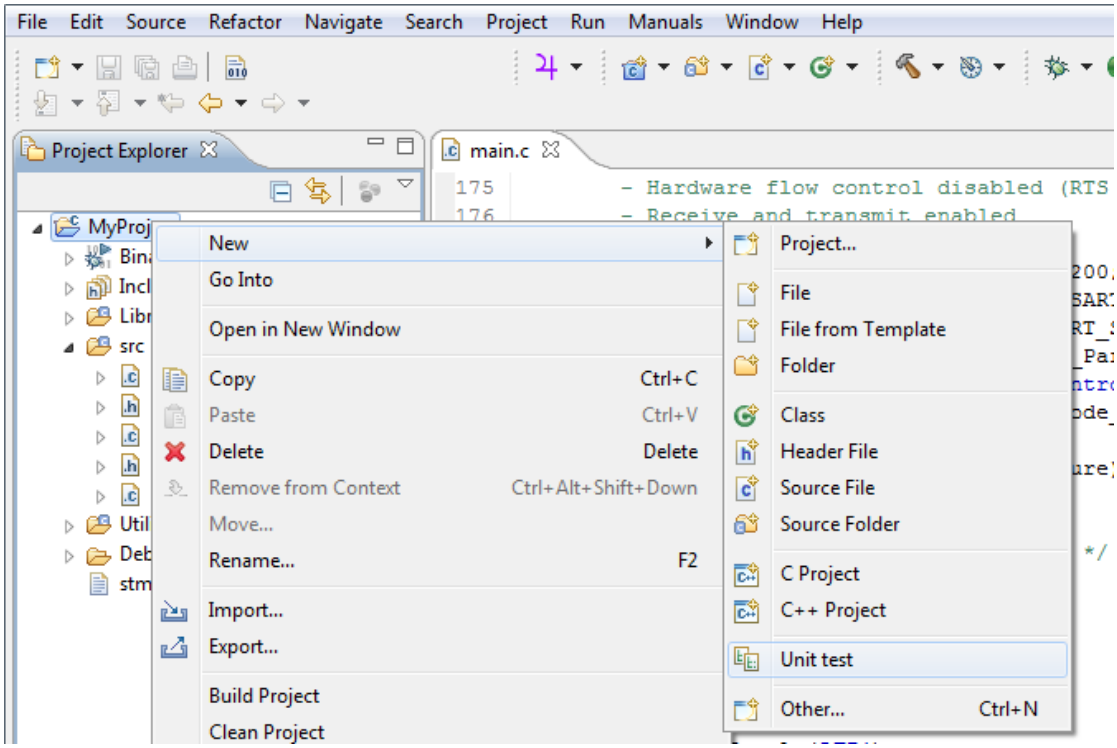


Figure 81 - Preparing a project for unit testing

Atollic TrueVERIFIER® parses the source code files in your project and detects what C functions are available for testing. The currently selected project, along with available functions for testing, is displayed in the dialog box below:

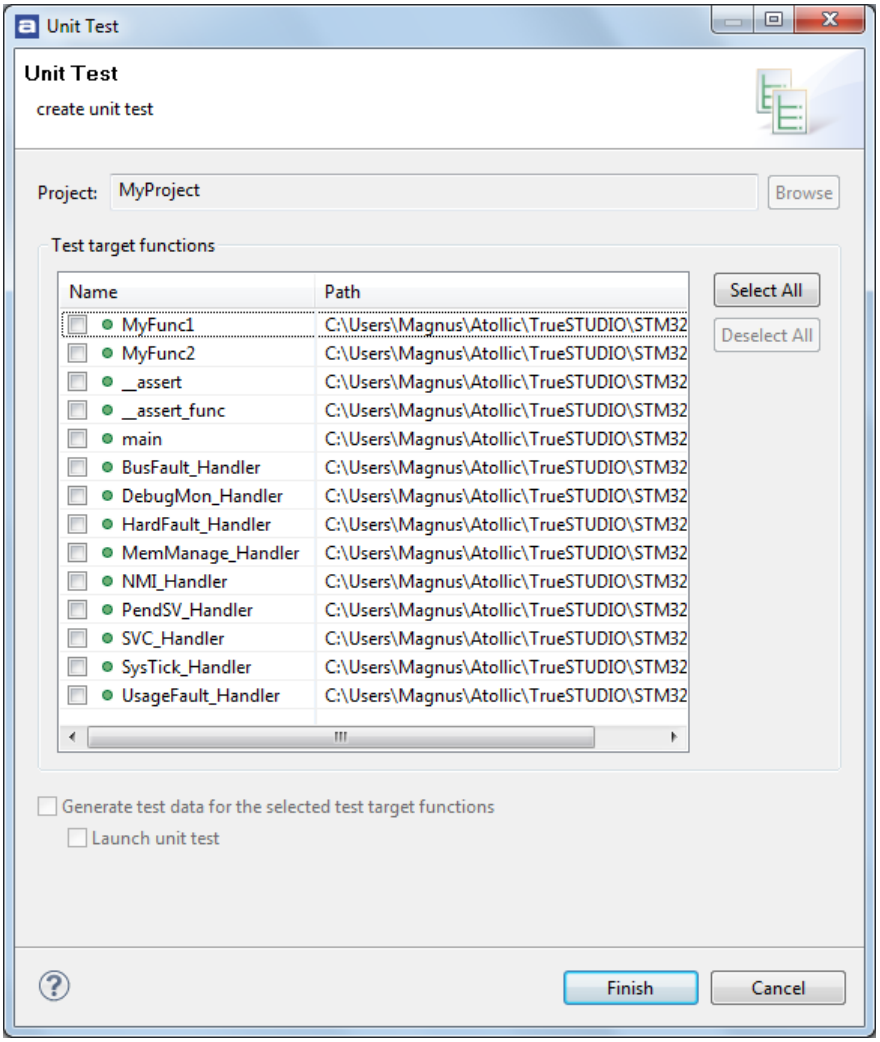


Figure 82 - The function selection dialog box

In this tutorial, we would like to auto-generate unit tests for the MyFunc1() and MyFunc2() functions. We thus select those functions for testing in the function list, and also select the checkbox **Generate test data for the selected test target functions**:

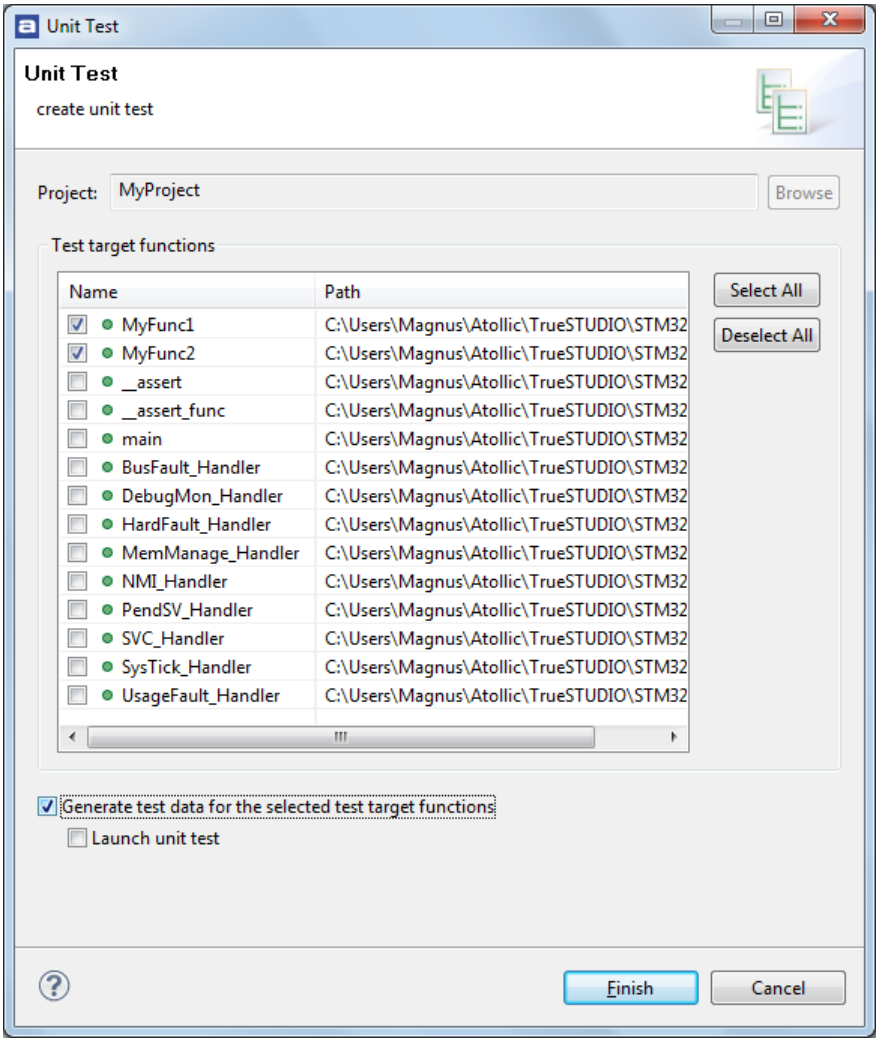


Figure 83 - Selecting functions for testing

These settings will enable auto-generation of:

- Unit test C code for MyFunc1()
- Unit test C code for MyFunc2()
- Test data, i.e. ***Atollic TrueVERIFIER®*** will parse the source code inside the selected functions, and work out what input parameter values will affect the code execution flow. ***Atollic TrueVERIFIER®*** will then auto-generate unit test cases with these important parameter values, ensuring as many combinations of potential execution paths as possible will be tested.

By clicking the **Finish** button, the project is configured for unit testing as defined above, and a new `<ProjectLocation>\unittest` folder is created to hold the unit test source code files.

If ***Atollic TrueVERIFIER®*** asks for permission to switch to the **Unit test** perspective, click the **Yes** button to do so:

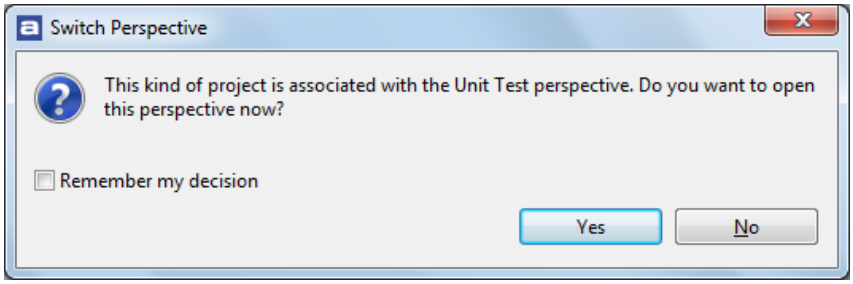


Figure 84 - Perpective swap message box

As can be seen in the screenshot below, ***Atollic TrueVERIFIER®*** has now activated the **Unit test** perspective.

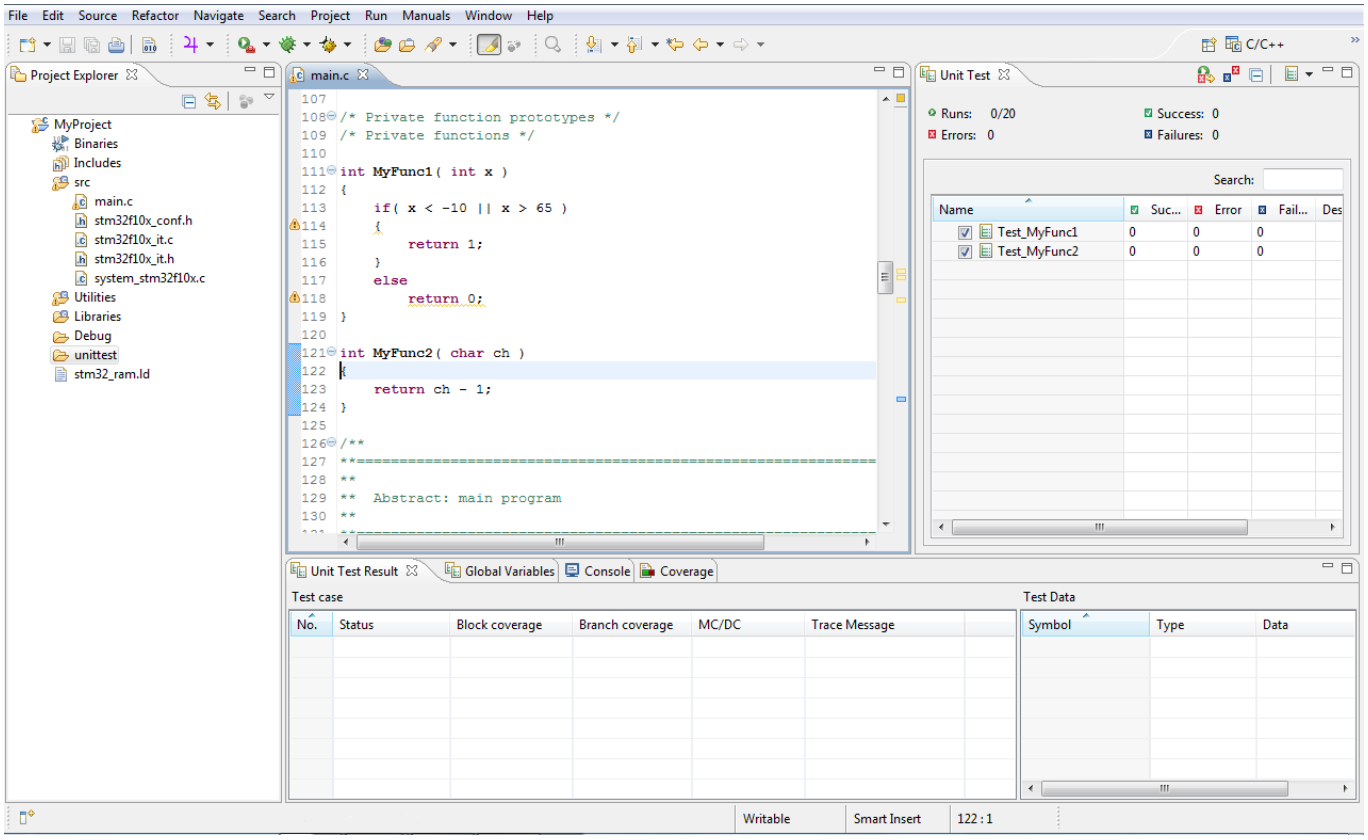


Figure 85 - The unit test perspective

Furthermore, there are four docking views being displayed:

- The **Unit Test** view: This view provides an overview of test results for each of the functions to test, including how many of the test cases for each function have

been run (with success, failure or error causing the test to stop). A summary information pane is displayed as well.

- The **Unit Test Result** view. By selecting one of the functions in the **Unit Test** view, the details of all test cases for that function is displayed in the **Unit test result** view. Each test case is a separate call to the function with different input parameter values. The view show Block-, Branch- and MC/DC test- and code coverage (as required by RTCA DO-178B for flight control system software for example), as well as the actual parameter values being used in that test case.
- The **Global variables** view provides information on global variables.
- The **Coverage** view provides summary information for the complete test.

SETTING UP THE HARDWARE AND RUN THE FIRST TEST

Before the tests can be run automatically, the hardware connection to the target board must be configured. In the **Unit test** perspective, select **the Run, Unit test configuration** menu command.

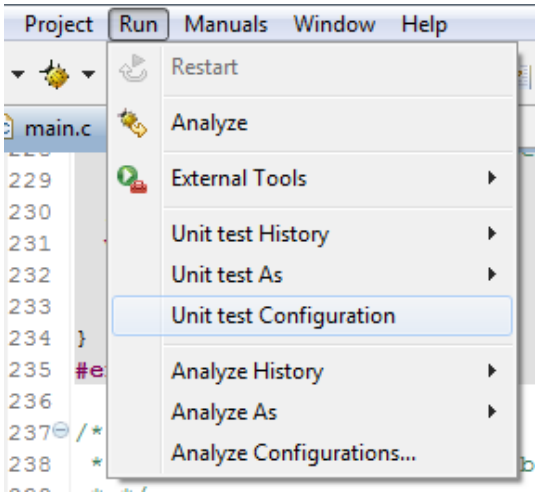


Figure 86 - Configuring the hardware connection

A dialog box with test launch configurations is displayed:

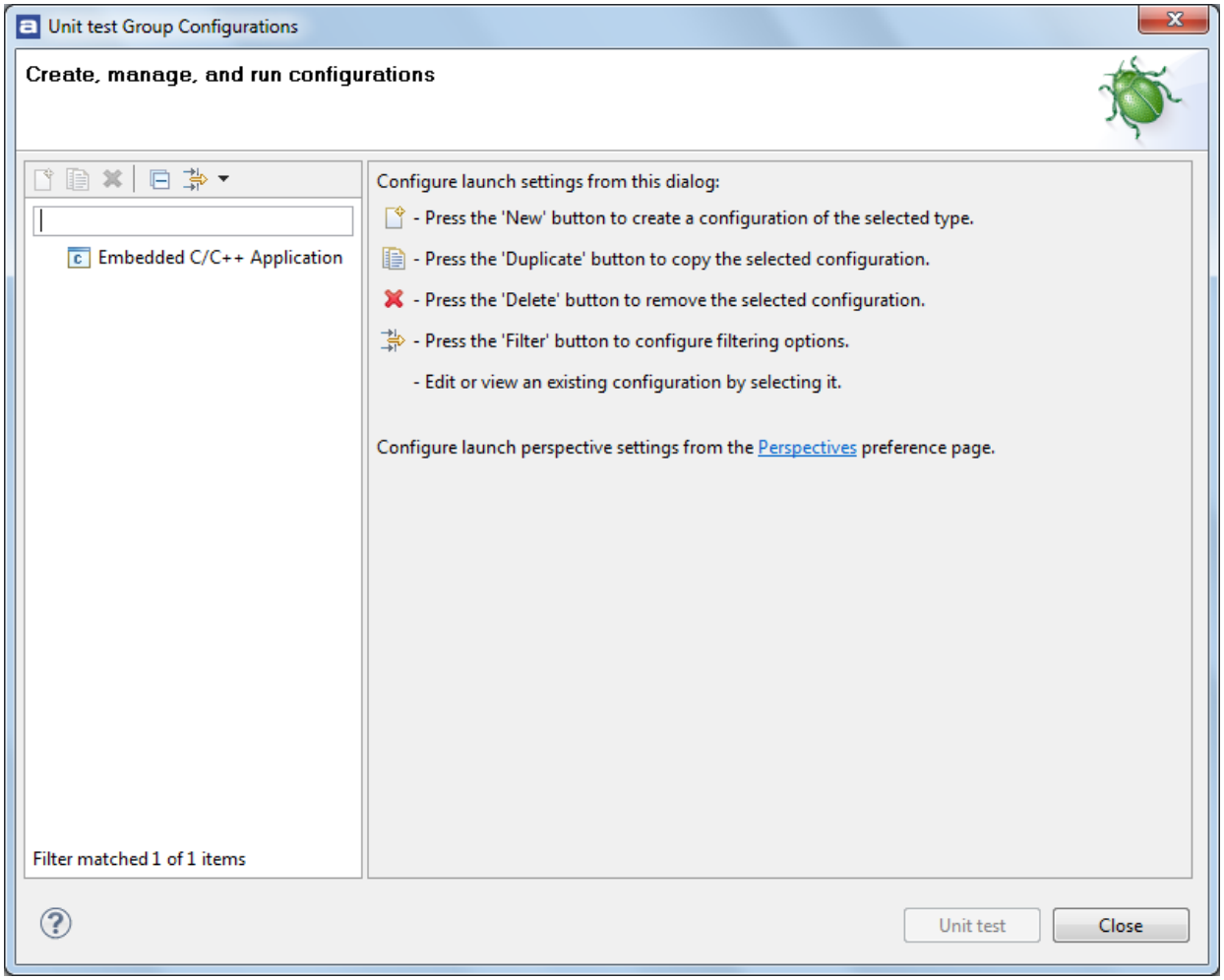


Figure 87 - Unit test launch configurations

As no hardware configurations exist initially, we must create one. Double-click on the **Embedded C/C++ Application** item in the pane to the left. A configuration is then created with default values. Select the newly created launch configuration:

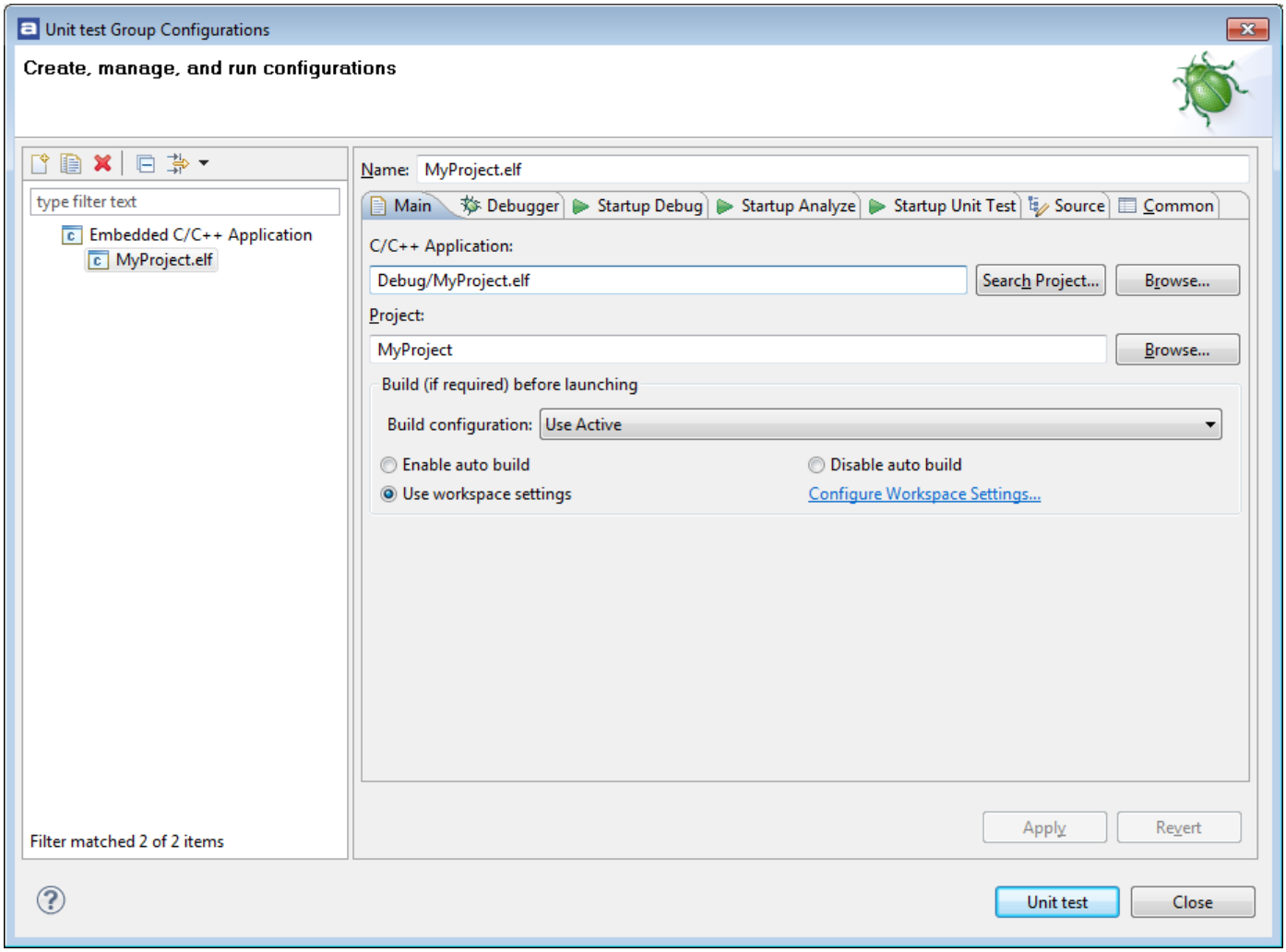


Figure 88 - The unit test launch configuration

Select the **Debugger** tab and ensure the settings of JTAG probe to use are correct (for example, we need to select the ST-LINK JTAG probe):

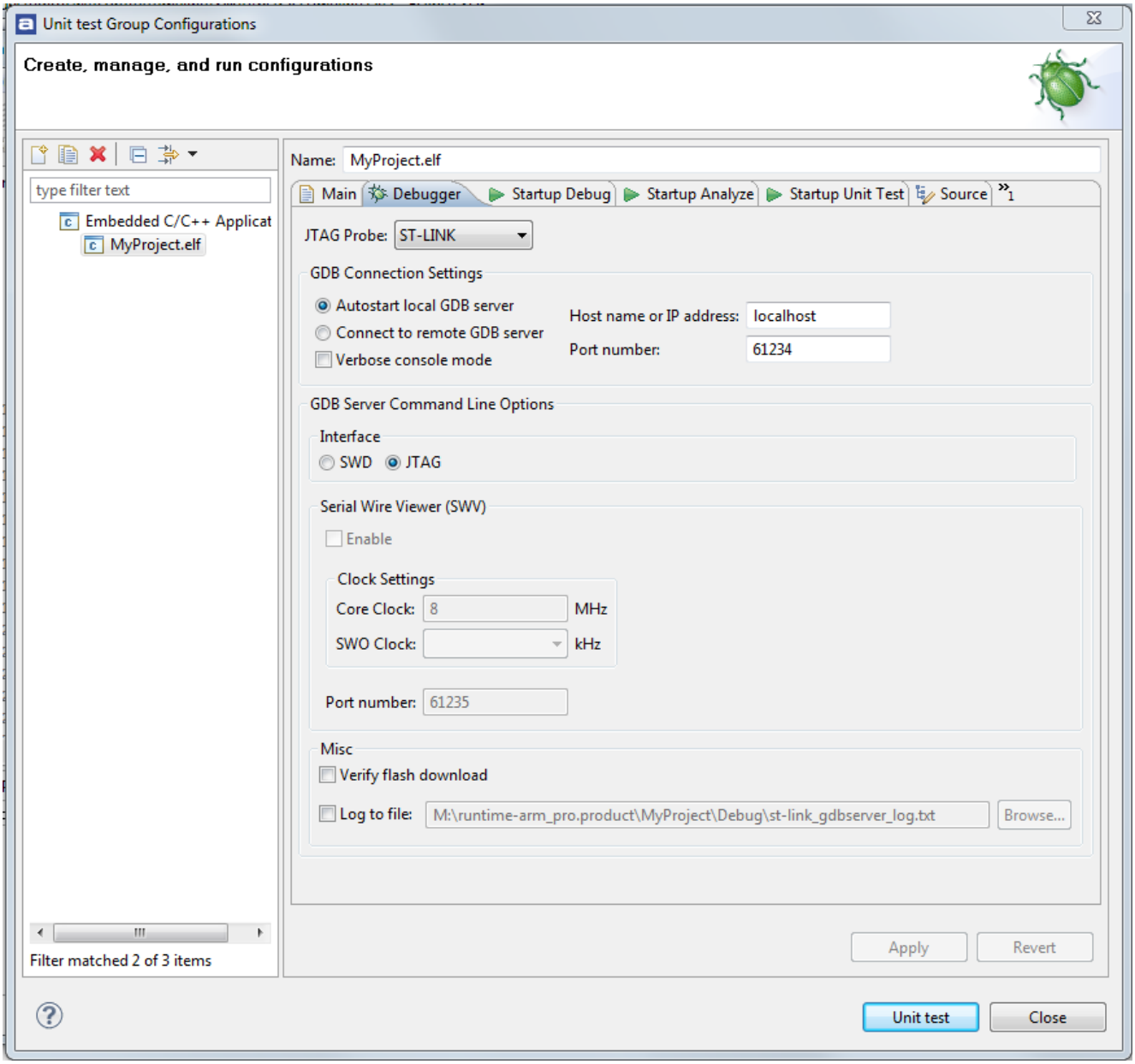


Figure 89 – The debugger JTAG probe configuration

Then switch to the **Startup Unit Test** tab, which provides a possibility to modify the debugger initialization script to use when connecting to the hardware. For this tutorial, make no changes unless required by a different board setup.

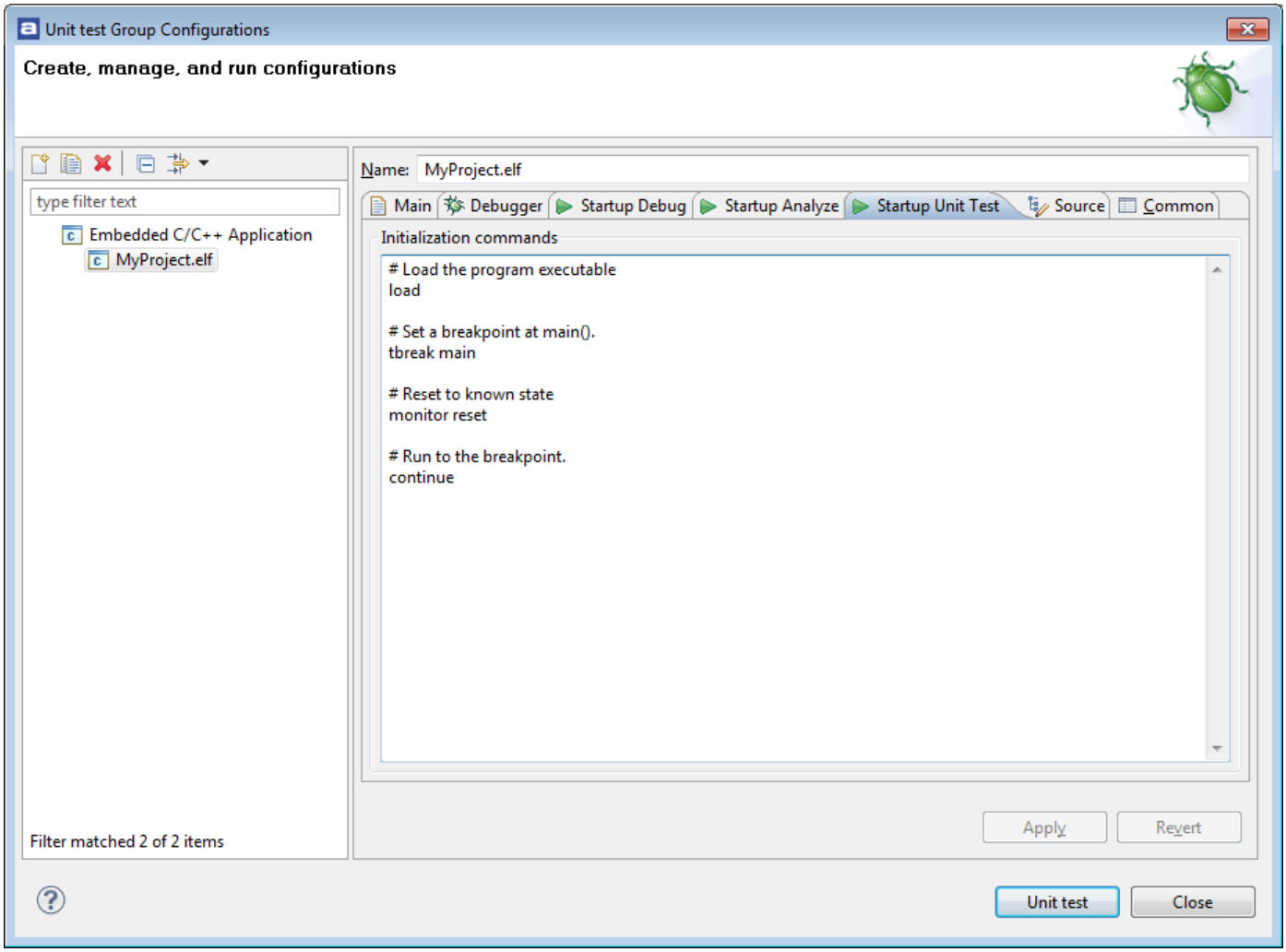


Figure 90 - The unit test debugger startup script tab

Click the **Unit test** button to start the unit test. Source code will now be analyzed; unit test source code files are generated and compiled, and finally downloaded and executed in the target board.

Once tests are completed, the unit test source code files are located in the `unittest` folder; and the docking views in the **Unit test** perspective provides detailed information on the test results.

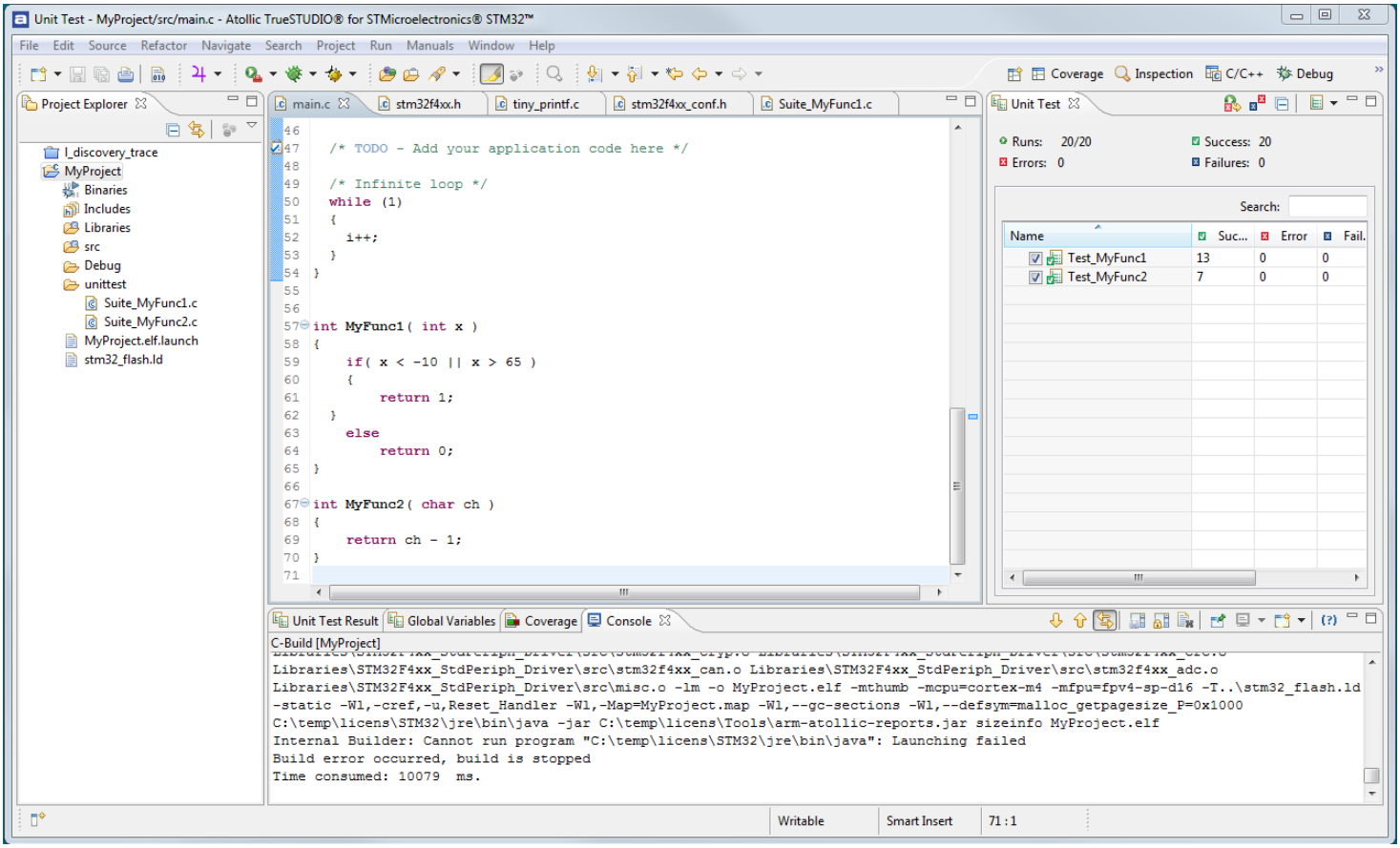


Figure 91 - The unit test perspective after a test session

ANALYZING TEST RESULTS

Once a test session has been completed, the test results can be analyzed in detail. For example, the **Unit Test** view display the overall status of the test session (number of tests that has been run, the number of successful and failed tests, as well as tests that could not be executed due to some error). It also provides a list of functions being tested, along with success/failure/error statistics for each function.

Initially, the **Unit Test Result** view is empty, as no function test is selected in the **Unit Test** view.

In the **Unit Test** view, select the test labeled **Test_MyFunc1**. The **Unit Test Result** view will now be filled with detailed information on each of the unit test cases that tested the `MyFunc1()` function:

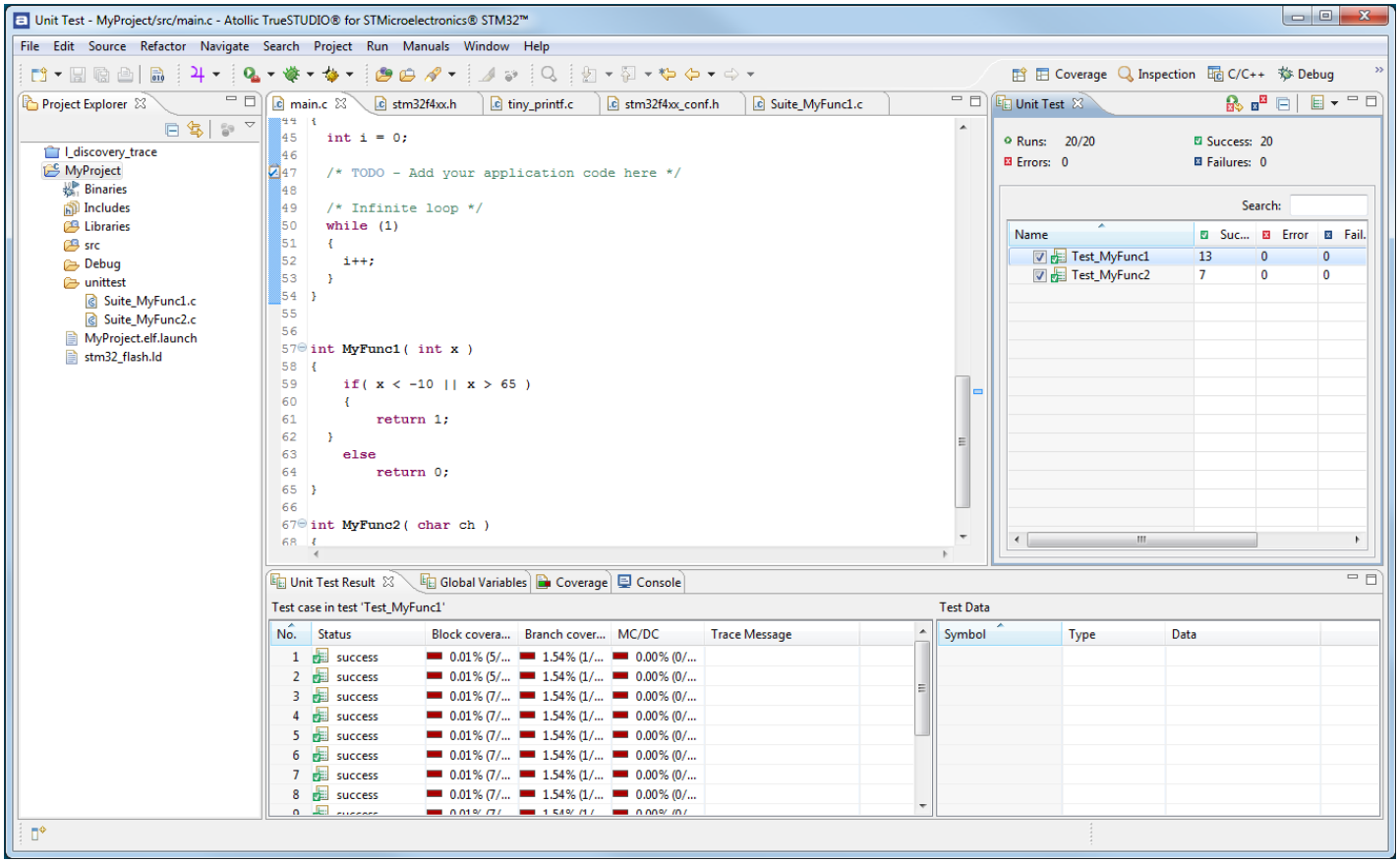


Figure 92 - The unit test result view

The function `MyFunct1()` was tested 13 times (i.e. it was called 13 times with different combinations of input parameter values). The **Unit Test Result** view lists the details for each of those tests, including result (success/failure or test error), as well as the code coverage achieved in that test run.

Atollic TrueVERIFIER® runts the test with execution path monitoring, i.e. it detects the exact execution flow during testing. Block coverage provides a fairly simple level of code coverage, as it only tells you to what extent all code lines/code blocks have been executed. The Branch coverage and MC/DC coverage is much more complex, and performs testing on the same level as is required for the most safety critical of flight-control-system software.

By selecting one of the test cases in the **Unit Test Result** view, the exact parameter values used for that particular test is displayed in the **Test Data** pane as well:

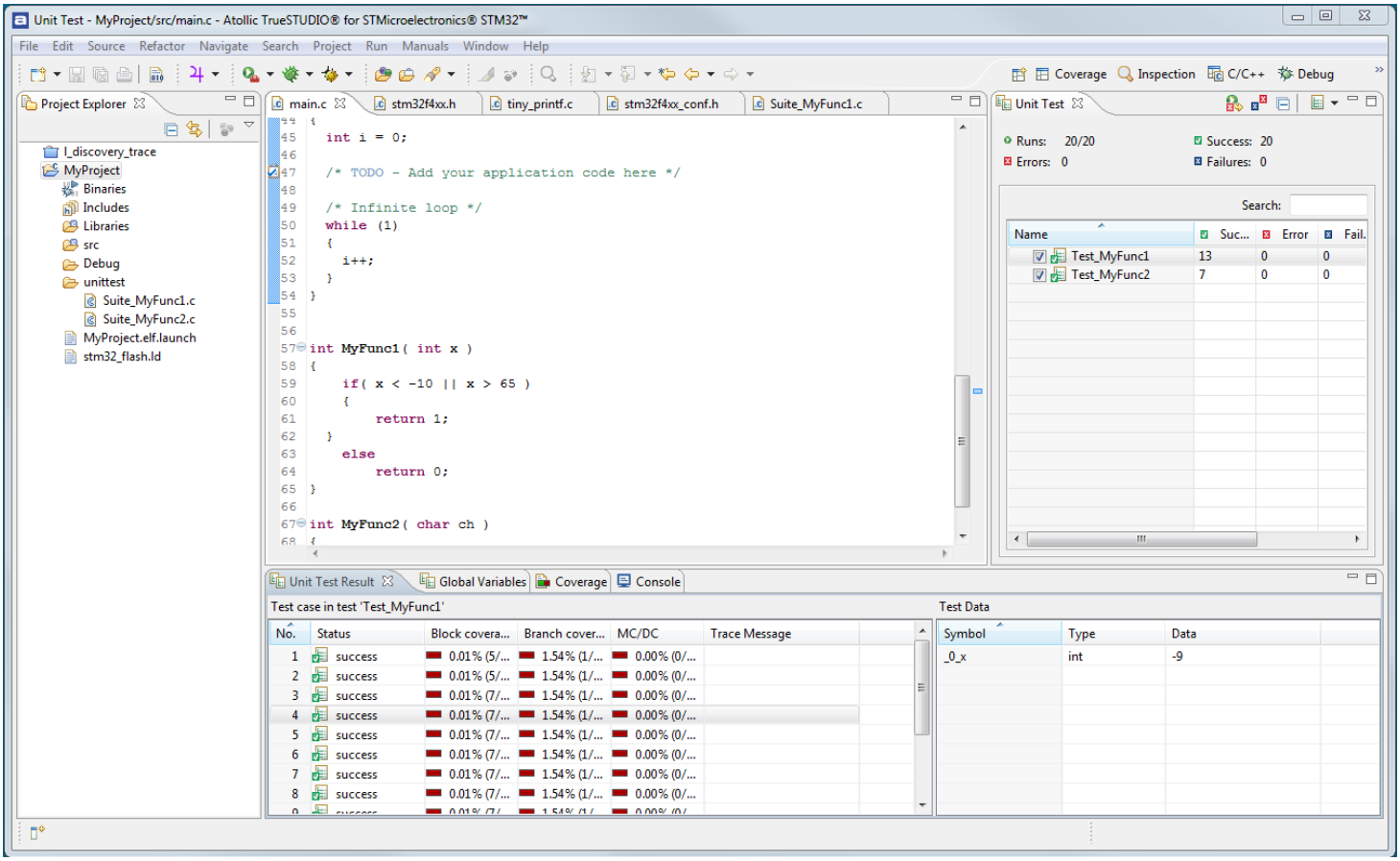


Figure 93 - The test data pane

CONTROLLING TEST CASES MANUALLY

Atollic TrueVERIFIER® analyses the source code of the application to test, and auto-generate a number of test cases with function calls with a suitable set of input parameter values. These are selected by parsing the source code of the functions.

Let's study the `MyFunc1()` function again:

```
int MyFunc1( int x )
{
  if( x < -10 || x > 65 )
  {
        return 1;
   }
  else
        return 0;
}
```

For this function, there is only one function parameter but two values that affects the execution flow; x is less than -10 or x is greater than 65. For complete testing though, it is useful to test some other values, like MAXINT, MININT, 0, and one or two values next to all these special values.

So, for the above function, ***Atollic TrueVERIFIER®*** will generate unit tests with input parameters for x as follows: -2147483648, -11, -10, -9, -2, -1, 0, 1, 2, 64, 65, 66 and 2147483647.

Or in C programming terms, unit tests with the following function calls have been auto-generated:

```
MyFunc1(-2147483648);
MyFunc1(-11);
MyFunc1(-10);
MyFunc1(-9);
MyFunc1(-2);
MyFunc1(-1);
MyFunc1(0);
MyFunc1(1);
MyFunc1(2);
MyFunc1(64);
MyFunc1(65);
MyFunc1(66);
MyFunc1(2147483647);
```

If you want to add more test cases for a function, and define the input parameter values manually, right-click on the desired function in the **Unit Test** view, and select the **Edit Parameter Values…** menu command.

In this case, right-click on **Test_MyFunc1** and select the menu command **Edit Parameter Values…**

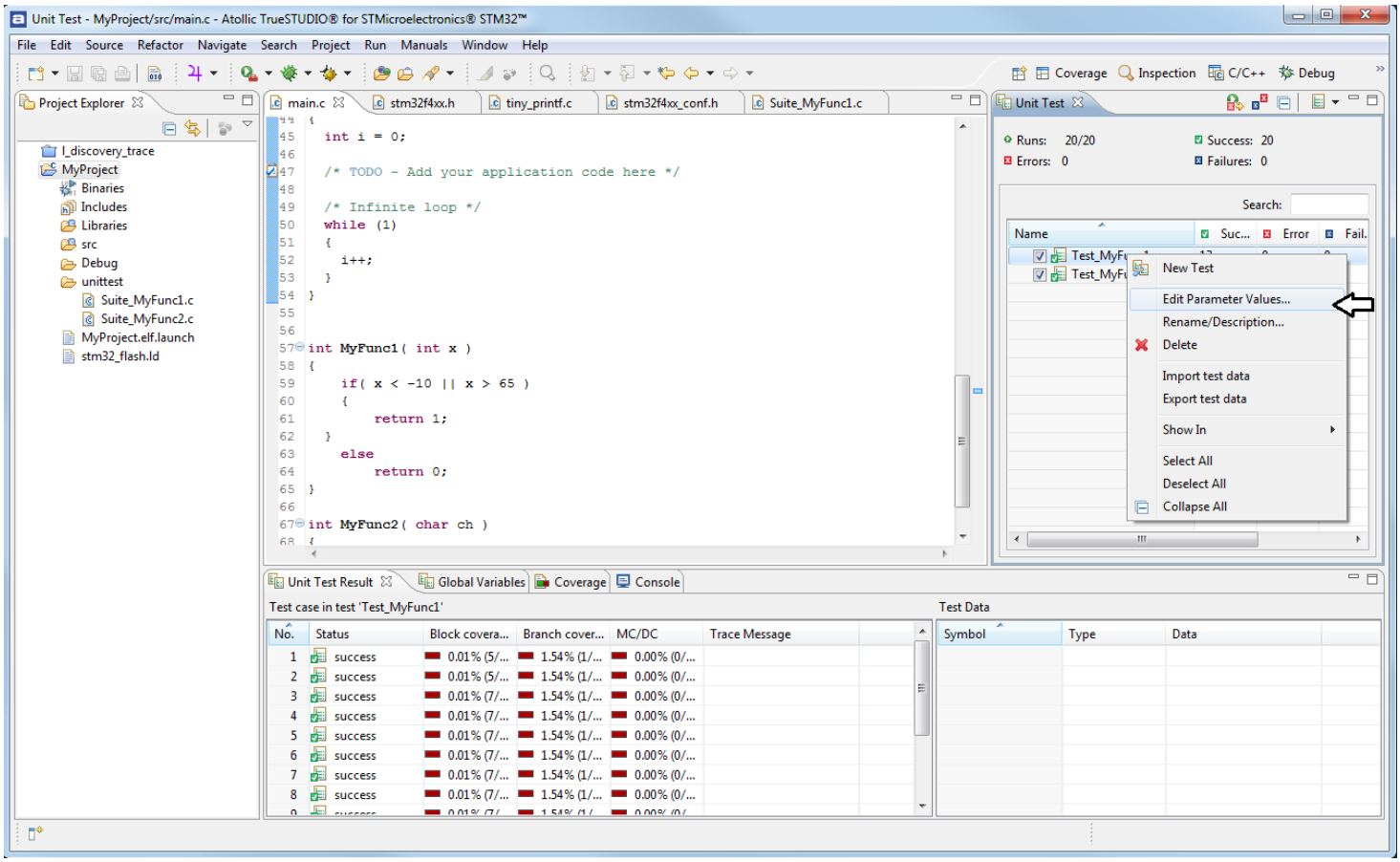


Figure 94 - Editing parameter values

A dialog box with the test data for each test case is displayed:

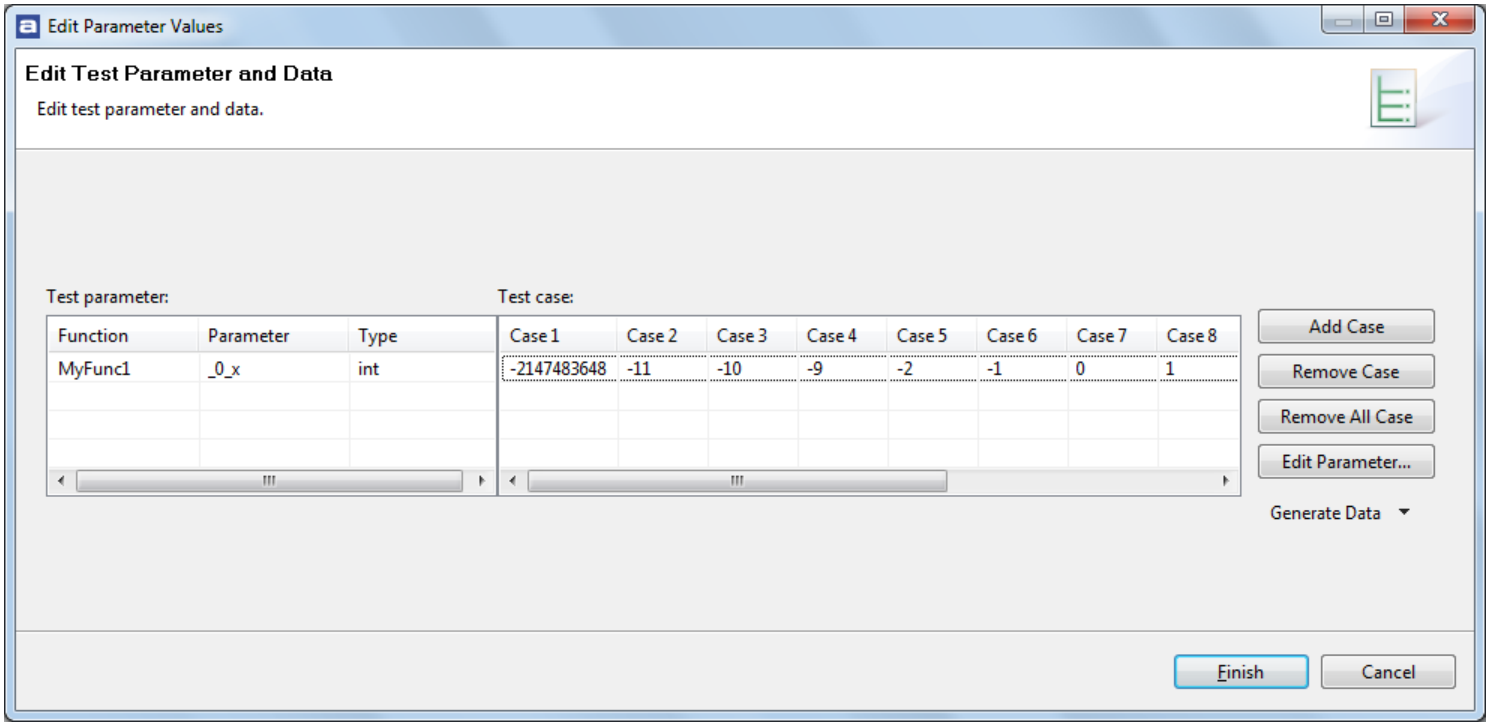


Figure 95 - The parameter editing dialog box

Double-click on the column header (such as **Case 5**) to disable or later re-enable it, and double-click in any cell to edit its numeric value:

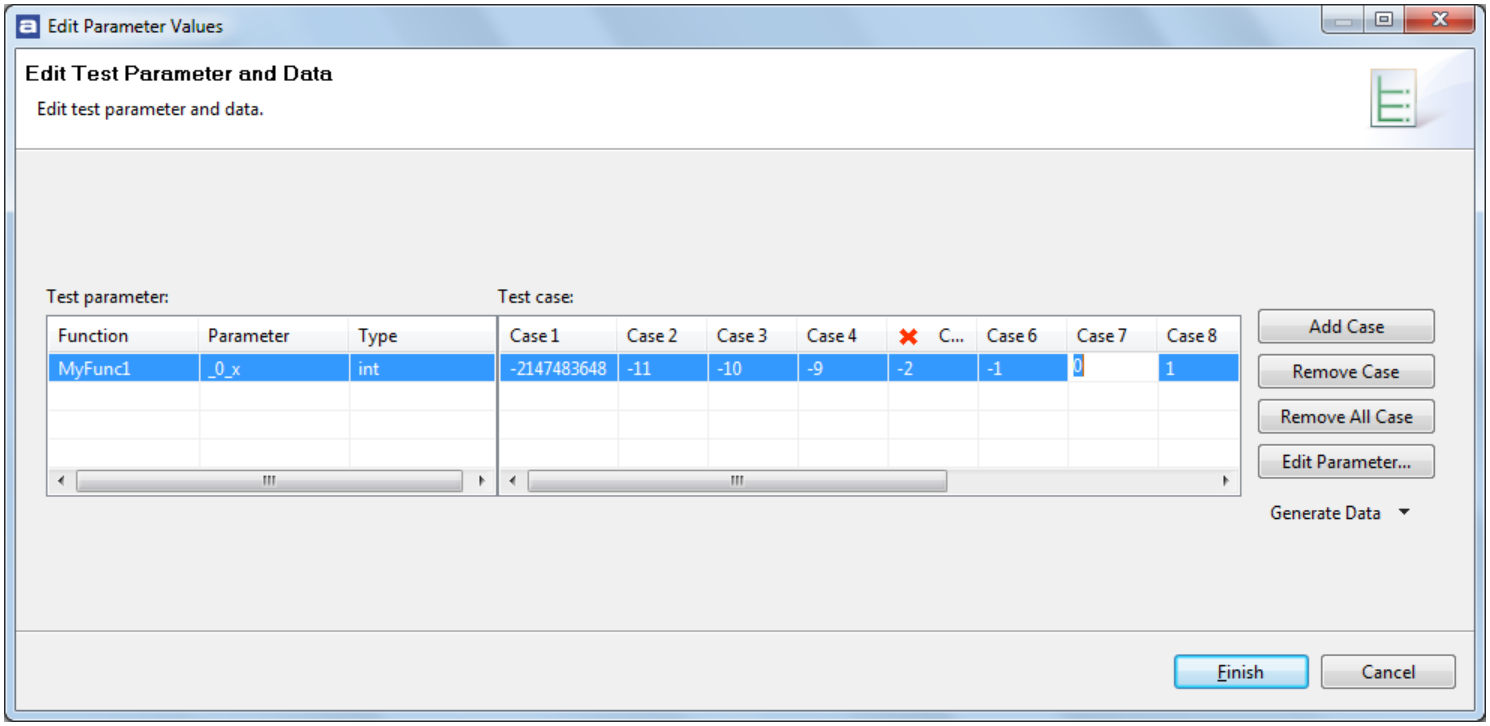


Figure 96 - Edit parameter values

Atollic TrueVERIFIER® has auto-generated 13 test cases for this function. Click on the **Add Case** button to add one more. A **Case 14** has now be created (use the scrollbar to bring it into focus), and double-click in its cells to edit the input parameter values for the new test case.

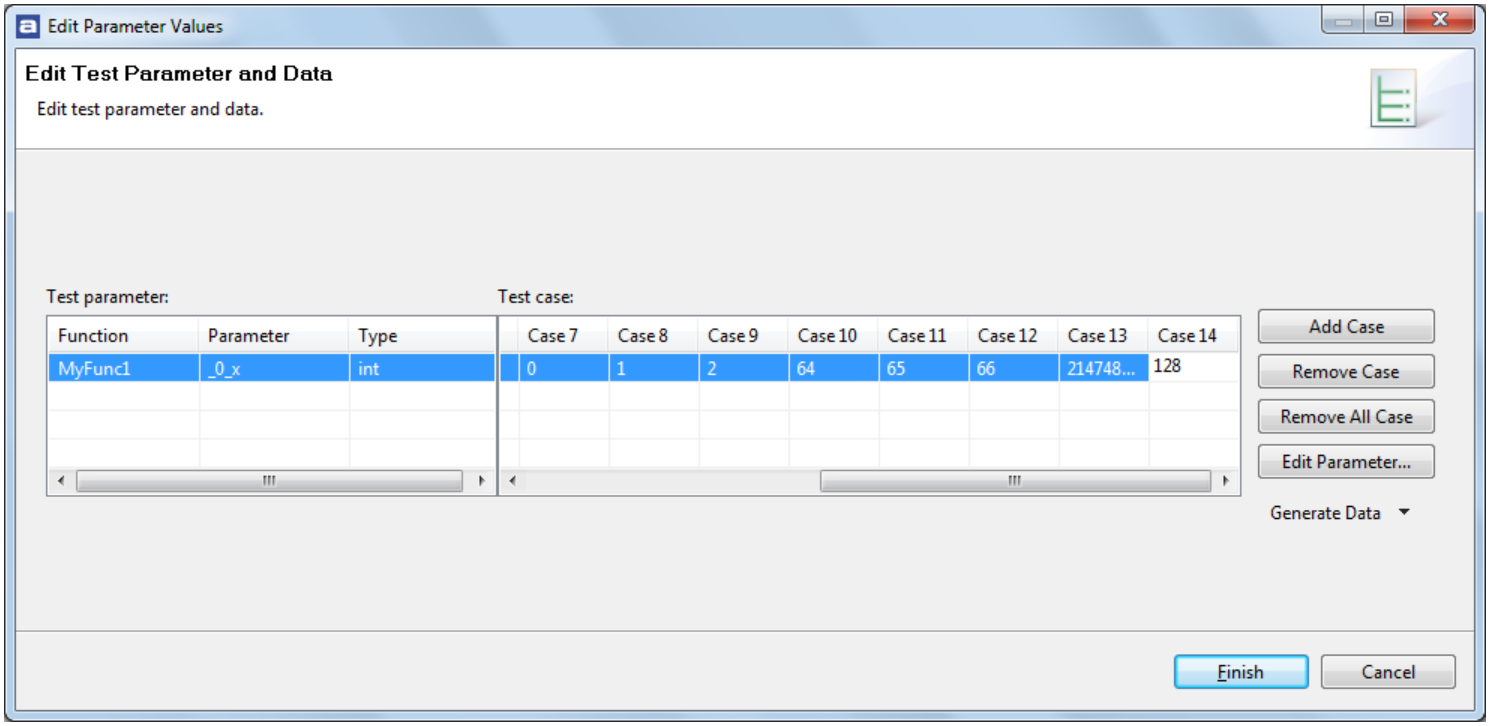
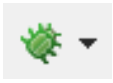


Figure 97 - Edit the new parameter value

Click the **Finish** button to exit the dialog box. Click on the **Run Unit Test** button in the toolbar to start a new test run with the larger set of test cases:



After the new test session has completed, the result views are updated as expected.

CHECKING RETURN CODES AND SPECIAL VARIABLE VALUES

Atollic TrueVERIFIER® generates the appropriate function calls automatically, and provides highly advanced code execution monitoring (up to MC/DC-level) during test execution.

But for unit tests to be fully useful, it is important to know if the return codes of functions are correct for certain input parameter values. It might also be important to check that some (combination of) variables have appropriate values before and/or after the function call.

To enable return code checking, right-click on the desired function in the **Unit Test** view, and open **the Edit Parameter Values…** dialog box.

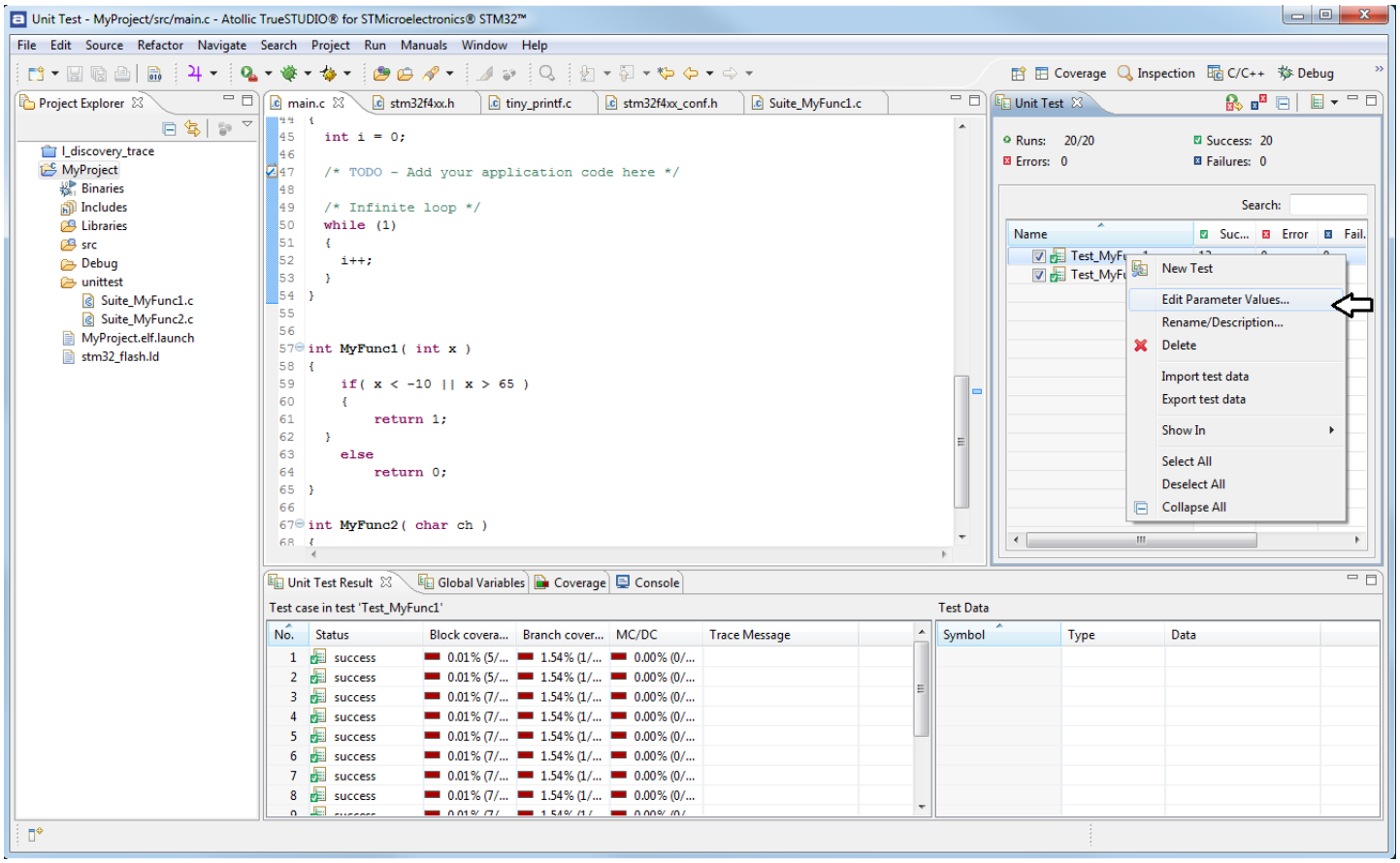


Figure 98 - Prepare for return code checking

The Edit Parameter Values dialog box will then be opened:

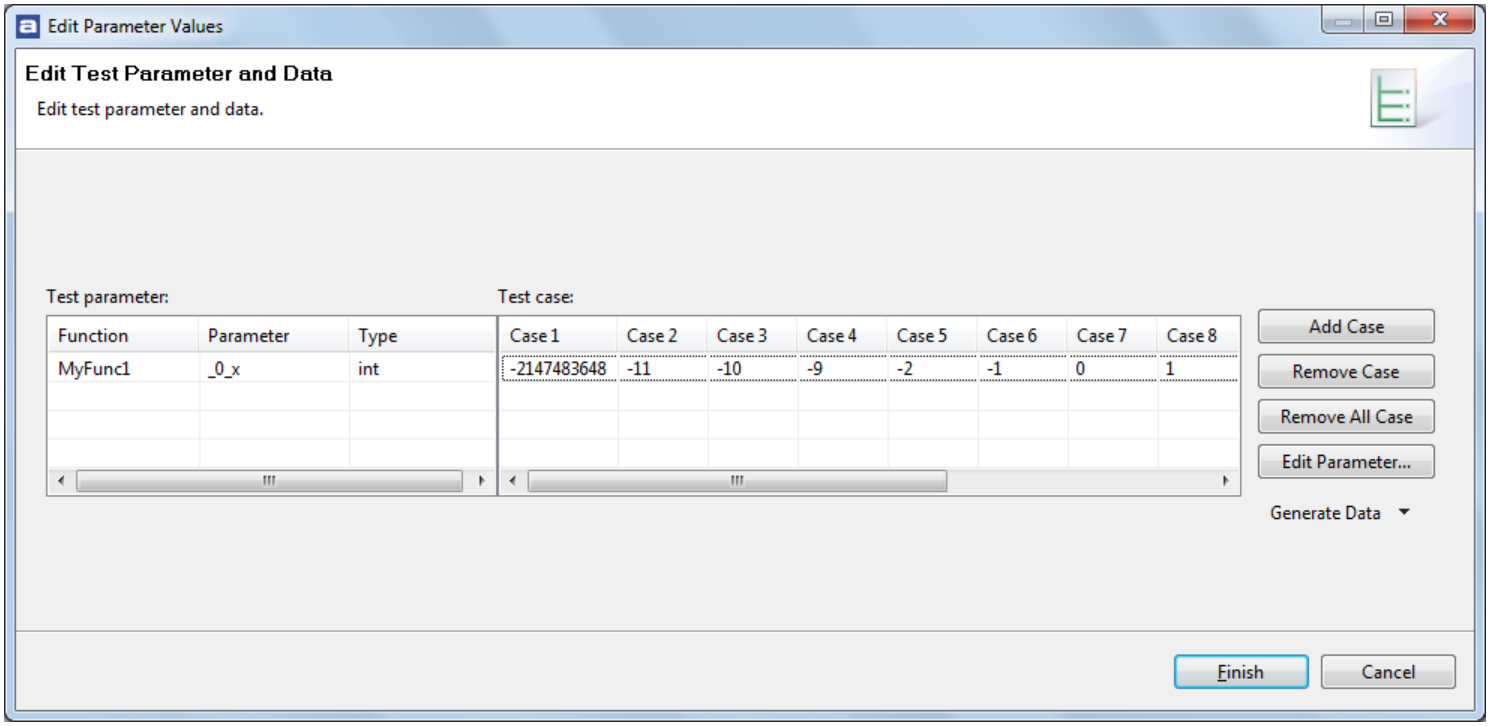


Figure 99 - The edit parameter value dialog box

Click on **the Edit Parameter…** button, and select the **User define parameter** tab:

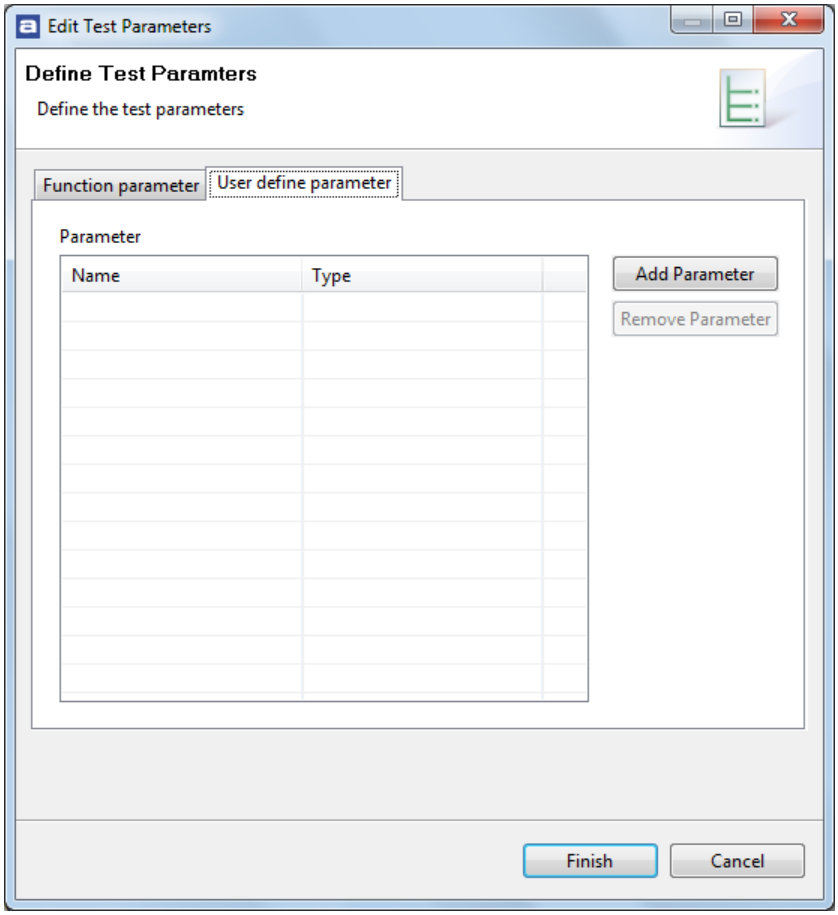


Figure 100 - The user defined parameter dialog box

Click on the **Add Parameter** button, and double-click on the parameter name and rename it to `returncode`. Then change the parameter type to integer:

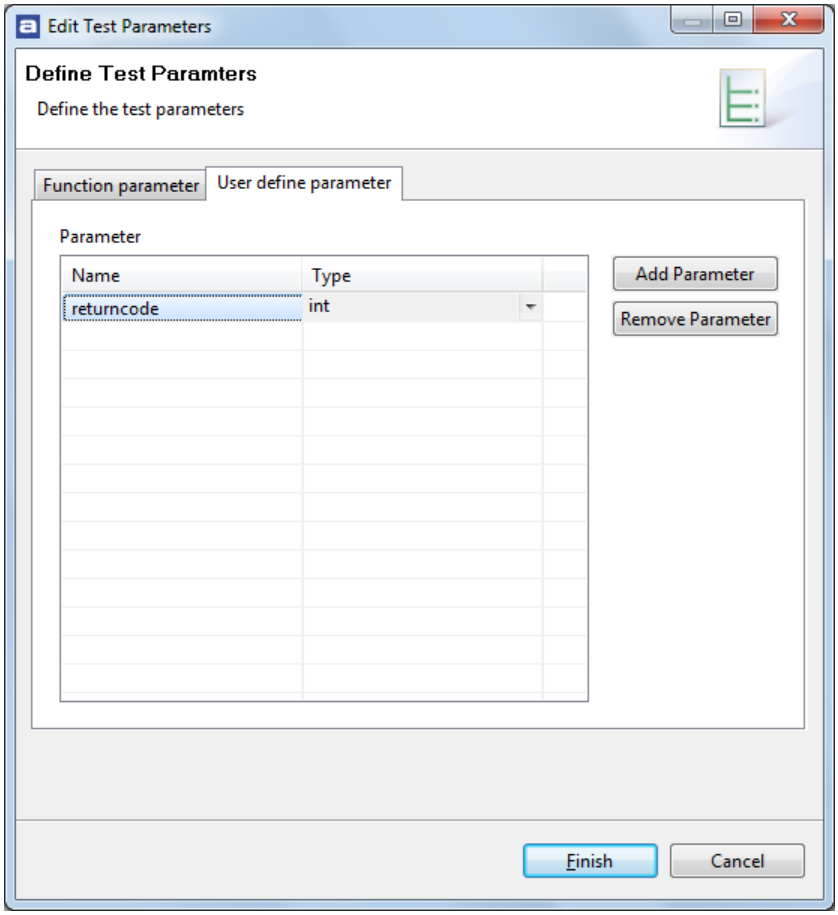


Figure 101 - Editing user defined parameters

Click the **Finish** button to go back to the data definition dialog box. Because the number of parameters is now different, all test data are removed and the dialog box is empty:

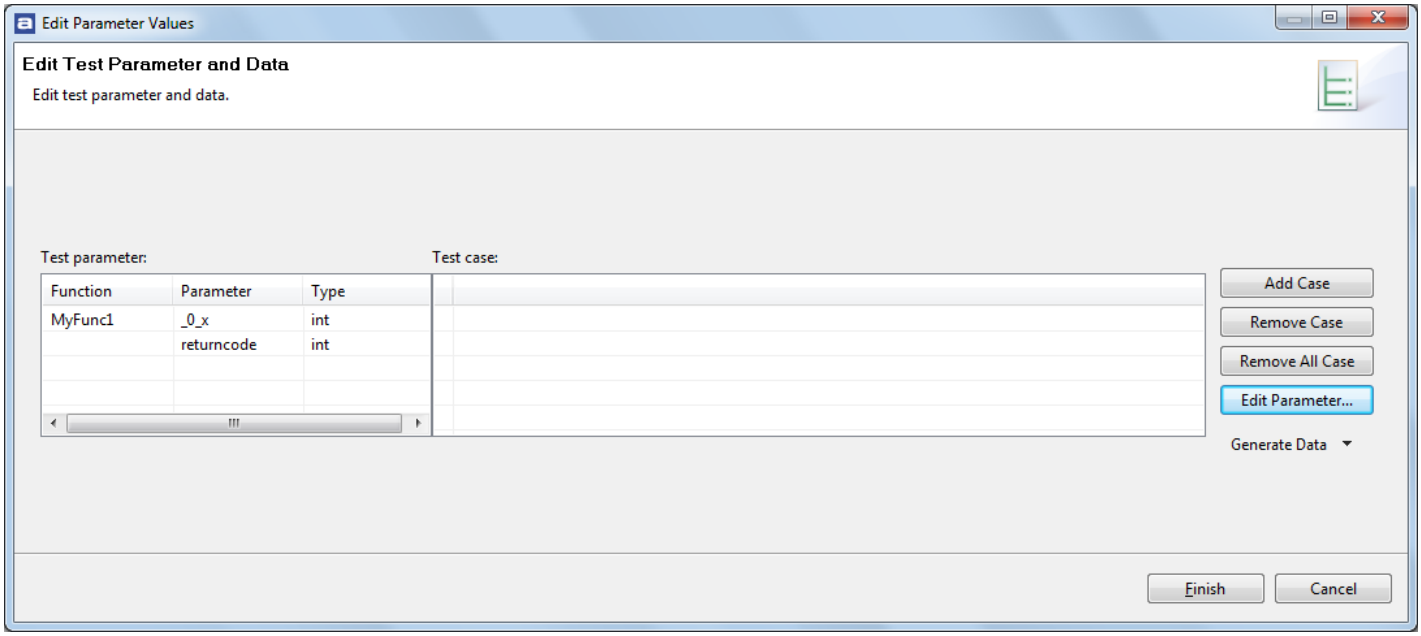


Figure 102 - The parameter editing dialog box

Click on the **Generate Data** button to re-initialize the test suite with test cases and auto-calculated parameter value data:

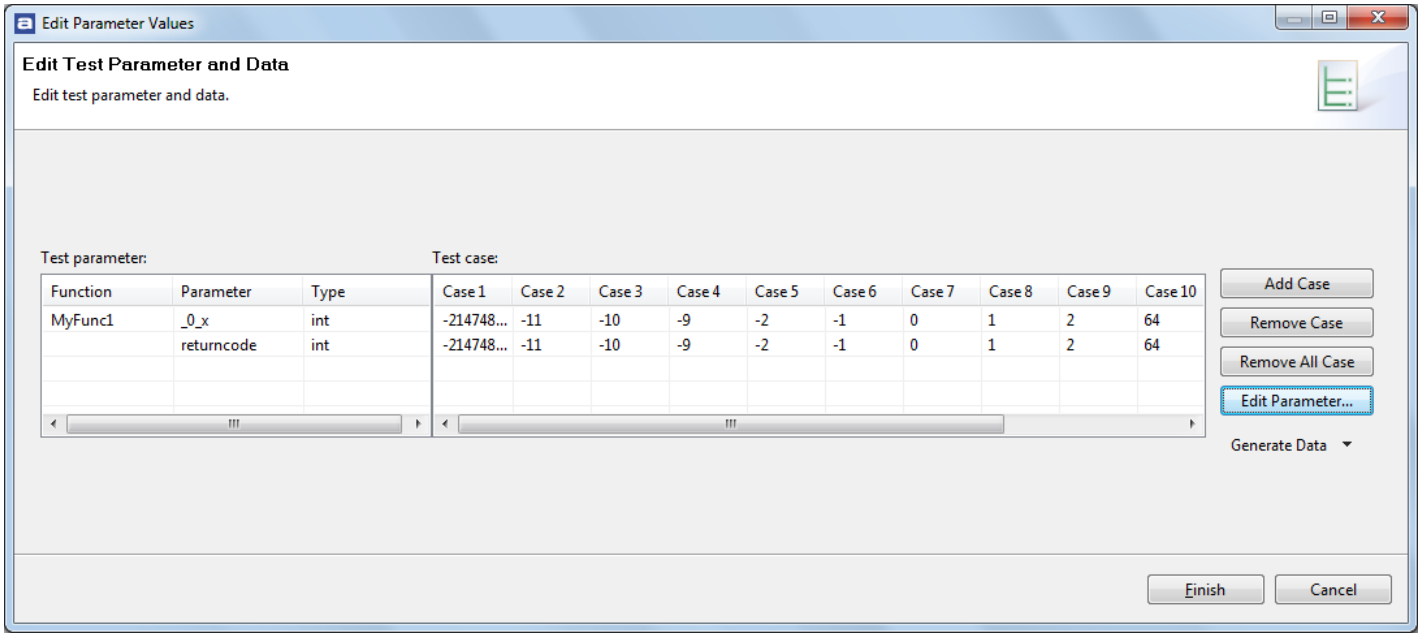


Figure 103 - The parameter editing dialog box

As can be seen from the function implementation below, the function returns 0 for input parameters -10 .. 65, and 1 otherwise:

```
int MyFunc1( int x )
{
  if( x < -10 || x > 65 )
  {
        return 1;
  }
  else
        return 0;
}
```

We must therefore change the expected return code for the various input parameters accordingly:

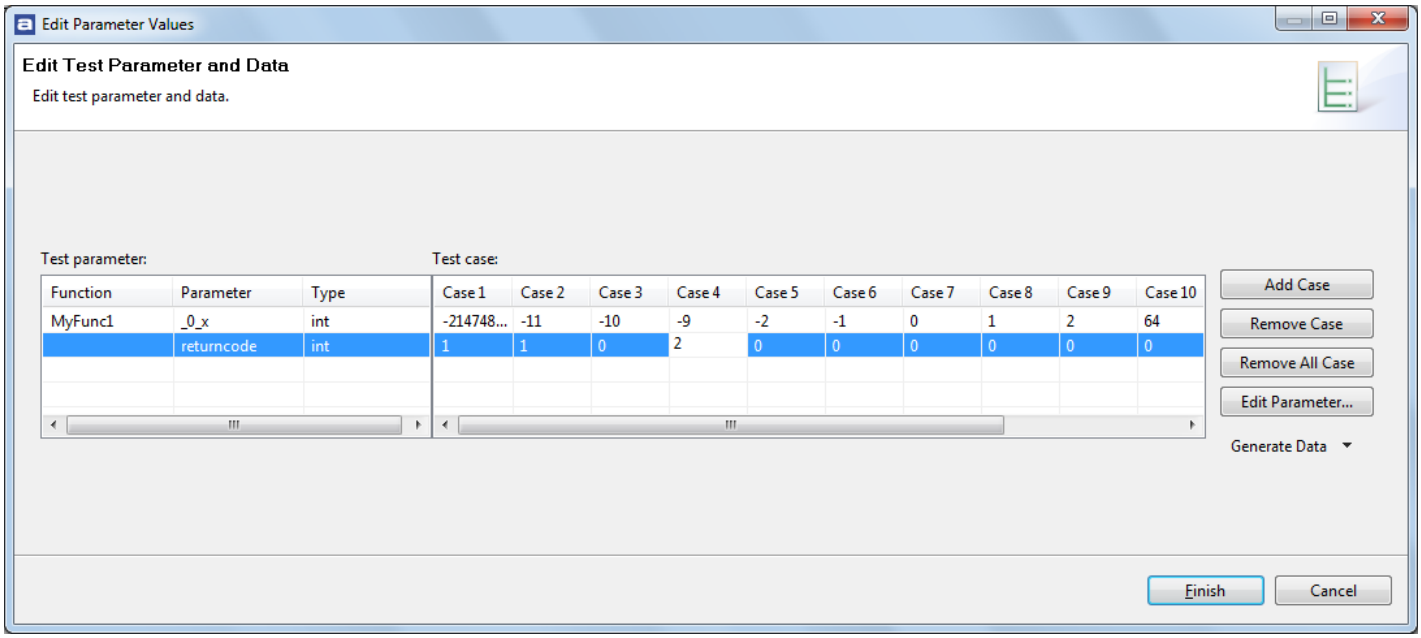


Figure 104 - The parameter editing dialog box

In fact, the return code for test case 4 is set to the wrong return value of 2 on purpose, to simulate a bug whereby the return code does not match the expected result (in this tutorial, it is easier to enter the wrong "correct" return code rather than introducing an actual bug in the source code).

Click the **Finish** button to close the parameter editing dialog box.

Finally, we must instrument the unit test source code to hook in the return code checking. Open the corresponding unit test file for this function to test, in this case the `unittest\Suite_MyFunc1.c file`:

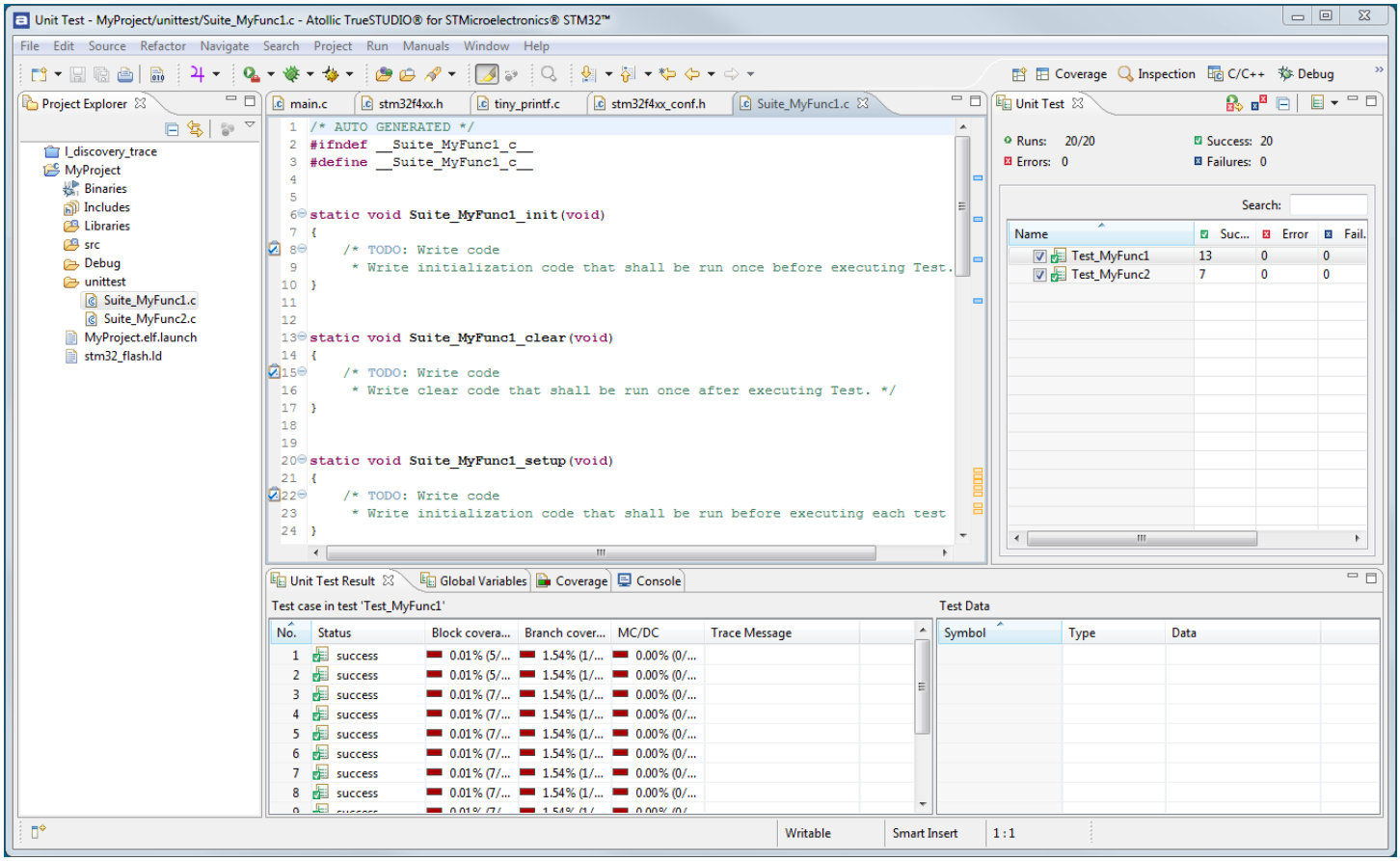


Figure 105 - Open the test case source code in the editor

Scroll down to the `Test_MyFunc1()` function (or more generically, the Test_<functionname> function). This is the piece of code that executes the unit test for this function, and developers can add their own code to check for certain combinations of input parameter values, return codes or output values of changed parameter variables.

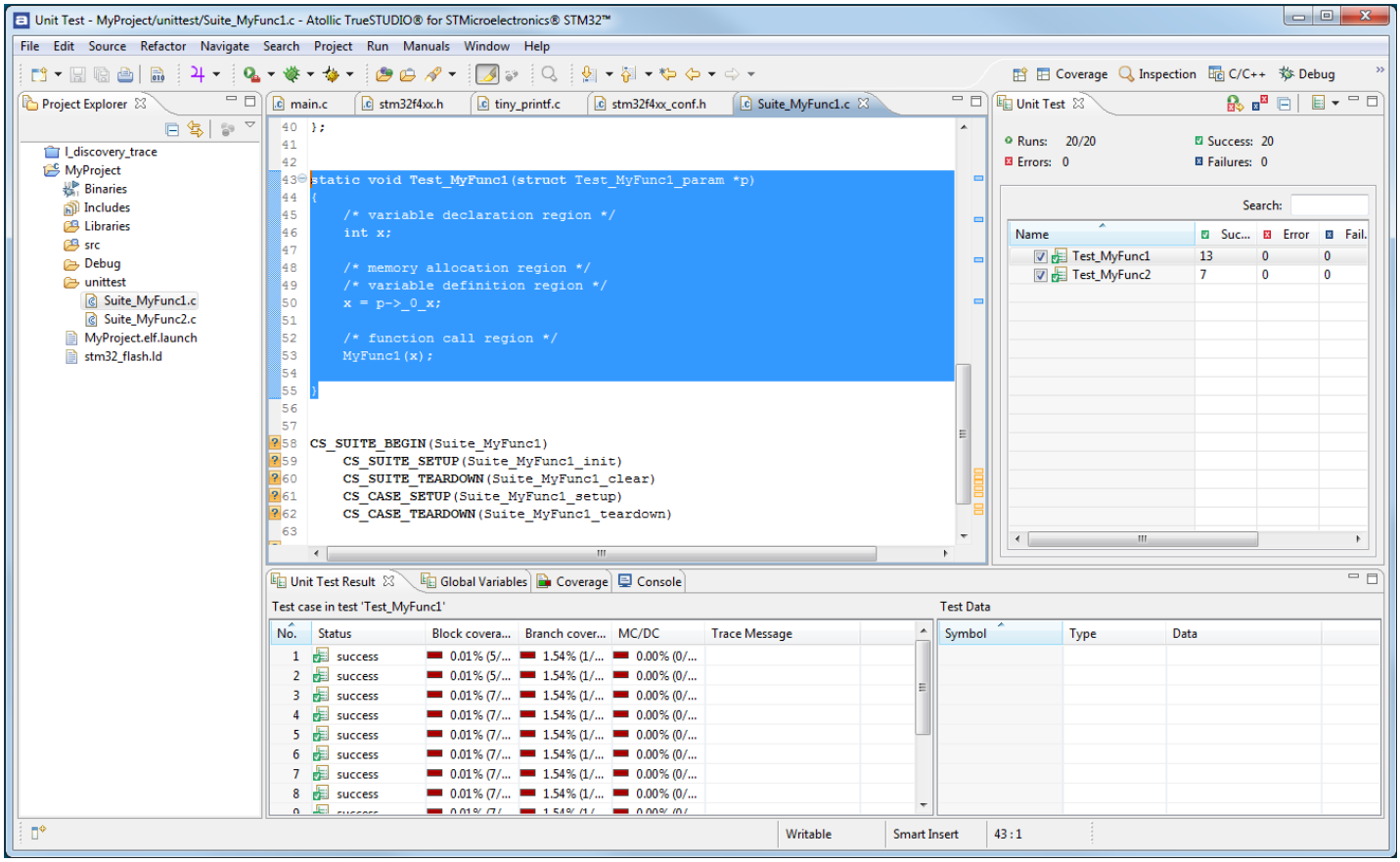


Figure 106 - The unit test function

To add checking for correct return code value as defined in the parameter editing dialog box, the function call to `MyFunc1()` is modified to save the return code, and a `CS_ASSERT` line is added after the call to validate the actual return code versus the expected one.

Finally, we add a check before the function call, to generate an error if the function call is called with -11 as input parameter (which we for the sake of this tutorial consider as an error for this function).

The `Test_MyFunc1()` function should now look like this:

```
static void Test_MyFunc1(struct Test_MyFunc1_param *p)
{
  /* variable declaration region */
  int x;

  /* memory allocation region */
  /* variable definition region */
  x = p->_0_x;

  // USERADDED: Test shall fail if x is -11
```

```
    CS_ASSERT( x != -11 );

    /* function call region */
    // USERADDED: Save return code
    int ret = MyFunc1(x);

    // USER ADDED: check return code
    CS_ASSERT( p->returncode == ret);

    /* memory free region */
}
```

The editor should now look like this:

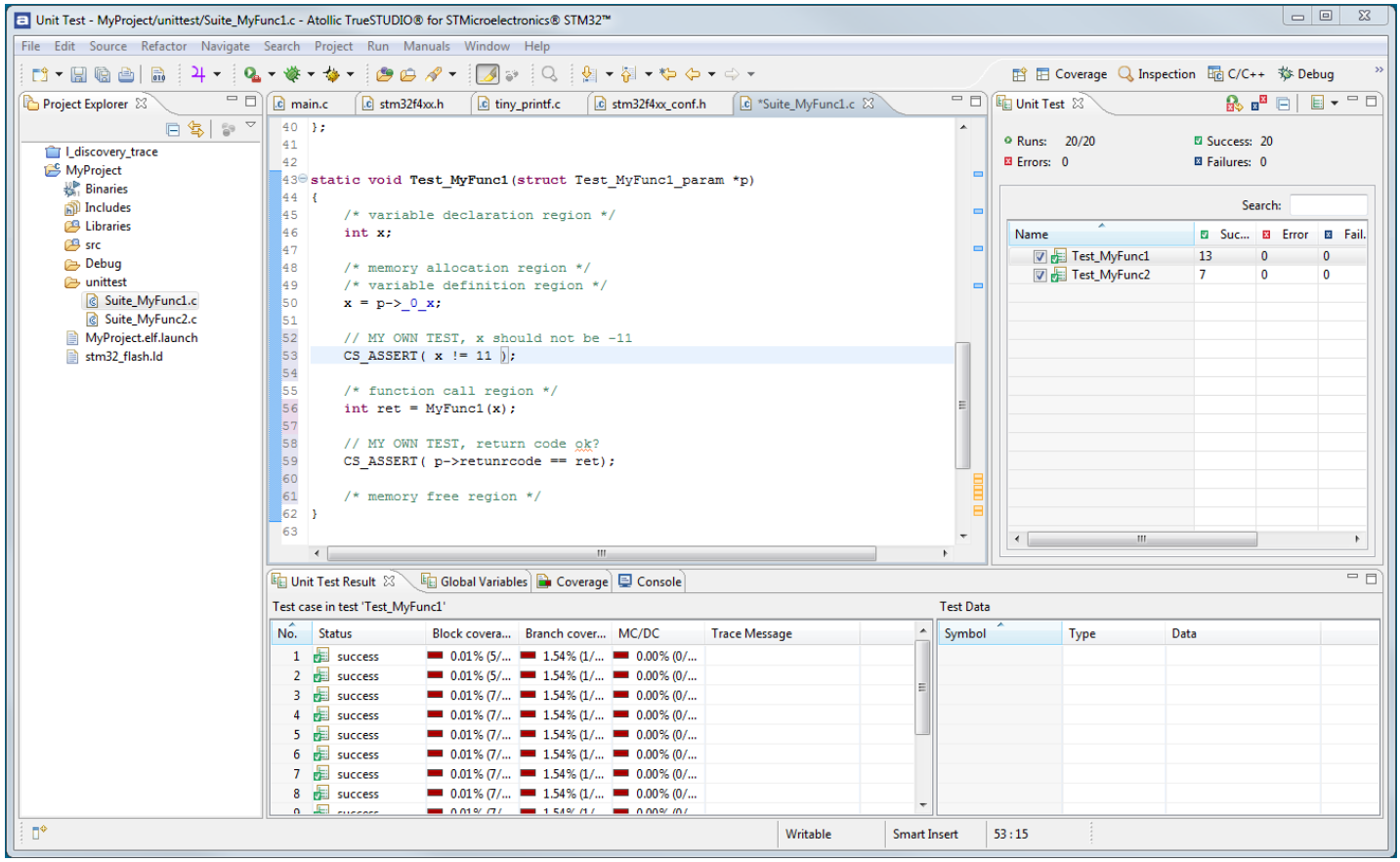


Figure 107 - The modified unit test source code

Save the modified file and re-start a test session by clicking on the **Run Unit test** toolbar button.

Once the test is completed, the test result views are updated:

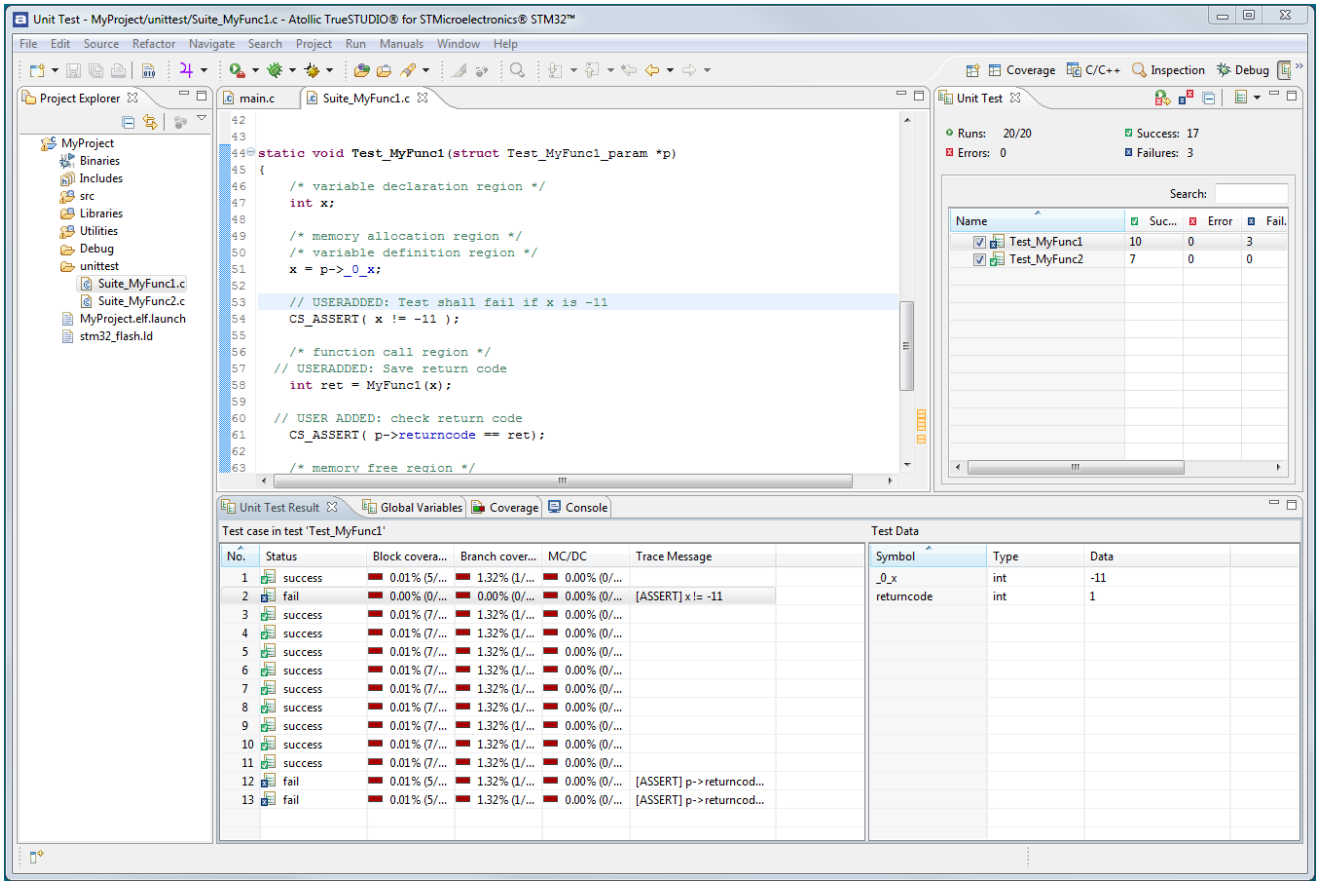


Figure 108 - Test results

As can be seen, all test cases for the function `MyFunc1()` have been successful, except for test case 2, 12 and 13, which reports test failure.

Test case 2 failed as the variable x had a value of -11, which we instrumented the test case to detect as a failure.

Test case 12 and 13 failed because the actual return code did not match the expected return value, which was caused by the fact that we introduced a mismatch on purpose when defining the correct return values in the parameter editing dialog box, in order to simulate a bug in the code.

The test cases can be instrumented with ASSERT functionality that checks any combination of input and output parameter values as appropriate.

There are two other practical test-macros worth mentioning:

1. `CS_LOG(message_string)`

```
char msgbuf[1024];
sprintf(msgbuf, "actual output: %d", return_value);
```

```
   CS_LOG(msgbuf);

2. CS_ASSERT_MSG( assert_condition, message_string)
   char msgbuf[1024];
   sprint(msgbuf, "assert actual value(%d) == expected
   value(%d)", return_value, p->rt);
   CS_ASSERT_MSG(return_value == p->rt, msgbuf);
```

CS_LOG will output the message string to the 'Trace Message' column in the 'Unit Test Result' view for each test.
CS_ASSERT_MSG will only output the message string to the 'Trace Message' column in the 'Unit Test Result' when the expression is evaluated false.

SUMMARY

Atollic TrueVERIFIER® is a tremendously powerful tool for test automation, as it auto-generates and auto-executes unit test in the actual embedded hardware. Furthermore, it does so with aircraft-grade execution-path monitoring, enabling test- and code coverage analysis up to MC/DC-level, as required by RTCA DO-178B for flight control system software.

Atollic strongly recommend you deploy ***Atollic TrueVERIFIER®*** in your project, thus improving your software quality to a completely new level!