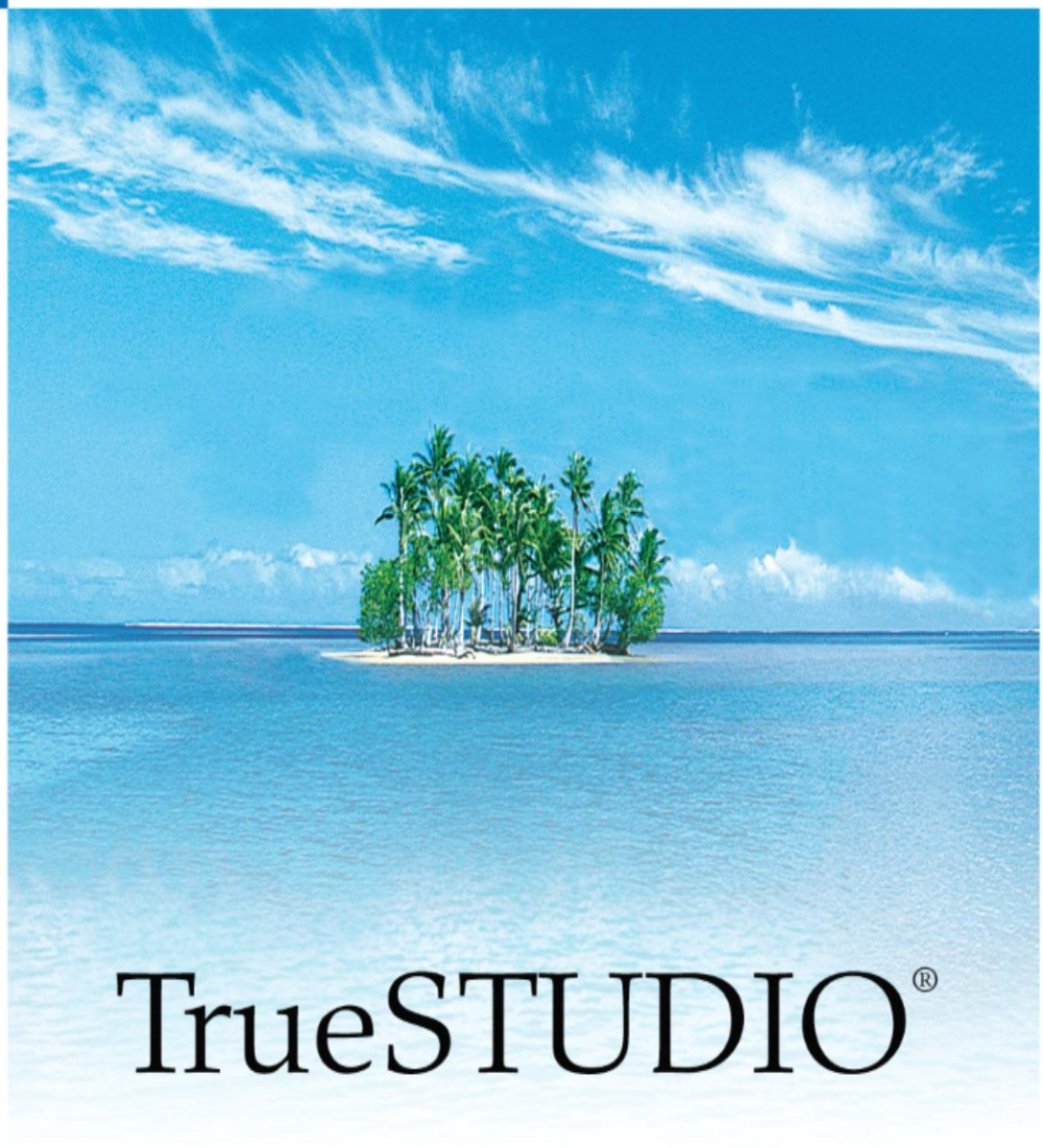
atollic



for ARM®

Migration Guide

IAR Embedded Workbench®

to Atollic TrueSTUDIO®

COPYRIGHT

TRADEMARKS

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment of Atollic AB. The information contained in this document is assumed to be accurate, but Atollic assumes no responsibility for any errors or omissions. In no event shall Atollic AB, its employees, its contractors, or the authors of this document be liable for any type of damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

DOCUMENT IDENTIFICATION

TS-MG-ARM-IAR October 2016 – Rewrite of TS-IAMG of January 2011

REVISION

1st October 2016 – Applies to Atollic TrueSTUDIO® v7.0.0

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

+46 (0) 36 19 60 50

Email: sales@atollic.com
Web: www.atollic.com

Atollic Inc.
241 Boston Post Rd. West (1st Floor)
Marlborough,
Massachusetts 01752
United States

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

Email: sales.usa@atollic.com
Web: www.atollic.com

Contents

About this Document 9

Intended Readers 9

Document Conventions 10

Section 1. Migration Decisions 11

Why migrate? 12

When to Migrate? 13

What to migrate and the implications of migration? 14

Project and build control 14

Infrastructure and work-flow 14

Application source and firmware 14

Third party O/S and libraries 15

Re-validation 15

How can migration be made easier? 17

Automated Project creation 17

CMSIS - Cortex® Microcontroller Software Interface Standard 17

Migration of Legacy Firmware 18

ABI Compliance 18

Section 2. Starting the Migration with Atollic TrueSTUDIO®. 20

Before you start 21

Workspaces & projects 21

Creating a new project 23

Configuring the project 27

Building the project 30

Build, Rebuild all 31

Importing Source Files 32

Using files in an external location 37

Using directories in an external location 38

Using IAR Eclipse plugin 41

Section 3. Migrating Source Files 51

C/C++ Source changes 52

The Pre-processor 52

Language extensions 54

Inline assembler 55

Inline functions 55

RAM based functions 56

Interrupt and exception functions 56

Nested interrupt functions 57

Non-returning functions 57

ARM® specific functions 58

Weak functions/data 58

Root functions and unreferenced data 58

Packed Data 59

Alignment of data 59

Endian setting of data 59

Non-initialised data 60

Location control of data 60

Built-in functions 63

Assembler source changes 65

Startup code 67

Section 4. Detailed Project Build Control 69

Migrating Build files 70

Compiler setup and control 71

Optimization 71

Implementation Specific Options 72

Link management 77

Linker Script/Command Files 77
Library management 84
Standard Libraries 84
Library Creation and Management 86
Migrating 3rd Party files 88
Vendor supplied ports 88
Source level porting 88
Binary level porting 88
Creating a binary interface 89
Function call/return 90
Use the compiler to create an interface for you 90

Figures

Figure 1 - Workspaces and projects 22
Figure 2 - Starting the project wizard 24
Figure 3 - C Project 24
Figure 4 - Hardware configuration 25
Figure 5 - Debugger configuration 26
Figure 6 - Open C/C++ Build settings 27
Figure 7 - C/C++ Target settings 28
Figure 8 - C/C++ Tool settings 29
Figure 9 - Adding C/C++ pre-defined symbol 29
Figure 10 - Workspace Build Preferences 30
Figure 11 - C/C++ Build Console View 31
Figure 12 - Project Explorer 32
Figure 13 - Deleting project files 34
Figure 14 - Project Explorer view 35
Figure 15 - Adding files, step 1 35
Figure 16 - Adding files, step 2 35
Figure 17 - Adding files, step 3 36
Figure 18 - Linking to files 37
Figure 19 - Linking to directories, step 1 38
Figure 20 - Linking to directories, step 2 38
Figure 21 - Linking to directories, step 3 39
Figure 22 - Project Explorer, final 40
Figure 23 - Import EWARM Eclipse project, step 1 41
Figure 24 - Import EWARM Eclipse project, step 2 42
Figure 25 - Import EWARM Eclipse project, step 3 42
Figure 26 – Delete multiple folders and files 43
Figure 27 – Drag-and-drop folders in TrueSTUDIO® 44
Figure 28 – Importing files to a project, step 1 45
Figure 29 – Importing files to a project, step 2 45
Figure 30 – Importing files to a project, step 3 46
Figure 31 – Importing files to a project, step 4 47
Figure 32 - C/C++ Include Path setting (start) 48
Figure 33 - C/C++ Include Path setting (end) 50
Figure 34 - Linker script, adding .ramfunc 56

Figure 35 - C/C++ remove <intrinsics.h> 61
Figure 36 - C/C++ enable/disable IRQ 61
Figure 37 - C/C++ Adding TIM1_UP_IRQHandler 62
Figure 38 - C/C++ finding start, end and size of sections 63
Figure 39 - Linker finding start, end and size of sections 64
Figure 40 - EWARM linker script file 80
Figure 41 - Linker script, defining symbols 80
Figure 42 - Linker script, defining memory and regions 81
Figure 43 - Linker script, adding stack and heap 81
Figure 44 - Linker script, initialized data 82
Figure 45 - Linker script, modifying memory regions 83
Figure 46 - Linker script, removing section placements 83
Figure 47 - Linker script, place sections in regions 83

Tables

Table 1 – Typographic Conventions 10
Table 2 - Files to keep, copy or link to..... 33
Table 3 - Matching example project include paths..... 49
Table 4 - IAR Embedded Workbench® Specific Predefined Symbols..... 53
Table 5 - Cross-assembler differences 66
Table 6 - Startup Code symbols..... 68
Table 7 - Compiler option cross-reference 76
Table 8 - Standard Libraries 86

ABOUT THIS DOCUMENT

Welcome to the ***Atollic TrueSTUDIO®*** Migration Guide. The purpose of this document is to help you to migrate an IAR Embedded Workbench® project to ***Atollic TrueSTUDIO®***.

INTENDED READERS

This document is primarily intended for embedded systems developers and project managers who want to understand the process of migrating a project (existing or new) from using the IAR Embedded Workbench® C/C++ compiler to ***Atollic TrueSTUDIO®*** for the ARM® processors.

DOCUMENT CONVENTIONS

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

Style	Use
`Command`	Keyboard Command or Source Code Section.
Object Name	Name of a User Interface Object (Menu, Menu Command, Button, Dialog Box, etc.) that appears on the computer screen.
Cross Reference	Cross reference within the document, or to an external document.
Product Name	Name of Atollic product.
	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 – Typographic Conventions

Section 1. MIGRATION DECISIONS

This document has been created to enable development teams to understand the need for, the mechanisms, and the implications of migrating from one development toolchain to another for a given processor or processor family.

The example used throughout the process is for migration from the IAR Systems® development toolchain to the ***Atollic TrueSTUDIO®*** IDE for ARM® processors. However the principles will remain constant across processor families and development tool vendors.

The reader should refer to the GNU C/C++ compiler documentation available with ***Atollic TrueSTUDIO®*** for detailed documentation on the usage and extensions supported by the compiler toolchain.

In addition, the Application Binary Interface (ABI) document is available directly from ARM Ltd. at the following location: www.arm.com. Finally, the user should cross-reference this information with that provided by their current compiler vendor (IAR Systems® in this example).

This section covers the high-level questions which need to be addressed before embarking on a project migration:

- Why migrate?
- When to migrate?
- What to migrate and the implications of migration?
- How can migration be made easier?

WHY MIGRATE?

Migration to a new development toolchain has to be driven by need; the need for better performance (of the embedded code), the need for standards compliance, the need for a better development workflow using higher functionality and more integrated development environments and/or the need for a better support model from the tools vendor.

The choice may be largely driven by engineering or commercial concerns, but ideally should provide benefits in both areas. As an example, the benefits of the ***Atollic TrueSTUDIO®*** IDE over its competitors could be summarized as:

- **Cost**: The ***Atollic TrueSTUDIO®*** product is partly based on open-source components that have been extended to match and surpass the feature-set in most other commercial offerings. By reusing some open-source components, the product can be offered at a substantially lower price than many other vendors.

- **Performance**: The GNU C/C++ compiler provides a world class compiler development toolchain, enhanced and maintained by thousands of developers and many companies worldwide. In recent years, it has become the de-facto standard toolchain for compiler research, further enhancing its capabilities in terms of optimization and processor support.

- **Standards**: The GNU C/C++ compiler supports C and C++ development with full support for both languages along with runtime libraries for both 'bare-metal' (where the runtime system runs directly on the processor) or Linux user-mode (where the runtime system interacts with the Linux kernel via system calls).

- **Workflow**: The ***Atollic TrueSTUDIO®*** IDE provides a modern and highly integrated development environment which directly supports the use of advanced workflow tools such as version control, bug tracking, code review, code analysis and distributed task-based development, along with tailored control for project and build control and a fully integrated debugger.

- **Support Model**: The ***Atollic TrueSTUDIO®*** IDE comes in a variety of packages enabling customers to select the features/price model best suited to their development needs. As the underlying compiler toolchain is based on the GNU C/C++ compiler, there is no worry about a 'proprietary' toolchain becoming out of date, or unavailable. The same goes for the ***Atollic TrueSTUDIO®*** IDE, as it is based on the open Eclipse framework.

WHEN TO MIGRATE?

Once the decision has been made to migrate to a new toolchain, the migration has to be planned according to the needs of the organization. Typically there are three scenarios for migration:

- At the start of a new project
- Parallel to a running project
- In a failing project to bring it back on line

Perhaps the simplest time to perform migration is at the start of a new project as the effort can be factored into the project plan, with resources and time being allocated before the project has started.

However, provided that the effort can be reasonably assessed, and the benefit from migrating can be measured (in performance, development time, cost or other terms), there is no reason why migration can't happen while a project is in progress.

Either resources can be allocated to do migration setup tasks while the rest of the team gets on with other areas of development, or the whole team can focus on the migration to enable a rapid transition.

Where companies are using version control systems, it makes sense to 'branch' the existing project to allow for migration changes to be contained in one development flow, allowing any other code changes on the original code base to be merged in as required later. In fact, as the ***Atollic TrueSTUDIO®*** IDE fully supports version control system integration, it facilitates this mode of operation.

WHAT TO MIGRATE AND THE IMPLICATIONS OF MIGRATION?

The ***Atollic TrueSTUDIO®*** IDE provides a wealth of facilities on top of the basic necessities such as the compiler toolchain, debugger and editor. It is entirely possible to phase the migration, taking advantage of certain features of the IDE when appropriate. The key areas to consider are described below.

The remainder of the document will examine some of the main issues raised.

PROJECT AND BUILD CONTROL

The ***Atollic TrueSTUDIO®*** IDE provides the ability to auto-generate projects for the supported embedded processors. These auto-generated projects provide a framework in terms of describing the source files and libraries that make up the project, and also provide a way to generate the scripts to automate the build process.

INFRASTRUCTURE AND WORK-FLOW

The ***Atollic TrueSTUDIO®*** IDE provides a complete project and build infrastructure for the GNU compiler toolchain, to include GUI level support for configuring target (processor) specific options.

It will auto-generate build scripts and linker command files which may be controlled entirely through the GUI, or edited by the user. This is however not mandatory, and so customers migrating legacy projects which have make files already may wish to continue to use them.

The IDE provides a mechanism to switch to make file use, and even provides a make file editor. Similarly, if version control and/or bug tracking systems are being used as independent applications, there is no requirement to switch to using them via the IDE. Use of such tools integrated within the IDE can be phased into the project as required.

APPLICATION SOURCE AND FIRMWARE

The majority of application code is usually written in a high level language (C or C++), in fact using common compiler extensions such as support for interrupt service routines in C, and it is possible to write nearly all of an application without using assembler.

Even assembler modules can be converted relatively simply as most cross-assemblers support similar functionality, differing only slightly in syntax (for example the way that some addressing modes are indicated, or that macros and other high level features are implemented), a simple search and replace or perhaps writing a file to map one symbol to another may suffice.

THIRD PARTY O/S AND LIBRARIES

Ideally, it should be possible to get a ported and supported version of your third party OS and/or library for the new GNU compiler toolchain. Make files and build control can then relatively simply be setup in the IDE.

Alternatively, some vendors sell source licenses, with the OS/library being provided in a portable high level language. In that case, the work is similar to that which was already undertaken when originally buying the license – i.e. configuration, build and test. The final possibility is that the OS/library is only available in a binary form, and no port for the GNU compiler toolchain is available.

It is still feasible to use a binary library as you are not changing the underlying processor being used, and for ARM® architectures there is a 'standard' Application Binary Interface (ABI) defined by ARM® which most compilers targeting ARM® processors implement.

You may be fortunate and discover that the libraries you wish to link into your new ported application will link and work without issue, however careful checking of how the two compiler implementations differ in their ABI compliance may be required. In the case of there being some difference, it is entirely possible to write an ABI compliance wrapper (in assembler) which ensures that the transition from GNU functions to legacy code works correctly.

It should be remembered that those pre-compiled binaries may have dependencies on 'standard' libraries such as the standard C library, and on compiler specific libraries such as intrinsic functions which are implicitly referenced according to the code.

Replacing the standard libraries with those provided by the GNU toolchain should present no problem, but the nature of the intrinsic libraries may mean that you have to include them in your final binary in order to make it work.

RE-VALIDATION

One of the major tasks of migration is re-validation. This is of course required, regardless of whether any code changes have been performed or not. The act of moving from one compiler to another will mean that slightly different code will be generated, as no two compilers (or even versions of a single compiler) will generate the same code, as each will optimize in a different way.

For new projects, the efforts of constructing new tests should not be any greater than with the legacy tools, for existing projects, the testing infrastructure may also require porting (depending on your application and system), which will need to be factored into the overall migration plan.

How can migration be made easier?

Firstly, the assumption in this document is that the migration does not entail switching processor architectures, and most probably that it is based on the same chip vendor and product family.

In this case there is no additional learning curve regarding the processor, the peripherals and interfaces – i.e. the system design problem has already been solved. In such a case the task is reduced to migration of project and build control, application source files and firmware.

Automated Project creation

The ***Atollic TrueSTUDIO®*** IDE supports automated, wizard-based project generation, which allows rapid creation of the project and build level control required for any project.

Part of the project generation allows the user to select the device being used (i.e. vendor and chip family), and will then auto-generate firmware code compliant to the processor/chip vendor's firmware library to support the device.

It is recommended to use the ***Atollic TrueSTUDIO®*** project generation code, whether a fully integrated build, or a makefile based build is being used, as it greatly simplifies the creation of a new project and can be used as a framework to compare to existing projects and to paste legacy files into where needed.

CMSIS - Cortex® Microcontroller Software Interface Standard

A standard firmware library infrastructure has been created by ARM Ltd. along with semiconductor and toolchain vendors. The *Cortex® Microcontroller Software Interface Standard* (CMSIS) defines a hardware abstraction layer which is available as a firmware library coded to support compilation by a number of compilers, including the GNU C/C++ compiler and the IAR Embedded Workbench® C/C++ compiler. Details can be found on the ARM® website www.arm.com.

The firmware generated by the ***Atollic TrueSTUDIO®*** IDE for the ARM® Cortex® series of processors includes all low-level device control via the CMSIS firmware library (including startup, interrupt and exception handlers) along with chip vendor supplied peripheral device drivers.

As the firmware library complies to a standard, and has been written to support both the GNU and IAR Embedded Workbench® compilers (by using conditional compilation), users

should find that they have a familiar Application Programming Interface (API) to code against, which reduces the porting exercise to one of tuning the build control and porting application source files.

MIGRATION OF LEGACY FIRMWARE

Even when the legacy project has not made use of the chip vendor's firmware library, the developer still has options on how to proceed:

1. The legacy firmware can be ported to the GNU C/C++ compiler (if a port is not already available).

2. For ARM® Cortex® processors, the legacy firmware library can be replaced within the migration project, by the CMSIS firmware library, providing a high quality and portable hardware abstraction layer which is supported and easily portable.

The initial investment in firmware porting may be significant, as firmware is by its nature at the closest level to the underlying hardware. This implies that compiler extensions have been used to directly interact with the underlying processor and peripherals to generate special functions (interrupts), control placement of data and code, control processor mode and initialization (using intrinsic functions) and control memory mapped hardware devices.

Where it is not feasible to use the CMSIS firmware library, it may be prudent to review the code to understand how the various hardware and compiler specific control is achieved.

ABI COMPLIANCE

The Application Binary Interface (ABI) defines implementation specific details of how a given toolchain supports a processor family. The ABI is usually owned and maintained by the processor vendor or on their behalf by a nominated third party.

ARM Ltd. provide and maintain a series of ABI documents which cover all aspects required for building code for the ARM® architectures on various platforms (bare-metal, Linux and mobile based). The ABI documentation set can be downloaded from the ARM® website at www.arm.com.

This document describes migration issues related to bare-metal applications, and therefore only requires an understanding of a subset of the ABI documentation.

Knowledge of the ABI is not required if migration at a C/C++ source level only is to be performed. The ABI defines the low-level information required for writing assembler functions which are callable from C/C++, and required by toolchain developers to enable interoperability between toolchains.

The ABI is therefore important at two levels:

- Assembler to C/C++ interface level (procedure call, return and stack frame definition)
- Object code format and manipulation

The IAR Embedded Workbench® and GNU toolchains both support the Procedure Call Standard for the ARM® architecture (AAPCS), which means that users can assume that functions written in assembler for IAR Embedded Workbench® based projects can be simply migrated to GNU based projects.

At the source level no changes will be required to change the calling/return mechanism. However changes may be required to conform to the instruction syntax defined by the GNU cross-assembler.

Alternatively, ABI compliance means that assembler source files which have been cross-assembled into relocatable object files using the IAR Embedded Workbench® toolchain (but not yet linked), may be linked with files built with the GCC toolchain successfully. This is because both toolchains support the same object file formats and relocation types.

Section 2. STARTING THE MIGRATION WITH ATOLLIC TRUESTUDIO®

The simplest way to start your migration project is to use the ***Atollic TrueSTUDIO***® to generate a complete skeleton project for you, including all required build control files (linker scripts and make files if required).

Once this has been created, the existing source files can be added into the project, and the build control files adjusted as necessary.

This section describes the process of creating a skeleton project, importing source files, and performing simple project configuration and build.

- Before you start
- Creating a new project
- Configuring the project
- Building the project
- Importing source files

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. Projects can contain files that are located outside the project directory using links to files and directories located anywhere.
- 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 very quick way of shifting work from one set of projects to another set of projects.

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

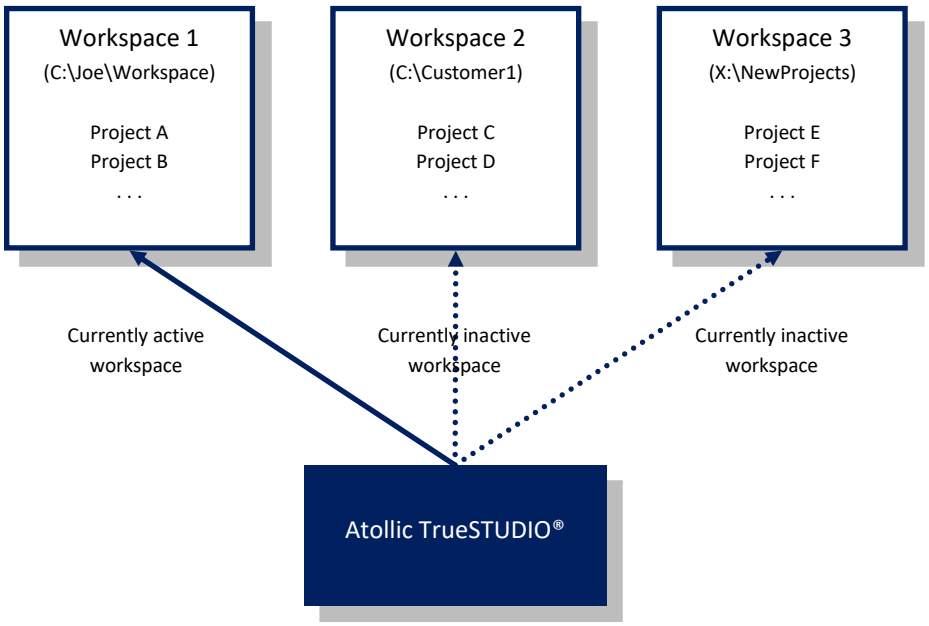


Figure 1 - Workspaces and projects

CREATING A NEW PROJECT

Atollic TrueSTUDIO® 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.

We will be using one and the same example project from IAR throughout this migration guide and we will at the end have a working copy of that project migrated into TrueSTUDIO®. The example we will be using is the IAR_STM32_SK GettingStarted project that comes with EWARM. This project is using a Cortex-M3 device family from STMicroelectronics, the STM32F103xB and we will for this exercise be using the STM32F103VB device.

Migrating a project from one tool to another can be done in many different ways. The way we will do this migration is just a suggestion but one that has been used many times and usually works well. The different steps we will take during the migration are as follows.

a) Create a C project that uses the same core/device as the original project we are migrating from

b) Configure our new project to match project setting of the original project

c) Import the source files and libraries from the original project

d) Set compiler (and if needed assembler) include file directories

e) Modify part of the source code to make it GCC compatible

f) Modify the linker script file in order to correctly locate the application in memory

As we go along the migration process we will also give you tips and ideas in general regarding migrating from one tool to another. Things that will be helpful when you migrate a more complex project than the example we use here and that is meant to give you an idea of the steps involved when doing a migration.

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

1. First, make sure you know what device the original project was using. As for us, we will be using STMicroelectronics STM32F103VB.

3. Start the C Project Wizard in TrueSTUDIO®. Click the icon as shown highlighted with a red square below.

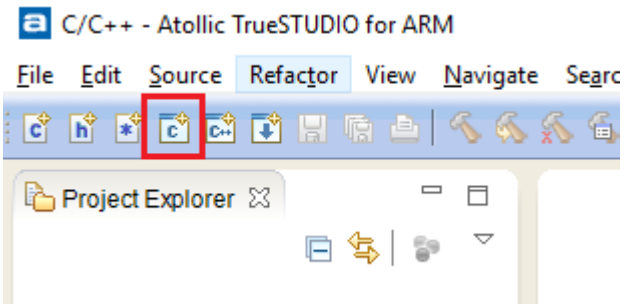


Figure 2 - Starting the project wizard

4. Name your project, select “Embedded C Project” as project type and “Atollic ARM Tools” as Toolchain. Click Next to get to the next step in the Project Wizard.

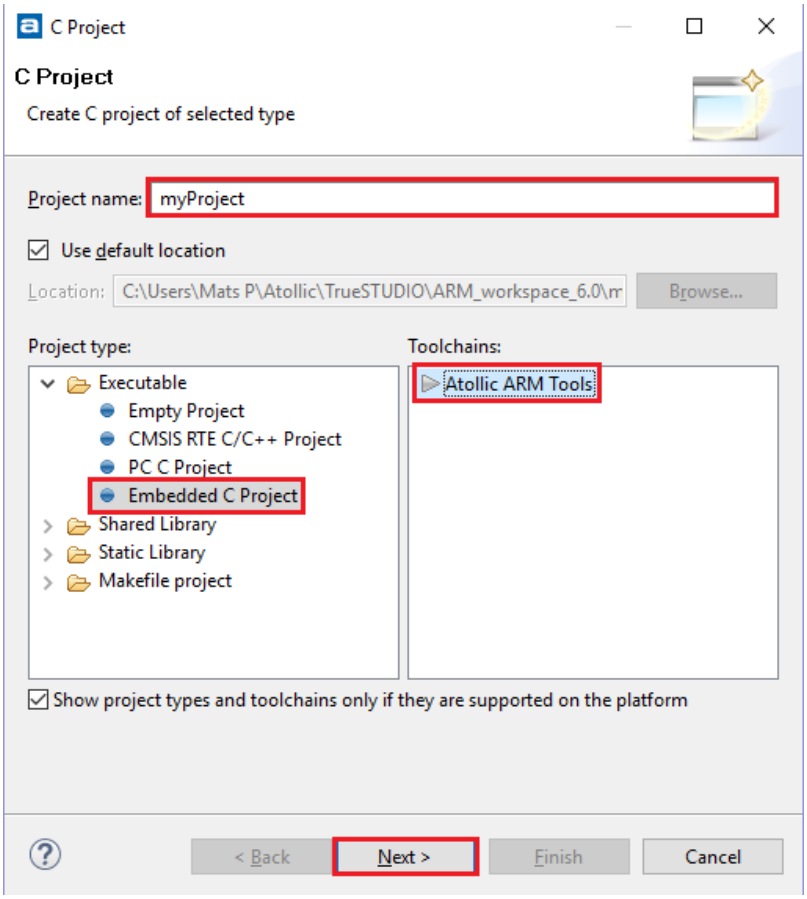


Figure 3 - C Project

5. Select Vendor, Microcontroller Family, Microcontroller and click Next.

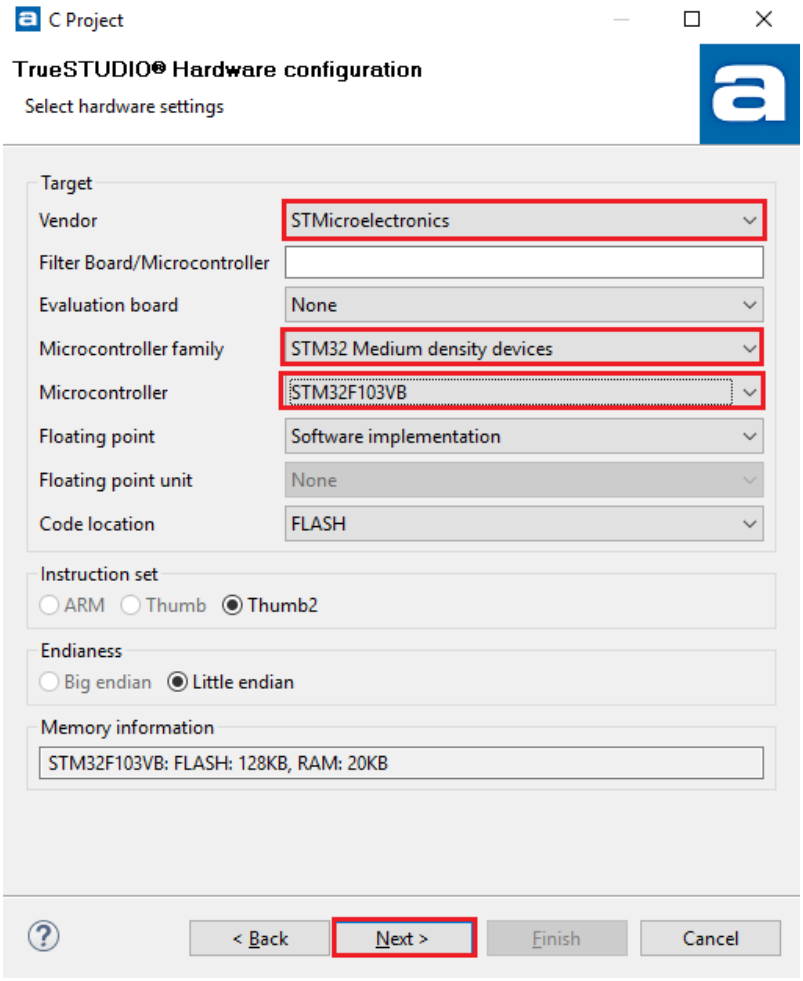
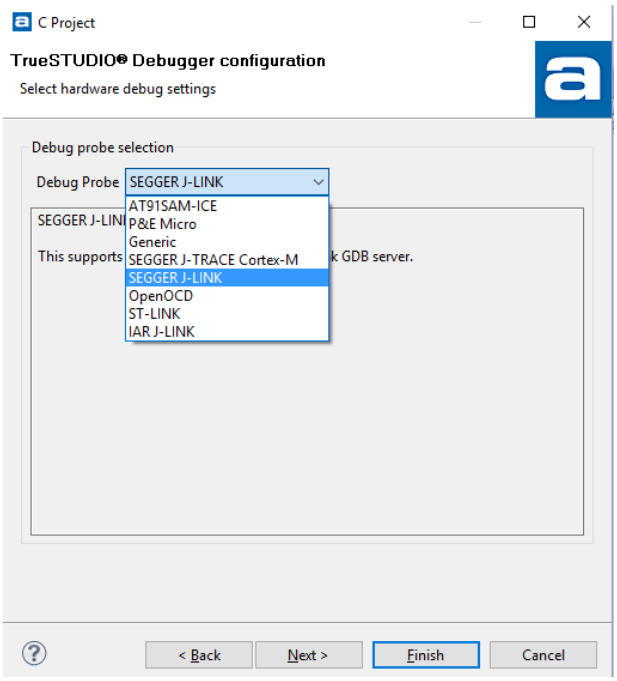


Figure 4 - Hardware configuration

6. The next page is the Software Configuration and we will accept the defaults and click Next.

7. After this we get to the Debugger Configuration page. Here we select one of the listed debug solutions, depending on what debug solution we have. For our example we will select SEGGER J-LINK.

Figure 5 - Debugger configuration



8. The last step is creating two build configurations, Debug and a Release. We accept this since we easily can add, remove or modify these build configurations later if needed.

At this point we have a simple project for a STM32F103VB device. Next we need to make sure that the most important build options are set correctly. It is helpful to have the original IAR project open when we do this, just so that we can compare build configuration settings for our two projects.

CONFIGURING THE PROJECT

Managed mode projects can be configured using dialog boxes (unmanaged mode projects require a manually maintained makefile). To configure a managed mode project, perform the following steps:

1. First select myProject in TrueSTUDIO® Project Explorer, and after that you click the "C/C++ Build settings for project 'myProject'" button  in the Toolbar.
   (It is important to make sure the top of the project is selected when configuring the entire project. It you would have a folder or file selected, then the configuration changes would only apply to that file or files in that folder.)

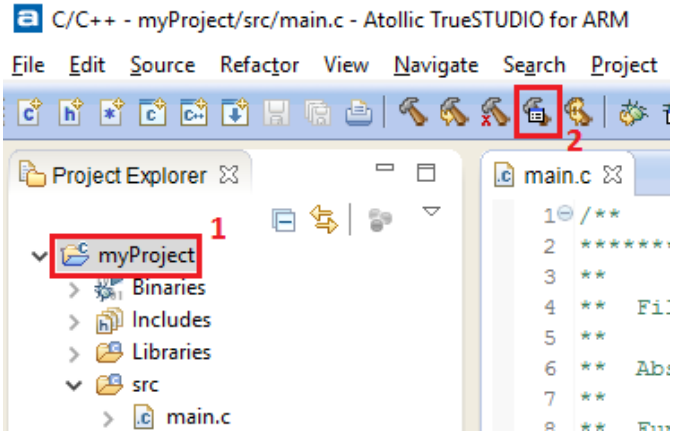


Figure 6 - Open C/C++ Build settings

2. The project Properties dialog box is displayed and by default the “Target Settings” tab is selected. Here you can view and modify the target selected if needed but we set this up during project creation so we will leave it as is. Note that project settings relevant for both managed mode projects and unmanaged mode projects are collected under the Target Settings tab.

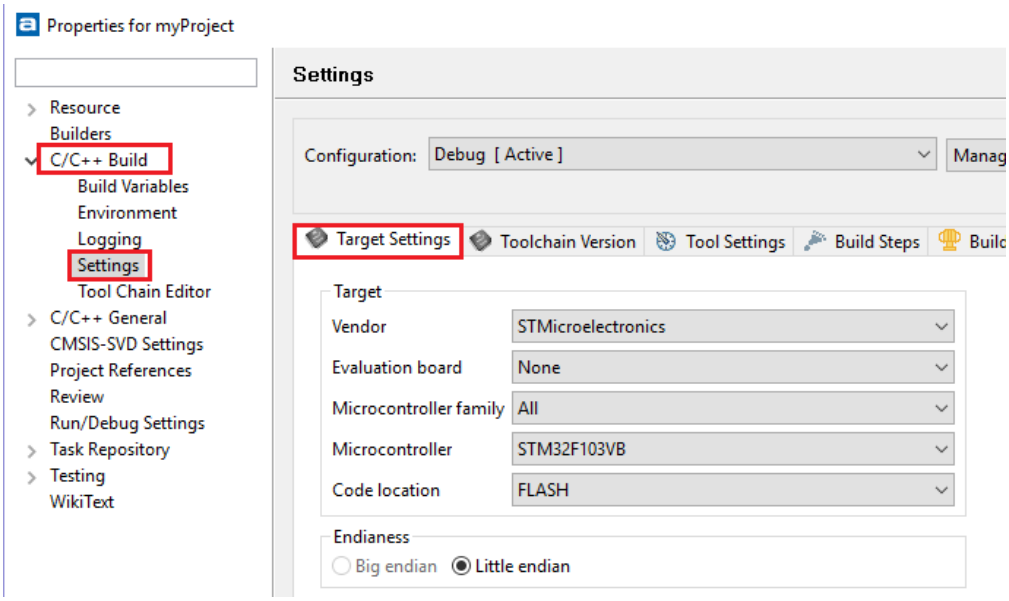


Figure 7 - C/C++ Target settings

3. Select the “Tool Settings” tab to display options for our build tools. We will have a look at some of the options here and compare them to how our original IAR project options were set. Let us first have a look at the selected C standard. From the IAR tool we see that they use C99 as default so we change our C language from gnu11 to “gnu99 (c99 + gnu extensions)”.

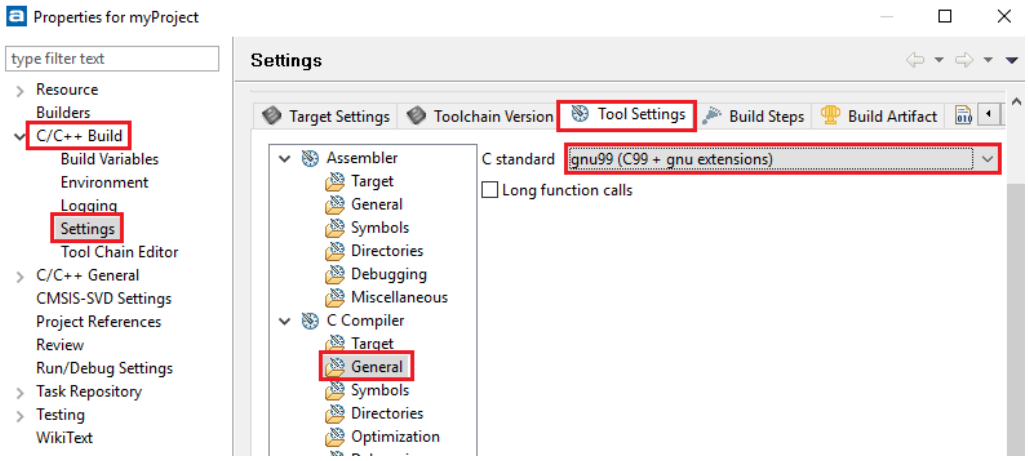


Figure 8 - C/C++ Tool settings

4. Second we will have a look at the preprocessor defines set for the compiler. The original project have EMB_FLASH , STM32F10X_MD and USE_STDPERIPH_DRIVER defined. So we are missing EMB_FLASH and to add this symbol, click “Add…” button  and type in EMB_FLASH into the Enter Value dialog that pops up. When done, click OK to commit to this new preprocessor define.

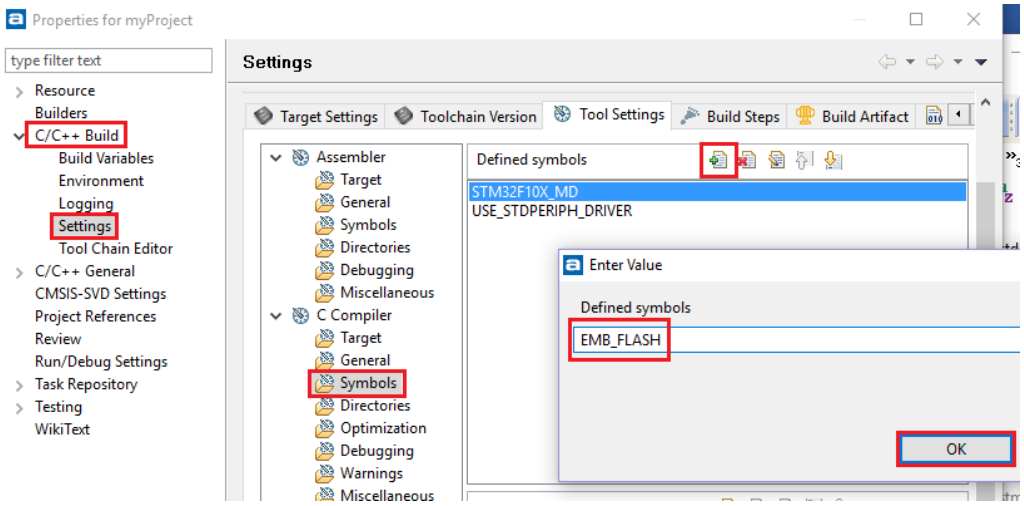


Figure 9 - Adding C/C++ pre-defined symbol

5. We are done for now configuring our build tools so we click OK to commit to the changes made and get out of the project Properties dialog.

Take some time to look around in the Tool Settings for TrueSTUDIO® to familiarize yourself with the different options here. Advanced users may want to enter command line options manually, and this can be done in the Miscellaneous panel for any tool. When done exploring the Tool Settings, click OK to accept the new settings.

In the C compiler -> Directories we have a list of include paths that the compiler will use when search for include files. These include paths differs from what you will see in the IAR tool, but this is ok for now. Once we have decided which files to add and how to add them, then we can update the list for compiler include paths and all this will be done later in this guide.

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 with the “Build automatically” option. You find this option if you select the Preferences entry in TrueSTUDIO® Windows menu. In the Preferences dialog you select General and Workspace.

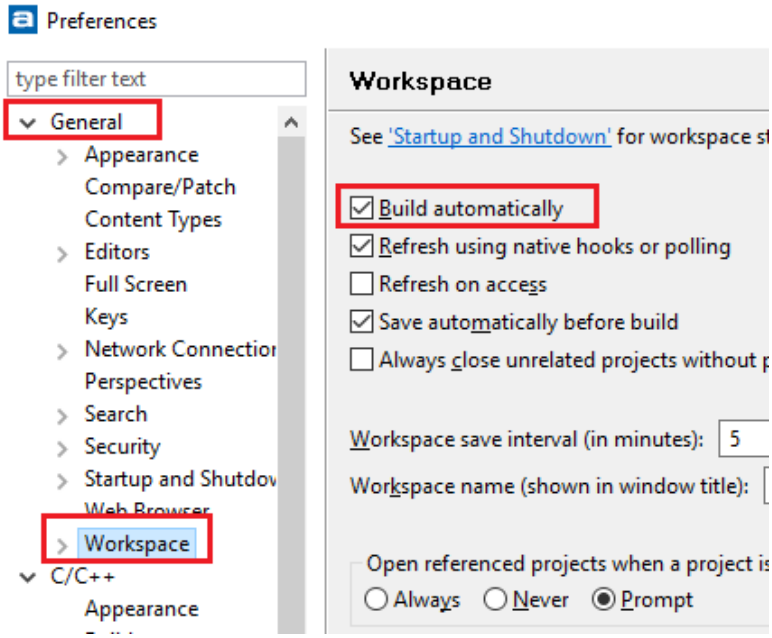


Figure 10 - Workspace Build Preferences

BUILD, REBUILD ALL

You have easy access to the different build commands from the Toolbar. You can do a Build, which would only build files that was changed since the last build. Or you can do a Clean which, would mark all source files in your project as in need of a build. A Rebuild will simply rebuild all your files, similar to doing a Clean followed by a Build.

There are by default five build buttons in the Toolbar  and if you hover over them with the mouse you will get a tooltip on what they do.

Now we do a Rebuild to ensure that we have a project that builds without warnings and errors. In our TrueSTUDIO® IDE you have a Window at the bottom that has multiple views. One is the Report view and should not have reported any errors or warnings. If you select the Console view you can see the command line options for each build step together with the output from our build tools.

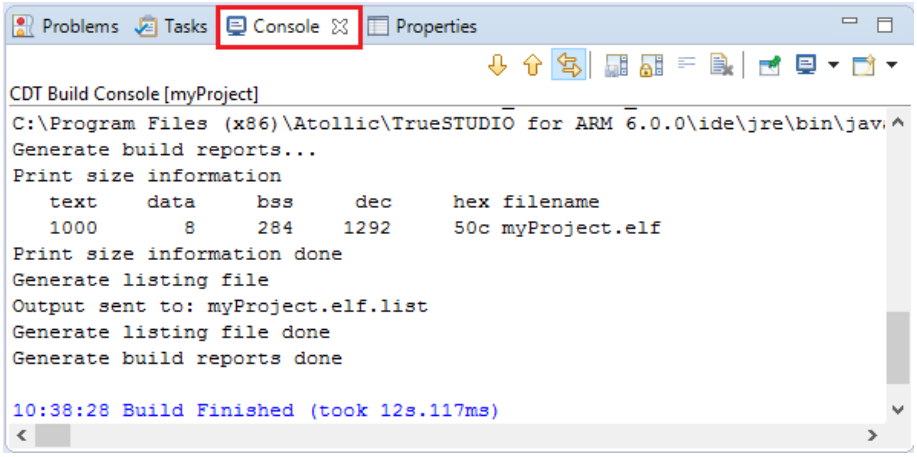


Figure 11 - C/C++ Build Console View

IMPORTING SOURCE FILES

Importing source files from the old IDE into Atollic TrueSTUDIO® is generally very simple. It is only a matter of copying the files into the Atollic TrueSTUDIO® project source directory. This can be done in Windows Explorer or from within the Atollic TrueSTUDIO® IDE.

If you have a large amount of source files in the project that you are migrating to Atollic TrueSTUDIO® then it might be worth considering using the IAR Eclipse plugin for ARM. With that plugin installed in Atollic TrueSTUDIO® you will have access to all source files for both the original EWARM project and the new Atollic TrueSTUDIO® project in the same Workspace. If this is the way you like to migrate your project, then go ahead and jump forward to section “Using IAR Eclipse plugin” below.

If you want to include specific source code files (or entire directories containing many source code files) that you wish to keep in an external location (i.e. in a location other than in the project directory tree), this can be facilitated with links to the external file or directory.

Using Project Explorer, we can have a look at the file and folder structure for our new TrueSTUDIO® project.

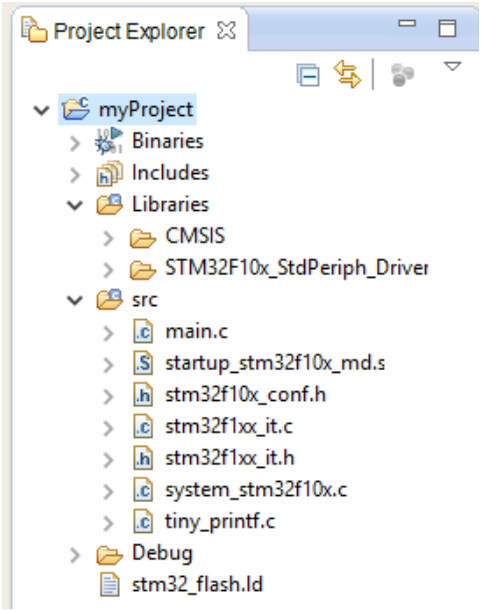


Figure 12 - Project Explorer

Our project “myProject” we have five folders. Binaries, Includes and Debug are generated for us and we can safely ignore them for now. Our source folders are Libraries and src. (The C in the folder map icon indicates that this is a source folder.)

So we can see that we have two different libraries, CMSIS and STM32F10x_StdPeriph_Driver. Going back to our original IAR project we see that these libraries are included in that project as well. So for these libraries we have two options. Either we use the new library that came with TrueSTUDIO®, and in that case we can just leave them unmodified. Or we can use the libraries we have in our original IAR project. In that case we can either copy them over to our TrueSTUDIO® project or we can link to their current IAR project location. (A rule of thumb here would be that if you will modify files, then copy them to your TrueSTUDIO® project location since modifications you make might corrupt the original project.) Since we do not expect to modify CMSIS or STM32F10x peripheral libraries we decide to link to the original library files from TrueSTUDIO®.

A closer look at the CMSIS library folder shows that the this folder in the original IAR project contains two files, startup_stm32f10x_md.s and system_stm32f10x.c. Both these files are located in the src folder for our new TrueSTUDIO® project. And the CMSIS Library folder in TrueSTUDIO® contains only header files. So if we reorganize our IAR project to TrueSTUDIO® format we really do not need the CMSIS library folder since it does not contain any C source files, only header files. We just need to make sure to point the compiler to the correct folders when the preprocessor searches for include files and we will do that in just a moment.

Next we have the src folder that contains our main application (in main.c) and some other system files. For each of these files we need to decide if we should keep it, replace it, or simply remove it. In the table below we have listed all files and added a not one how we will handle that file for our migration process.

File	Content	Decision
startup_stm32f10x_md.s	Startup code for our device	We will **keep** the TrueSTUDIO® startup file. If we had a __low_level_init function defined in IAR them we would move that code in to TrueSTUDIO®.
stm32f10x_conf.h	Configuration setup for STM32F10x_StdPeriph_Driver library	We will **copy** the original file from IAR to TrueSTUDIO® since we need to make sure we use the same peripheral configuration for both projects, but we might also want to change this configuration in the future
stm32f1xx_it.c stm32f1xx_it.h	Contains the main ISR's and the original IAR project has added a ISR handler for TIM1_UP_IRQHandler	We will **keep** the version provided by TrueSTUDIO® and just add TIM1_UP_IRQHandler and make sure it behaves the same as for the original project
system_stm32f10x.c	This is the access layer to the device drivers	We should use the same version as we use for our STM32F10x peripheral library so we will **link** to the file that is in the original IAR project
tiny_printf.c	This is a "small" version of printf, more suitable for small embedded applications	We will just **keep** this since we really don't use any printf or similar calls in our application

Table 2 - Files to keep, copy or link to

Before we start adding files from the original project and our new TrueSTUDIO® project we need to remove some source files from TrueSTUDIO® that would otherwise conflict with the new files. In our example that would be the files “main.c”, “STM32F10x_conf.h” and “system_stm32f10x.c” as well as the two library folders “CMSIS” and “STM32F10x_StdPeriph_Driver”.

To remove these files, in TrueSTUDIO® Project Explorer you right-click the file or folder to remove and select Delete.

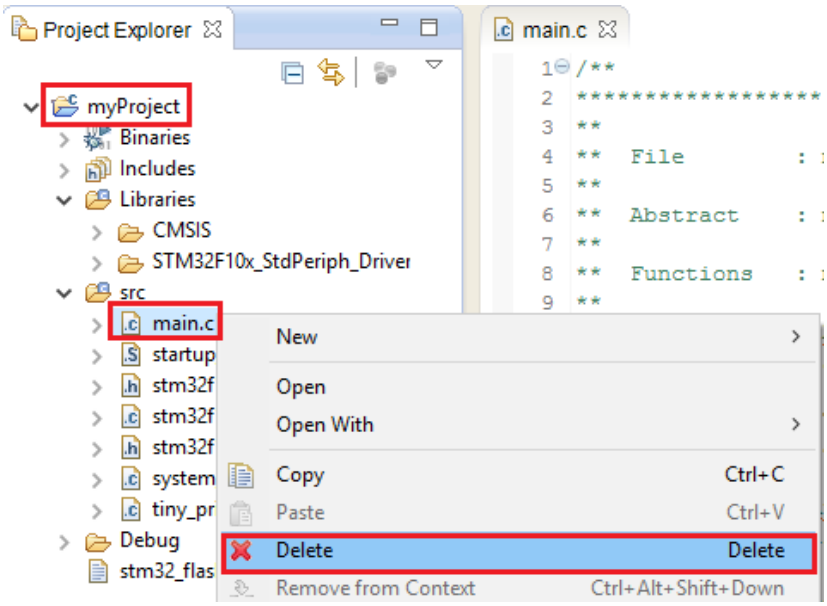


Figure 13 - Deleting project files

Do the same for files STM32F10x_conf.h, system_stm32f10x.c and folders CMSIS and STM32F10x_StdPeriph_Driver. Your Project Explorer should look something like this now.

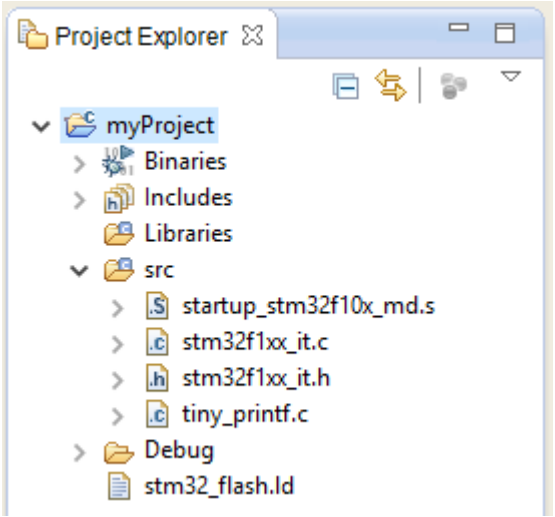


Figure 14 - Project Explorer view

Now we are ready to Import new source files, either by copying or by creating links to the original source file or folder.

If you don't know where a file is physically located in your IAR project, you can find the location if you in the EWARM IDE right-click on that file and select File Properties. The Location entry will have a path to the physical location of that file.

COPY FILES TO NEW PROJECT LOCATION

1. In TrueSTUDIO® Project Explorer, right-click on the src folder and select Import…

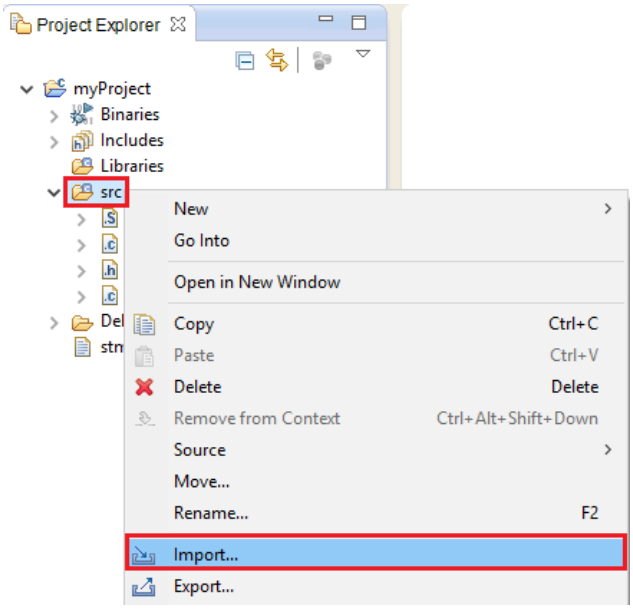


Figure 15 - Adding files, step 1

2. In the Import Dialog, select General and Filesystem and click Next.

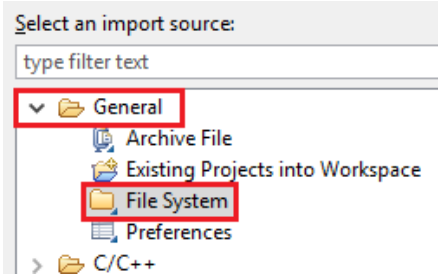


Figure 16 - Adding files, step 2

3. At last page of the import wizard, we Browse to the location of the original main.c and select main.c as well as stm32f10x_conf.h since they both are at the same location. When done we click Finish to have these two files copied to our new TrueSTUDIO® project location.

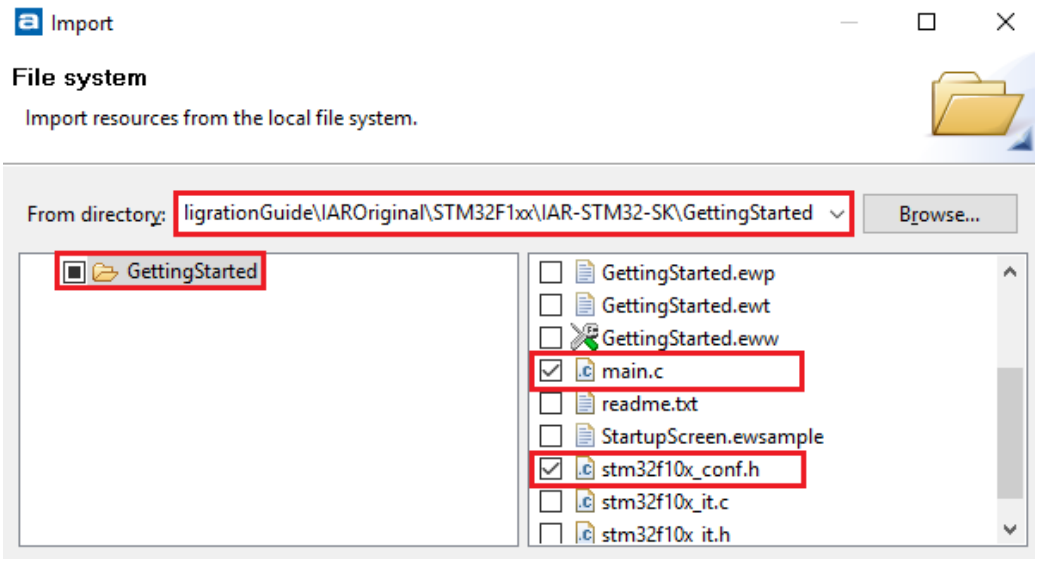


Figure 17 - Adding files, step 3

USING FILES IN AN EXTERNAL LOCATION

For the rest of the files we need we will, instead of copying files to our project, create a links to the source files. We have one file and two folders to link to and we start by creating a link to the file system_stm32f10x.c.

1. Just as we did in step 1 when copying files, we right-click on the src folder in our Project Explorer and select Import…
2. Step 2 is also the same as when copying files, so select General and File System, and click Next.
3. Now we click the Advanced button and check the box “Create links in workspace”. Then we Browse to the location of system_stm32f10x.c and after that we can select the same file in the Import dialog. When done click Finish.

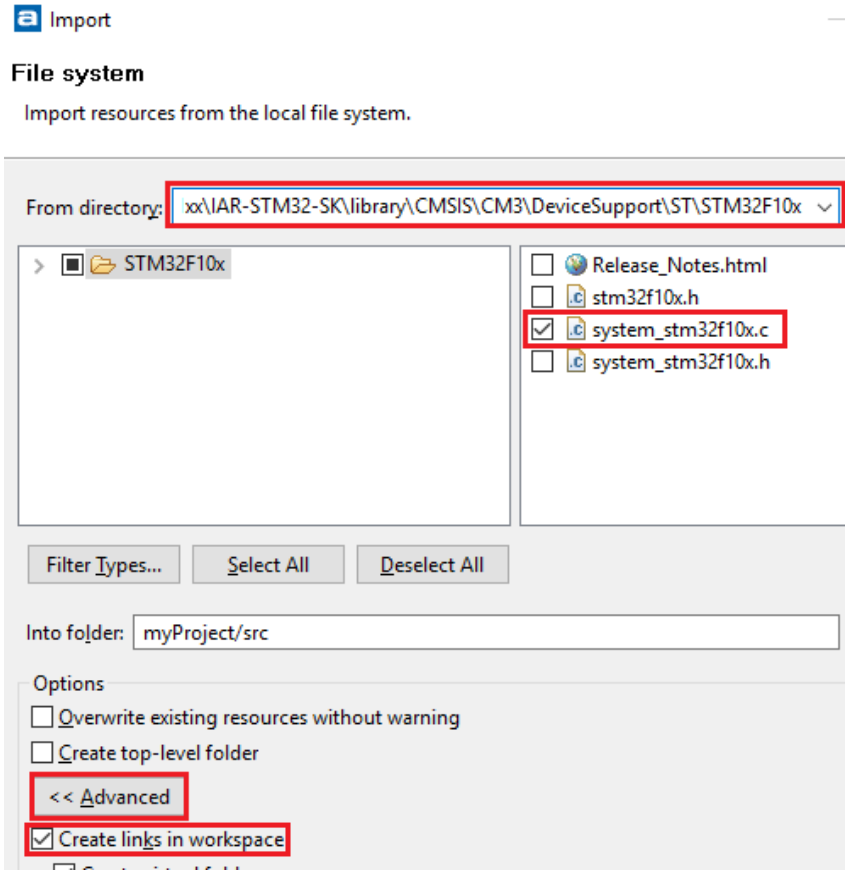


Figure 18 - Linking to files

USING DIRECTORIES IN AN EXTERNAL LOCATION

Now we will link to the folder STM32F10x_StdPeriph_Driver and with that have access to all the source files in our TrueSTUDIO® project. (Remember we did not need the CMSIS folder for source code.)

1. Right-click on the Libraries folder and select Import. We selected Libraries instead of src just to keep a nice and consistent structure for our project.

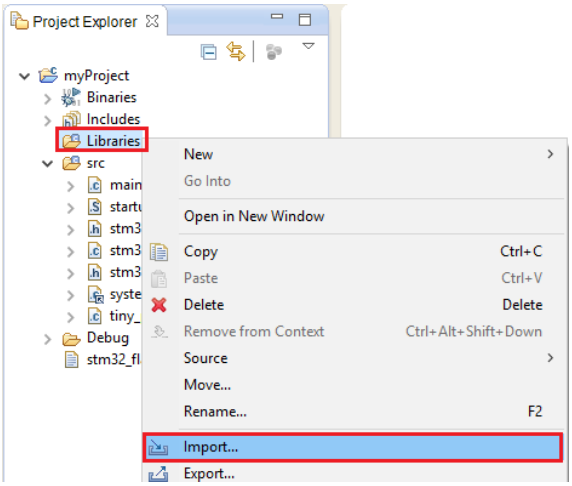


Figure 19 - Linking to directories, step 1

2. Next we select General and File System and click Next.

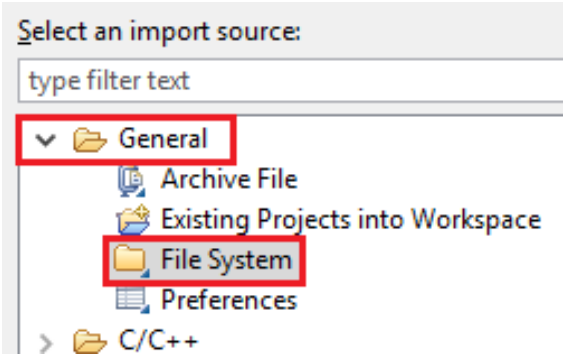


Figure 20 - Linking to directories, step 2

3. Now we select Advanced and check the “Create links in workspace” option. Then we Browse to the location of the original STM32F10x_StdPeriph_Driver. Once there we expand the content of that folder to the left in the Import dialog and check the src box. (We could also select inc to add the include folder but we don’t have to.) Click Finish when done.

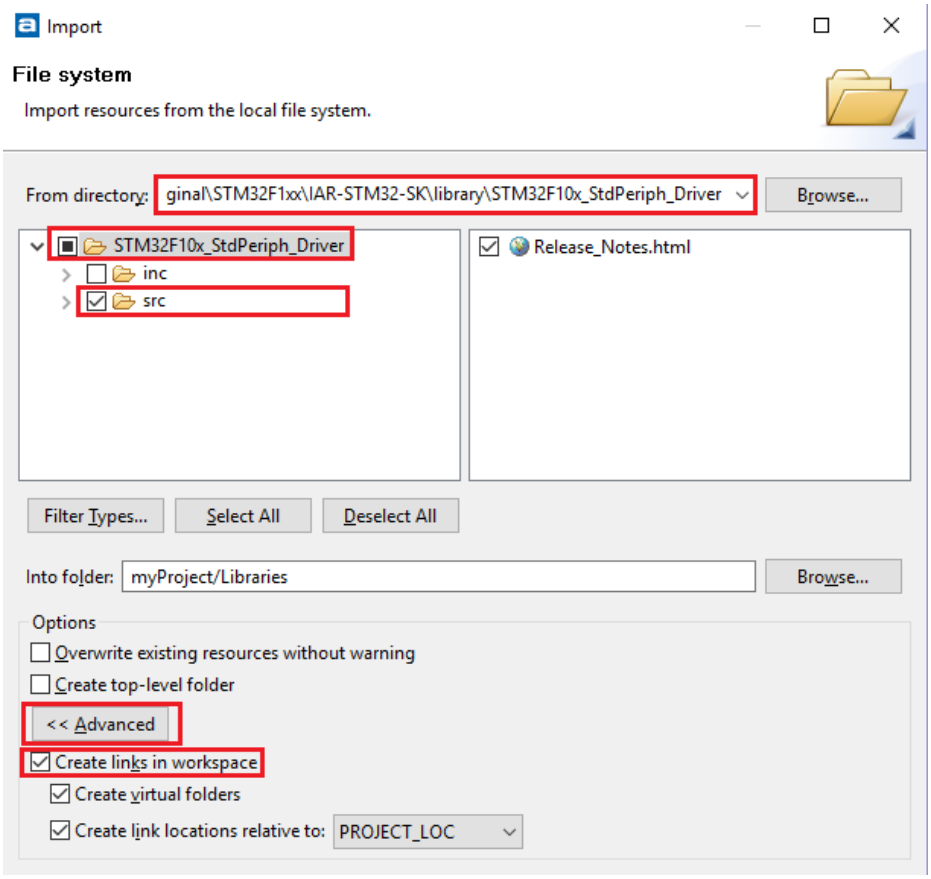


Figure 21 - Linking to directories, step 3

Now we have all our source files in TrueSTUDIO® and your Project Explorer should look something like this.

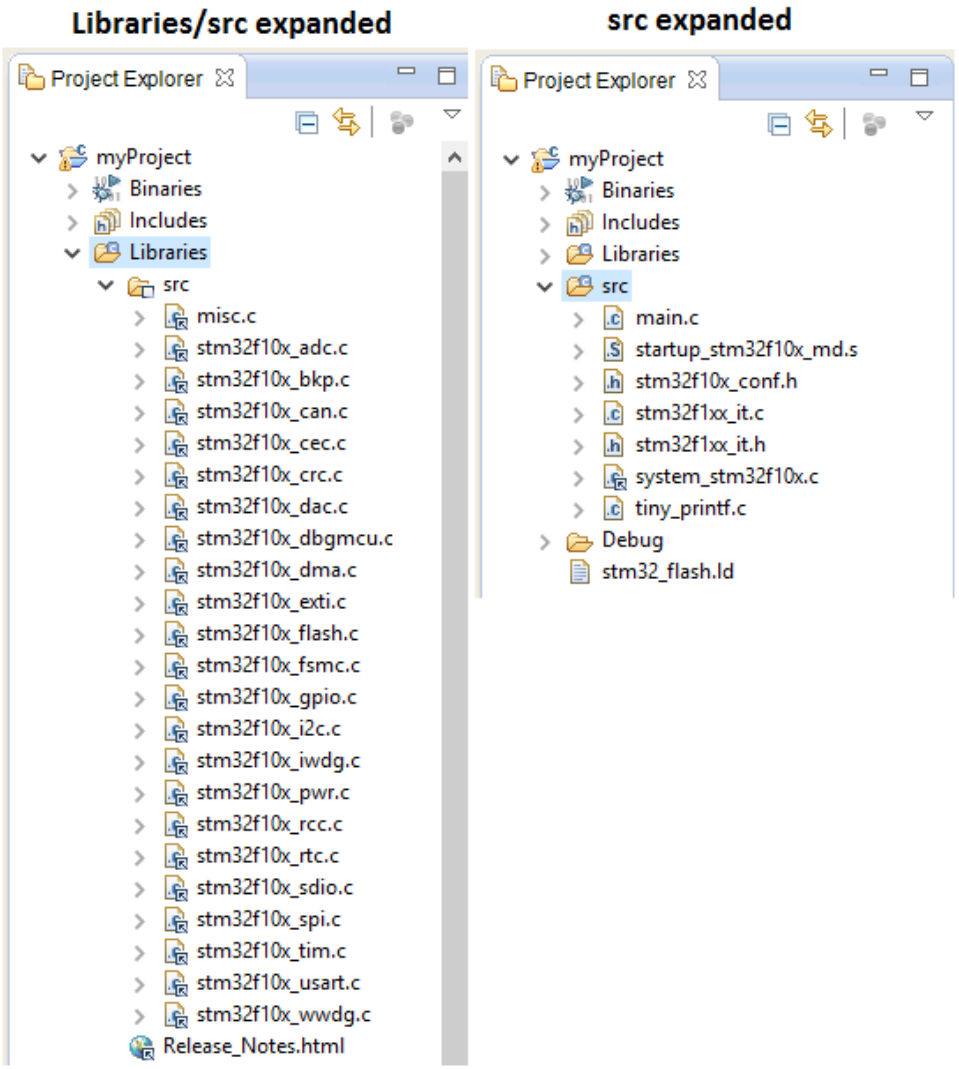


Figure 22 - Project Explorer, final

Next chapter is only for users who uses the IAR Eclipse plugin to import files and if you are not doing this, then move on directly to section "**THE PREPROCESSOR INCLUDE DIRECTORIES**".

USING IAR ECLIPSE PLUGIN

Before we can start using the IAR Eclipse plugin we have to install it in Atollic TrueSTUDIO®. The link below describes how this is done.
http://eclipse-update.iar.com/plugin-manager-install.html

Once we have this plugin installed in Atollic TrueSTUDIO® we can import the original EWARM project as described below. The basic idea when migrating IAR projects to TrueSTUDIO® using the Eclipse plugin is this. We import the original EWARM project into the same Workspace as our new TrueSTUDIO® project using links to the files in that project. When this is done we can simply drag-and-drop folders and files that we link to link to from the original EWARM project to our new TrueSTUDIO® project. For files in the original EWARM project that we need to physically have in the new TrueSTUDIO® project we will import them into the appropriate source folder.

1. First we import the original EWARM project into Atollic TrueSTUDIO®.

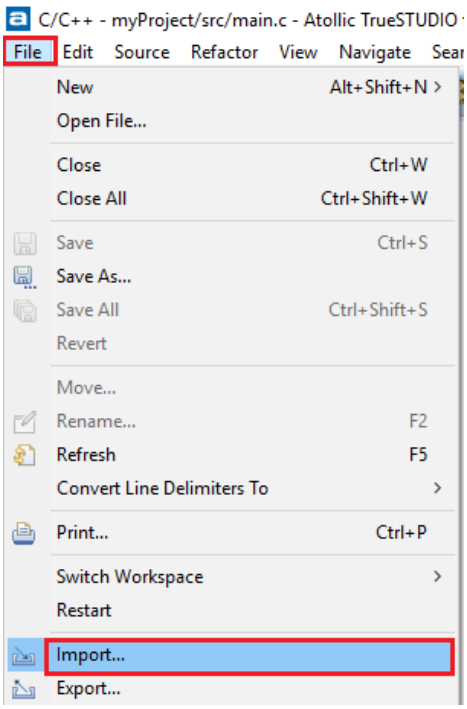


Figure 23 - Import EWARM Eclipse project, step 1

2. Next we select “Import IAR Embedded Workbench project” and click Next.

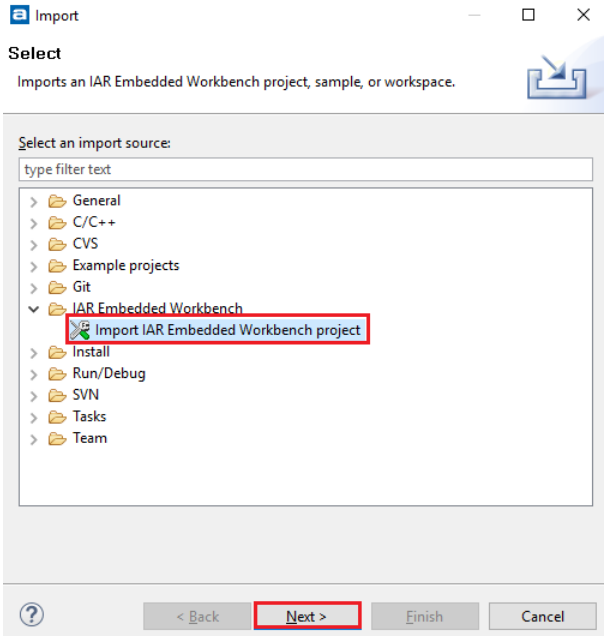


Figure 24 - Import EWARM Eclipse project, step 2

3. In the last step of this import wizard you browse to the original EWARM project file. Make sure that the option “Create links” are checked before clicking Finish.

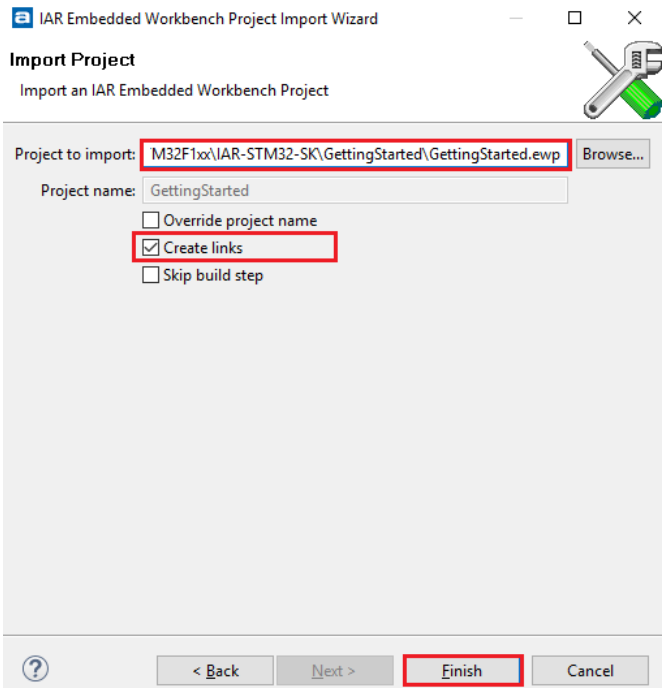


Figure 25 - Import EWARM Eclipse project, step 3

Now we have the original EWARM project we like to migrate from in the same Workspace as the new TrueSTUDIO® project we are building up. Below is a short description of what files to move to the new TrueSTUDIO® project using the IAR Eclipse plugin. For details on why we are moving some files and not others you should just read through section “Importing Source Files” above.

1. We start by deleting the folders and files in our new TrueSTUDIO® project that we later will get from the original EWARM project. The files to delete are main.c, stm32f10x_conf.h, system_stm32f10x.c and we will also delete the folder STM32F10x_StdPeriph_Driver.

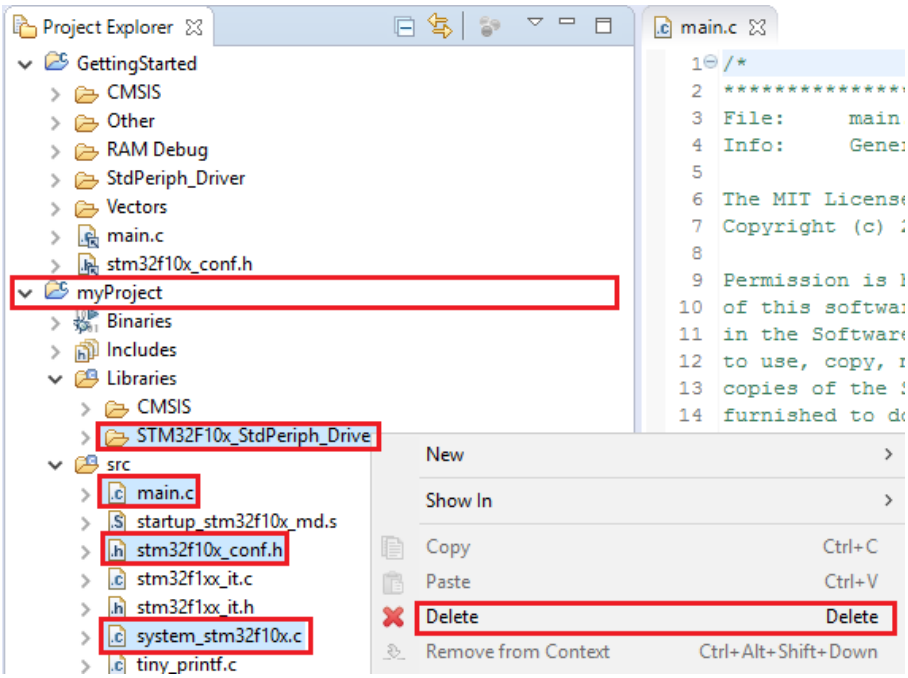


Figure 26 – Delete multiple folders and files

2. Now we need to make sure we use the STM32F1 peripheral library that were used in the original EWARM project. We can easily do this by dragging the folder StdPeriph_Driver from GettingStarted to myProject. Note that we are actually not moving any files around with this since the content in that folder links and not physical files.

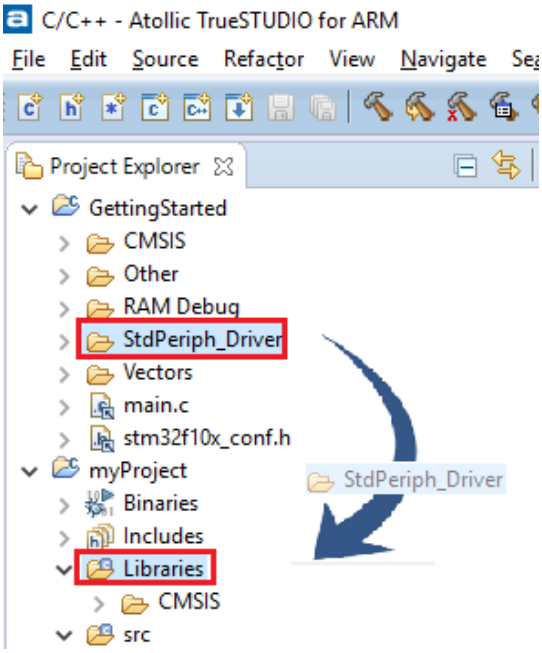


Figure 27 – Drag-and-drop folders in TrueSTUDIO®

3. Next we need to get the physical file stm32f10x_conf.h and main.c from the original project into the new TrueSTUDIO® project. We can do this by right-clicking on the src folder in myProject and selecting Import.

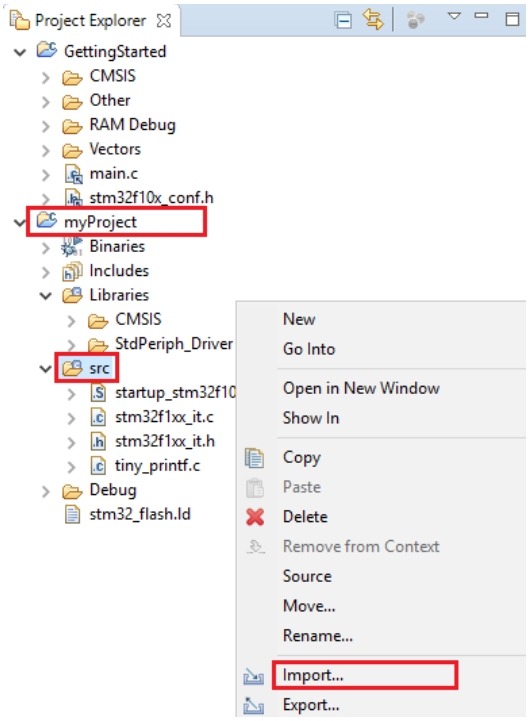


Figure 28 – Importing files to a project, step 1

4. Next we select File System and click Next.

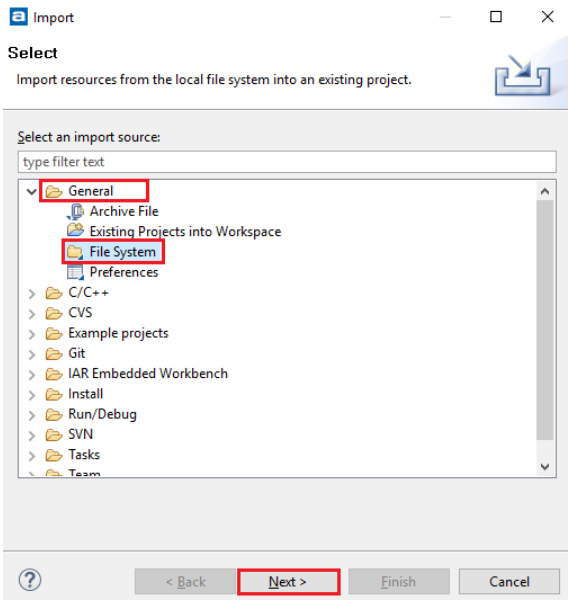


Figure 29 – Importing files to a project, step 2

5. In the last step we browse to the location of the original EWARM project and select the two files main.c and stm32f10x_conf.h. Make sure that the “Into folder” edit box says “myProject/src”.

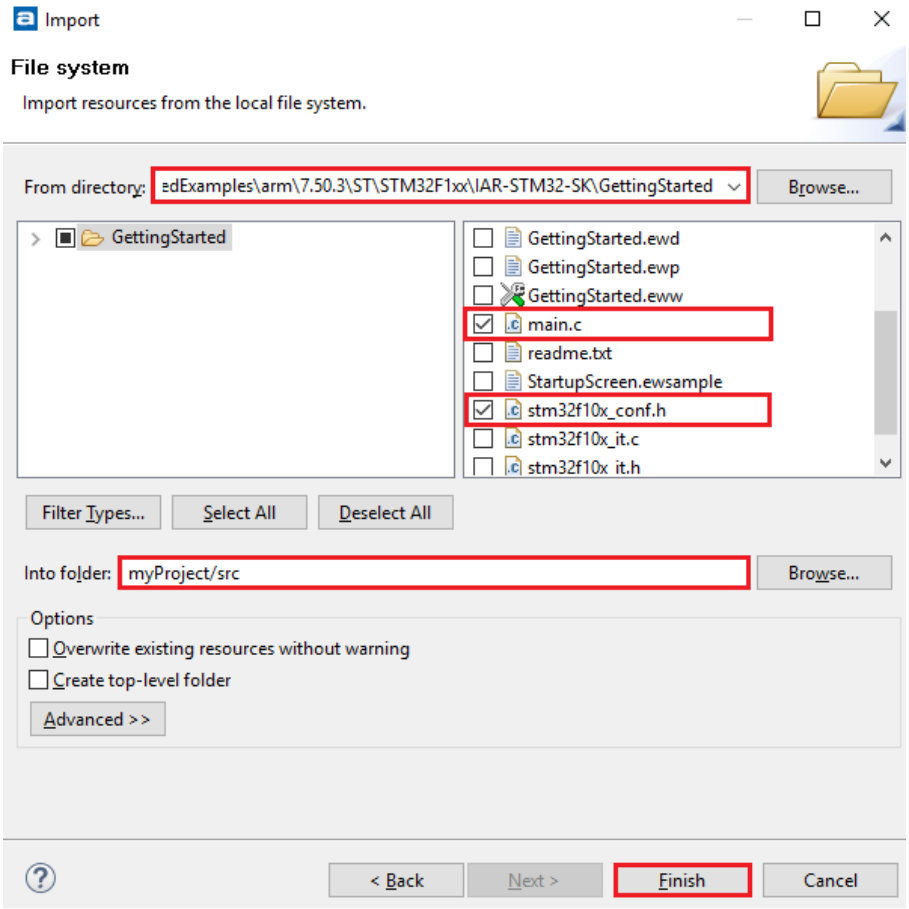


Figure 30 – Importing files to a project, step 3

6. We also need to link to system_stm32f10x.c in the original project and we can do this by importing a link to that file. It is the same procedure as when we added main.c and stm32f10x_conf.h above, except for one thing. In the final step we click the “Advanced>>” button and making sure that the “Create links in workspace” is checked before clicking Finish.

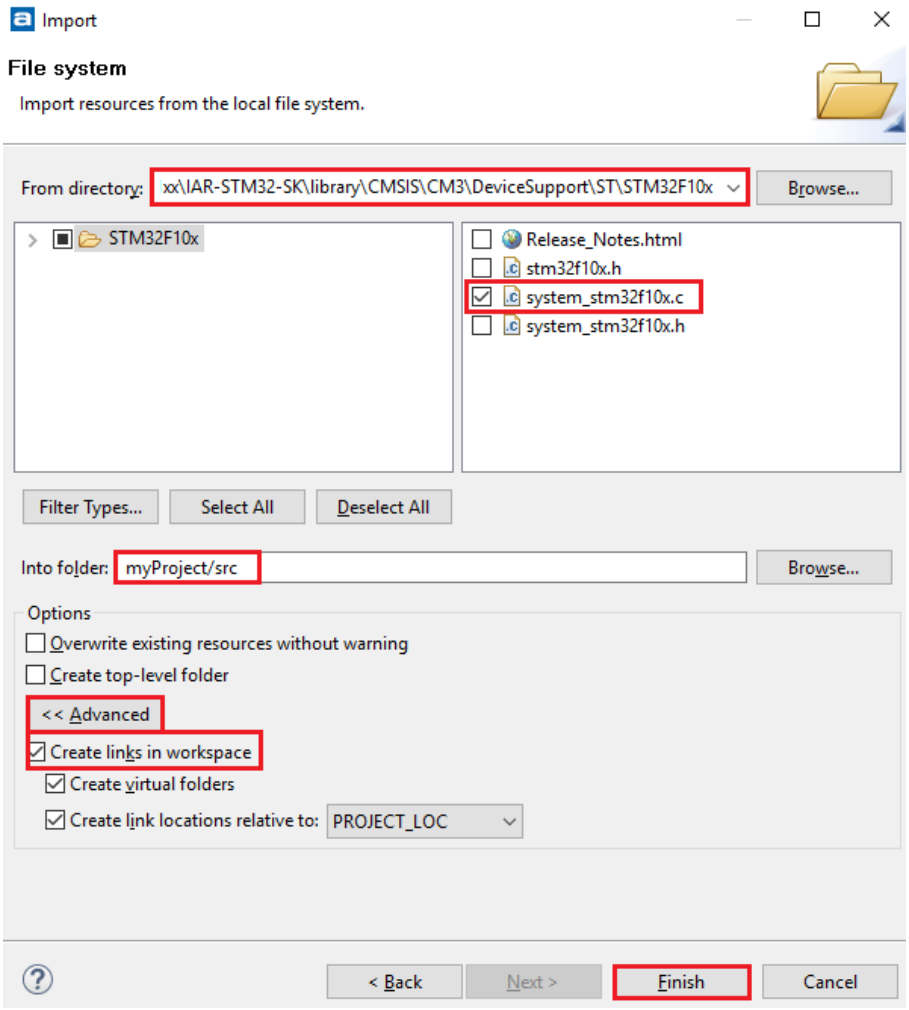


Figure 31 – Importing files to a project, step 4

Now we are done with the step to removing and adding files and folders to our new Atollic TrueSTUDIO® project and are ready to continue to the next section.

THE PREPROCESSOR INCLUDE DIRECTORIES

Before we modify the source code and do any build we need to make sure that the preprocessor will be able to find the correct include files. The safest and fastest way to do this would be to have a look at the compiler include directories for the original IAR project. When we do we will find that the following include paths were set up for that project.

```
$PROJ_DIR$\
$PROJ_DIR$\board\
$PROJ_DIR$\..\library\CMSIS\CM3\DeviceSupport\ST\STM32F10x
$PROJ_DIR$\..\library\STM32F10x_StdPeriph_Driver\inc\
```

The "`$PROJ_DIR$\`" correspond to the our "../src" include path so we have that covered. The "`$PROJ_DIR$\board\`" is a mistake by IAR since there are no "board" folder in the IAR project directory. We can safely ignore that path. The last two paths we will need to "translate" into something useful for new TrueSTUDIO® project.

There is also an extra include path used in IAR if the EWRM IDE option General options -> Library configuration -> Use CMSIS is checked. In that case the IAR will also include "<EWARM-Installation-path>\CMSIS\Include" for the preprocessor search paths. Our original IAR project do have the Use CMSIS checked so we need to add that path to our TrueSTUDIO® project.

Going back to TrueSTUDIO® and looking C Compiler Directories Include paths we see that our project corresponds almost 100% with what we have in the original IAR project.

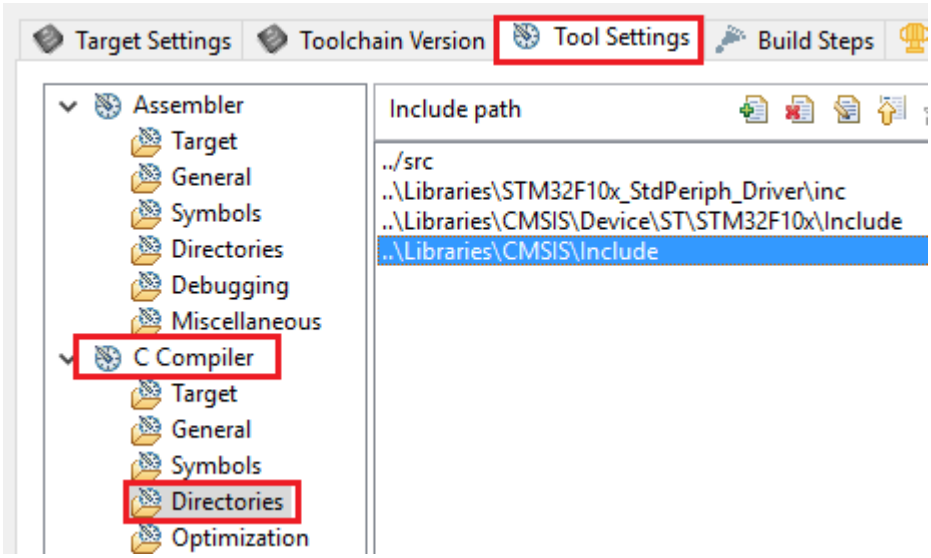


Figure 32 - C/C++ Include Path setting (start)

What we need to do now is to make sure that the include paths to STM32F10x_StdPeriph_Driver and CMSIS are correct and correct is to use the include files that are in the original IAR project.

The table below shows which include path to delete and add in order to replace the deleted path.

Delete	Add
..\Libraries\STM32F10x_StdPeriph_Driver\inc	$PROJ_DIR$\..\library\STM32F10x_StdPeriph_Driver\inc
..\Libraries\CMSIS\Device\ST\STM32F10x\Include	$PROJ_DIR$\..\library\CMSIS\CM3\DeviceSupport\ST\STM32F10x
..\Libraries\CMSIS\Include	<EWARM-Installation-path>\CMSIS\Include

Table 3 - Matching example project include paths

You will have to replace $PROJ_DIR$ with the path to the folder where your original IAR project file (GettingStarted.ewp) is located, and replace <EWARM-Installation-path> with where your EWARM installation is located. One way to find these paths is to have a look in the Messages View in EWARM. If you compile for example main.c in EWARM then you can see the full command line in the Messages View and that command line will have all the include paths expanded as parameters to the -I option. For a EWARM project located at C:\Projects\STM32\... we could have the following command line where the paths we are interested in are highlighted.

```
main.c
iccarm.exe C:\Projects\STM32\arm\7.50.3\ST\STM32F1xx\IAR-STM32-SK\GettingStarted\main.c
-D STM32F10X_MD -D USE_STDPERIPH_DRIVER
-o C:\Projects\STM32\arm\7.50.3\ST\STM32F1xx\IAR-STM32-SK\GettingStarted\RAM Debug\Obj
--no_cse --no_unroll --no_inline --no_code_motion --no_tbaa --no_clustering --no_scheduling
--debug --endian=little --cpu=Cortex-M3 -e --fpu=None
--dlib_config C:\Program Files (x86)\IAR Systems\Embedded Workbench 7.3\arm\INC\c\DLib_Config_Normal.h
-I C:\Projects\STM32\arm\7.50.3\ST\STM32F1xx\IAR-STM32-SK\GettingStarted\
-I C:\Projects\STM32\arm\7.50.3\ST\STM32F1xx\IAR-STM32-SK\GettingStarted\board\
-I C:\Projects\STM32\arm\7.50.3\ST\STM32F1xx\IAR-STM32-SK\GettingStarted\..\library\CMSIS\CM3\DeviceSupport\ST\STM32F10x\
-I C:\Projects\STM32\arm\7.50.3\ST\STM32F1xx\IAR-STM32-SK\GettingStarted\..\library\STM32F10x_StdPeriph_Driver\inc\
-Ol --use_c++_inline
-I C:\Program Files (x86)\IAR Systems\Embedded Workbench 7.3\arm\CMSIS\Include\
```

The Compiler Include path configuration page should look something like this when we are done. **Note that we need to use double-quotes (“) around paths with spaces**.

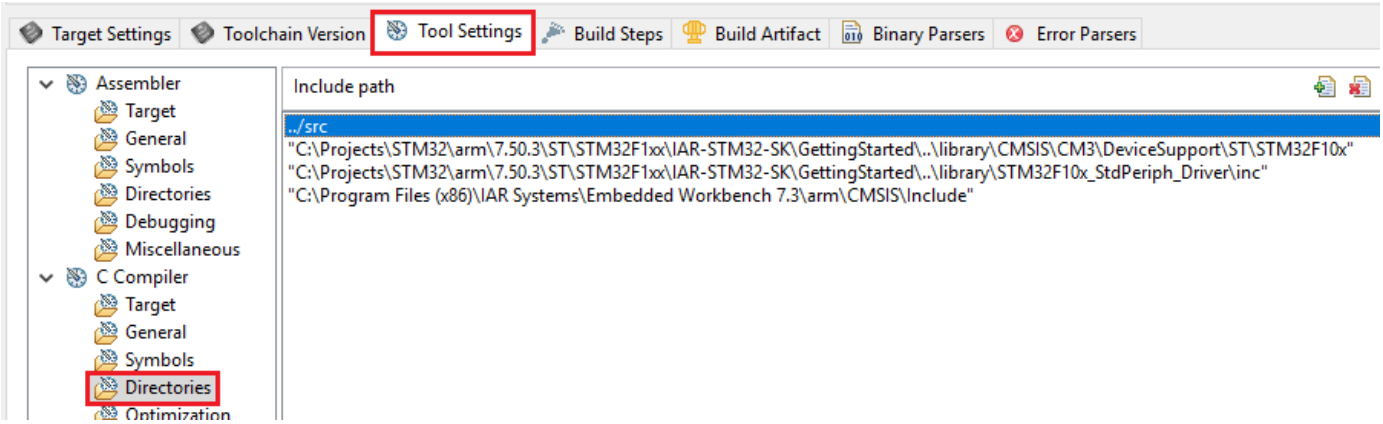


Figure 33 - C/C++ Include Path setting (end)

We are now done with adding files and setting the options to match the original project. The new TrueSTUDIO® project will not build yet without errors, but we will soon fix that as well. In the next section we will modify the source code in order to make it TrueSTUDIO® compatible and GNU.

Section 3. MIGRATING SOURCE FILES

Changes may be required to your source files in order for them to be accepted by the GNU compiler toolchain. These changes can be due to a number of factors as detailed in this section:

- C/C++ source level changes
- Assembler source changes
- Startup code

C/C++ Source changes

Dependent on your current implementation of source code files, you may or may not need to make modifications to your source code files.

Before considering any compiler extensions, it is worth noting that the level of 'compliance' to the C and C++ standards varies between all compilers.

The GNU compiler toolchain can claim a high level of standards compliance, so the chances of your legacy code using some feature of C or C++ which it doesn't support is remote.

The GNU compiler documents supplied with ***Atollic TrueSTUDIO®*** detail all aspects of compliance, and are worth checking if you are in any doubt.

Much more likely is the different behaviour of the compilers in terms of warnings and errors. You may find code that was compiling using the IAR tools with no errors, and perhaps few or no warnings, suddenly generates many warnings, and even errors.

A lot of work as gone into the GNU compiler in recent years to improve the error and warning checks, which means that it may be quite a lot stricter than you have experienced using IAR.

The recommended guidelines are that all errors should be treated as such, and serious review of the code should be undertaken to understand why the compiler is generating an error. It is possible that incorrect code either 'just worked' before, or was never actually exercised, so the bug went unnoticed.

Warnings are a matter of taste. Some companies require that all applications compile without warnings, but it is likely that due to the complexity of the code, this is not achievable.

Instead, it is worth reviewing all warnings, and where the warnings do not represent anything that you can or are willing to fix, they may be disabled using one of the many command line options.

For example, your code may use a deprecated library function. This may be intentional and therefore you do not wish your build logs to be polluted with such messages. Judicious use of command line options to reduce warnings should mean that any warnings which are generated can be easily caught, reviewed and dealt with appropriately.

The Pre-processor

The pre-processor runs before the compiler and is used to do textual insertion of header files, and definitions along with macro expansion. All C/C++ compiler toolchains support standard pre-processor directives, but also all compilers implement compiler specific definitions which allow you to write source files which can be conditionally compiled according to the compiler in use.

This allows code sharing and is typically employed in writing libraries, operating systems and cross-platform software. Therefore it is important to understand the compiler specific pre-processor directives and symbols for each compiler to enable you to migrate your source cleanly.

Some compiler extensions are also expressed using pre-processor directives known as 'pragmas'. These are however covered in the next section which looks at C and C++ language extensions.

Each compiler will have its own set of compiler specific symbols which may be used to write code that is only compiled by a specific toolchain, or even version of a toolchain. The GNU C/C++ compiler documentation can be referenced to understand the symbols provided, or alternatively the pre-processor cpp can be run directly in order to display the symbols for the specific architecture as below:

```
gcc -E -dM test.c
```

The compiler specific predefined symbols for IAR Embedded Workbench® are listed below along with the equivalent/counterpart symbols for the GNU compiler (GCC).

Note that the asterisk '*' values show where there are multiple variants of a symbol.

IAR Symbol	GCC Symbol	Comment
__BUILD_NUMBER__	__GNUC__, __GNUC_MINOR__, __GNUC__PATCHLEVEL__	GCC provides individual symbols for each level of the build number X.Y.Z
__CORE__	__ARM_ARCH_*__	Architecture variant being built for
__ARMVFP__	__VFP_FP__	VFP usage
__CPU_MODE__	__THUMB_INTERWORK__ __thumb__	Whether running in ARM, thumb or interwork mode.
__embedded_cplusplus	N/A	GCC does not have a single switch to restrict support of C++ features. Instead individual features (e.g. RTTI) can be controlled using the command line.
__IAR_SYSTEMS_ICC__	__GNUC__	Used to check the compiler toolchain in use.
__ICC_ARM__	__arm__	Symbol denoting building for ARM®
__LITTLE_ENDIAN__	__IEEE_LITTLE_ENDIAN, __IEEE_BIG_ENDIAN	Endian setting for compilation
__VER__	__VERSION__	String representation of compiler version “x.y.z”
__TID__	N/A	Target identifier, may be formed using other GNU symbols.

Table 4 - IAR Embedded Workbench® Specific Predefined Symbols

Generally the pre-defined pre-processor symbols are used to control compilation of code that has been specifically written for compilation for more than one project, architecture or toolchain. In such cases the user can simply select the GNU equivalent from the table.

The simplest way to determine the symbol(s) required is to compile a test file with the required switches, but to only run the pre-processor, with the **-dM** setting.

For example:

```
gcc -E -dM -mthumb test.c
```

Will show the **__thumb__** symbol , whereas

```
gcc -E -dM test.c
```

will show the **__arm__** symbol (which is the default).

It can sometimes be useful to have a list of the compiler defined macros when porting the source code. We can get that from the IAR compiler if add the EWARM compiler option '--predef_macros' to the command line, or to the Extra Options tab in the IDE. The syntax is:

```
--predef_macros {filename|directory}
```

Below are a couple of pages that explains some of the differences between IAR and GCC language extensions. After that section we will continue the example we started to see what we need to modify in in our code in order to make it GCC compatible, maintaining the same behavior as our original project.

LANGUAGE EXTENSIONS

All C/C++ compilers provide language extensions to allow programmers to be more productive, and to allow fine control over the code generation process. Language extensions can be classified in the following way:

- Generic extensions, not targeted to any architecture, but providing some feature seen to be missing in the language (e.g. inline functions in C, macros that can return values).
- Extensions designed to allow the programmer to control the placement of code and/or data (e.g. definition of the owning section).
- Extensions to provide special 'attributes' to functions or data to change the way they are treated by the compiler (e.g. interrupt handler functions, packed structures).
- Extensions to provide access to low level programming (in line assembler and intrinsic functions).

The extensions can be implemented as one or more of the following:

- **Keywords:** Very commonly used features (such as inline functions, or inline assembler) warrant a keyword to enable clear code to be written. The number of extended keywords is usually restricted to limit 'namespace pollution'.
- **Attributes** set via a single __attribute__ keyword: This mechanism allows any number of attributes to be applied to the definition of a data object or function using a single extended keyword. This is the recommended approach for using the GNU C/C++ compiler.
- **#pragma directives:** These look like pre-processor directives, and have implementation specific behavior. They have the advantage of not requiring any new keywords to support extended features, but suffer from the fact that they cannot be used in macros (like other directives), and usually toggle a feature on/off for the rest of the file, or until another #pragma directive is used to change the behavior.

The GNU Compiler toolchain provides a large number of language extensions, allowing for generation of powerful and highly targeted code. This document does not cover all GNU extensions, but rather examines the extension provided by the IAR Embedded Workbench® toolchain, and discusses how they may be supported in a file to be compiled using the GNU toolchain.

INLINE ASSEMBLER

	IAR	GCC
Keyword:	`__asm`	`__asm`
Pragma:		
Syntax:	`__asm("<assembler>")`	`__asm("<assembler>")`
Comment:	GCC supports a superset of the IAR Embedded Workbench® functionality. Allowing for C level variables to be accessed from the assembler. Advanced inline assembly is beyond the scope of this document. Please see the GNU compiler manual for details. Note that GNU uses the semi-colon character to delimit instructions, whilst the IAR Embedded Workbench® toolchain uses the newline character '\n'.	

INLINE FUNCTIONS

	IAR	GCC
Keyword:	`inline`	`inline`
Pragma:	`inline`	`inline`

Syntax:	`#pragma inline`
	`void foo (void)`
	`{…}`
	`inline void foo2 (void)`
	`{...}`
		`inline void foo2 (void)`
		`{...}`

Comment:	GCC supports only the keyword. The pragma extension is seen to be unnecessary, particularly with C99 support for inline functions.	

RAM BASED FUNCTIONS

	IAR	GCC
Keyword:	`__ramfunc`	`__attribute__((section("NAME")))`
Pragma:		
Syntax:	`__ramfunc void foo (void) {...}`	`__attribute__ ((section (".ramfunc"))) void foo (void) {…}`
Comment:	GCC does not directly support RAM functions which are copied from ROM to RAM at startup. However this uses the same mechanism as initialized data, and so can be achieved by creating a section to contain RAM functions (".ramfunc" in the example), and modifying the linker script as shown below.	

```
/* Initialized data sections goes into RAM, load LMA copy after code */
.data :
{
  . = ALIGN(4);
  _sdata = .;        /* create a global symbol at data start */
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */
  *(.ramfunc)
  *(.ramfunc*)

  . = ALIGN(4);
  _edata = .;        /* define a global symbol at data end */
} >RAM AT> FLASH
```

Figure 34 - Linker script, adding .ramfunc

INTERRUPT AND EXCEPTION FUNCTIONS

	IAR	GCC

Keyword:	`__swi`
	`__fiq`
	`__irq`
		`__attribute__((interrupt("SWI")))`
		`__attribute__((interrupt("FIQ")))`
		`__attribute__((interrupt("IRQ")))`

Pragma:	`swi_number=NN`	
Syntax:	`__irq void foo (void) {...}`	`void foo (void)__attribute__ ((interrupt("IRQ"))) {...}`
Comment:	The IAR Embedded Workbench® "#pragma swi_number=N" is required in addition to the __swi keyword to define the SWI number used.	

NESTED INTERRUPT FUNCTIONS

	IAR	GCC
Keyword:	`__nested`	`__attribute__((nesting))`
Pragma:		

Syntax:	`__nested __irq`
	`void foo (void)`
	`{...}`
		`void foo (void)__attribute__ ((nesting))`
		`{...}`

Comment:	The tagged function is marked as non-returning.	

NON-RETURNING FUNCTIONS

	IAR	GCC
Keyword:	`__noreturn`	`__attribute__((noreturn))`
Pragma:		
Syntax:	`__noreturn void foo (void)` `{...}`	`void foo (void)__attribute__ ((noreturn))` `{...}`
Comment:	The tagged function is marked as non-returning.	

ARM® SPECIFIC FUNCTIONS

	IAR	GCC

Keyword:	`__arm`
	`__thumb`
	`__interwork`
		__attribute__((arm))
		__attribute__((thumb)

Pragma:		`long_calls`
Syntax:	`__arm void foo (void)` `{...}`	
Comment:	The command line options can be used to enable any of the above at a module level. No feature is directly added to support control at an individual function level in GCC currently. However the **long_call** and **short_call** attributes can be used to achieve calling to/from thumb functions from code located far away in the address map.	

WEAK FUNCTIONS/DATA

	IAR	GCC
Keyword:	`__weak`	`__attribute__((weak))`
Pragma:	`weak`	`Weak`

Syntax:	`__weak int i;`
	`__weak void foo (void)`
	`{...}`
		`int i __attribute__((weak));`
		`void foo (void) __attribute__((weak))`
		`{...}`

Comment:	The option to generate interwork code is set as a command line option in ***Atollic TrueSTUDIO®*** and is only available for targets that support both ARM and Thumb instructions.	

ROOT FUNCTIONS AND UNREFERENCED DATA

	IAR	GCC
Keyword:	`__root`	`__attribute__((used))`
Pragma:	`required`	

Syntax:	`#pragma required=i`
	`int i;`
	`__root void foo (void)`
	`{...}`
		`int i __attribute__((used));`
		`void foo (void) __attribute__((used))`
		`{...}`

Comment:	IAR Embedded Workbench® uses two separate mechanisms to force the inclusion of unreferenced code/data at link time, GCC uses a single attribute.	

PACKED DATA

	IAR	GCC
Keyword:	`__packed`	`__attribute__((packed))`
Pragma:	`pack`	`Pack`

Syntax:	`#pragma pack(4)`
	`struct tag1`
	`{`
	`  char a;`
	`  int b;`
	`} mystruct1;`
	`#pragma pack()`
	
	`__packed struct tag1`
	`{`
	`  char a;`
	`  int b;`
	`} mystruct1;`
		`#pragma pack(4)`
		`struct tag`
		`{`
		`  char a;`
		`  int b;`
		`} mystruct;`
		`#pragma pack()`
		
		`struct tag1`
		`{`
		`  char a;`
		`  int b __attribute__ ((packed));`
		`} mystruct1;`

Comment:	GCC supports extension using the '__attribute__' keyword, and allows packing on individual elements of a structure.	

ALIGNMENT OF DATA

	IAR	GCC
Keyword:		`__attribute__((aligned(N)))`
Pragma:	`data_alignment`	
Syntax:	`#pragma data_alignment(8)` `int i;`	`int i __attribute__((aligned(8)));`
Comment:	The IAR Embedded Workbench® toolchain only supports a #pragma, whereas GCC supports an __attribute__	

ENDIAN SETTING OF DATA

	IAR	GCC
Keyword:	`__big_endian` `__little_endian`	
Pragma:		
Syntax:	`__big_endian int i;`	
Comment:	The GNU toolchain only supports defining the endian setting at a module level.	

NON-INITIALISED DATA

	IAR	GCC
Keyword:	`__no_init`	`__attribute__((section("no_init")))`
Pragma:		
Syntax:	`__no_init int i;`	`int i __attribute__((section("no_init")));`
Comment:	The variable will not be initialized at startup as it is placed in a separate section. Use the GNU section attribute to achieve the same thing.	

LOCATION CONTROL OF DATA

	IAR	GCC
Keyword:	`@`	`__attribute__((section("NAME")))`
Pragma:	`location`	

Syntax:	`#pragma location i=0x0100`
	`int i;`
	`__no_init int j @ 0x0104;`
		`#define i (*(int *)0x0100)`

Comment:	The GNU toolchain only supports defining the location of variables at a section level. Use the section attribute to control the link location of a variable. The #define shown would also produce the same functionality.	

If we go back to our example project and do a Rebuild we will get an error and the build process will stop. The error is because the compiler cannot find the include file "intrinsics.h". This is an IAR special header file that declares intrinsic functions that are provided by the IAR compiler.

Doing builds like this is a good way to find problems where the IAR and GCC compiler differs in language extensions and where some code modification is needed to migrate from one toll to another.

To solve the problem with IAR intrinsic functions we can try to manually find where in our code intrinsic functions are used, or we can decide to not include intrinsics.h, build again and the compiler will tell us this where intrinsic functions are being used. We go for the second option and comment out the line including intrinsics.h.

```
24  ******************************
25  //#include <intrinsics.h>
26  #include "stm32f10x.h"
```

Figure 35 - C/C++ remove <intrinsics.h>

When we now do a Build we get three warnings. One is about main() having the wrong return type and the others are about two intrinsic functions that are not declared before they are used. So we found our intrinsic functions and reading the IAR manual we see that these functions, __disable_interrupt and __enable_interrupt, modifies PRIMASK to disable and enable interrupts. Knowing CMSIS we know we have access the same functionality with __disable_irq and _enable_irq. This will also make our code more portable, using the CMSIS standard and not tool vendor specific extension. After modification our code now looks like this.

```
94  TIM_TimeBaseInitTypeDef TIM1_TimeBaseInitStruct;
95
96  //__disable_interrupt();
97  __disable_irq();
98
99  /* Setup STM32 system (clock, PLL and Flash configuration) */
100 SystemInit();
...
166 TIM_Cmd(TIM1,ENABLE);
167
168 //__enable_interrupt();
169 __enable_irq();
170
171 while(1)
172 {
173 }
```

Figure 36 - C/C++ enable/disable IRQ

We Build again and now we are left with one warning, the one saying main has the wrong return type. We can fix that by defining main as returning an int or just ignore the problem. It is not really important for our embedded application since it will not return anyway. It's going to be stuck in the while(1) loop above and only do something else when an interrupt/exception occur.

From section 2 in this guide we know that the IAR version of stm32f10x_it.c contains a definition of TIM1_UP_IRQHandler and we need to create the same behavior for our TIM1_UP_IRQHandler ISR. Having a look at the original stm32f10x_it.c we see that it only calls the function Timer1IntrHandler(), and Timer1IntrHandler is already defined in our main.c. To make it simple we can just copy the definition of TIM1_UP_IRQHandler from the original project to our stm32f10x_it.c and paste it in at the end of that C file.

Note that we add a declaration of Timer1IntrHandler before we start using it to make sure the compiler knows the return type and any parameters used by that function.

```
30  /* Includes ------------------------------------------------------------------*/
31  #include "stm32f1xx_it.h"
32
33  extern void Timer1IntrHandler (void);
34
...
166
167 /*******************************************************************************
168 * Function Name  : TIM1_UP_IRQHandler
169 * Description    : This function handles TIM1 overflow and update interrupt
170 *                  request.
171 * Input          : None
172 * Output         : None
173 * Return         : None
174 *******************************************************************************/
175 void TIM1_UP_IRQHandler(void)
176 {
177   Timer1IntrHandler();
178 }
179
180
181 /************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

Figure 37 - C/C++ Adding TIM1_UP_IRQHandler

With this we have migrated the source code over to ***Atollic TrueSTUDIO®*** and GCC and it should build without any errors. What we still have to do is to make sure that we are locating all sections at the correct locations and we will do this in later on in this guide. But before we go there let us have a look at assembler code and more importantly the startup code.

BUILT-IN FUNCTIONS

In addition to built-in functions to access processor instructions, the IAR Embedded Workbench® compiler supports access to the start address of any named section using the below functions.

```
void *   __section_begin(char const * section)
void *   __section_end  (char const * section)
size_t * __section_size (char const * section)
```

These convenience functions are not supported by GCC, though the user can simply determine the values by defining link-time symbols at the start and end of the sections of interest within the linker command file. The symbols can then be used to determine the start, end and size of any section.

As an example, if you have a global variable and you need to have access the start address, end address or size of that variable, then you can do something like this in your C source code.

```
// Let the compiler know these variables will be defined later
extern unsigned int __start_myData1__, __end_myData1__;

// Add a variable to my ".myData1" segment
__attribute__((section(".myData1"))) unsigned int myValue = 123;

// More code can go in here...

start = __start_myData1__;
end   = __end_myData1__;
size  = __end_myData1__ - __start_myData1__;
```

Figure 38 - C/C++ finding start, end and size of sections

So first we tell the compiler that __start_myData1__ and __end_myData1__ will be defined later in the build process. Second we define a variable and locate this in section ".myData1". After that is done we can start using __start_myData1__ and __end_myData1__ in the code. What remains to get this to build is to add some code to the linker script file and for the example above we would add this.

```
/* Initialized data sections goes into RAM, load LMA copy after code */
.data :
{
  . = ALIGN(4);
  _sdata = .;        /* create a global symbol at data start */
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */
  __start_myData1__ = .;
  *(.myData1)
  *(.myData1*)
  __end_myData1__ = .;

  . = ALIGN(4);
  _edata = .;        /* define a global symbol at data end */
} >RAM AT> FLASH
```

Figure 39 - Linker finding start, end and size of sections

ASSEMBLER SOURCE CHANGES

Although the underlying instructions and the addressing modes supported by them remains constant in a development tools migration project, the one non-standardized part of the toolchain is the syntax supported by the cross-assembler.

At the lowest level, cross-assemblers simply provide support for creating application code by directly specifying which instructions get executed. In order to make the programmer's task simpler, cross-assemblers offer additional 'productivity' features which include:

- Support for symbols and labels for symbolic addressing
- Support for pre-processing to allow for conditional cross-assembly and for the use of definitions and macros
- Support for external file inclusion to allow for common definitions to be shared
- Support for arithmetic expressions and strings
- Support for some high-level language features (e.g. structures)

Unfortunately, some cross-assemblers do not only differ in the 'value added' features, but also in the characters used for describing instructions (for example the symbols used to differentiate between different addressing modes, or to define numeric constants).

This kind of variance between toolchains is simply resolved by using a simple mapping scheme.

The GNU toolchain supports a directive to force the instruction syntax to be the same as defined by the ARM Instruction Set reference. The IAR toolchain is also compliant, which greatly simplifies porting. To enable the compliance add the following line to your source file:

```
.syntax unified
```

This ensures that the instructions and registers used in the source file will be consistent with any other toolchain compliant to the ARM Instruction Set reference.

The table below summarizes the key differences between the two toolchains in assembler syntax. Note that there are too many directives to list, in many cases there is a direct equivalent, but the user's manual should be consulted to check.

Area	GNU	IAR

Area	GNU	IAR
Directive naming	.<name>	<NAME>
Multiline comment	/* … */	/* … */
Single line comment	# or @	// or ;
Statement delimiter	; or newline	Newline only
Binary Constants	**0b**nnn or **0B**nnn	**b'**nnn or nnn**b**
Octal Constants	**0**nnn	**q'**nnn or nnn**q**
Decimal Constants	Nnn	nnn or **d'**nnn
Hexadecimal constants	**0x**nnn **0X**nnn	**0**nnn**h** or **0x**nnn or **h'**nnn

Table 5 - Cross-assembler differences

STARTUP CODE

On reset/power-up, startup code should be executed to initialize the runtime system (C and/or C++). This includes initialization of the stack, heap and data sections, along with the execution of global constructors for C++ applications. This functionality is automatically supported by the standard C runtime provided with every toolchain.

- A vector table should be set up to connect reset, exception and interrupt table entries to the respective handlers.
- Other processor specific initialization should be performed (e.g. clock and memory subsystem initialization).

For the majority of systems only the hardware specific code may need to be modified, there is usually a function which is called as part of the system startup code, before the main function which may be used to do things such as setup clocks, PLLs and memory controllers.

The hardware specific function (SystemInit in the case of the GCC tools for ARM® processors) is written in C, and supplied as a source module which can be modified according to need.

The interrupt vector table and interrupt handlers are also included in the startup code. For the GNU toolchain, the vector table will be populated with links to a default interrupt handler unless the user includes a function of the same name in their source code – this is possible due to the use of weak symbols.

A weak symbol (and the data or function associated with it) will be included in the final executable file unless another symbol of the same name is included in the link – when the non-weak symbol is discovered by the linker, the weak symbol is discarded. This mechanism allows default behavior to be defined in low-level code with override code existing in the user's high level source files.

The table below summarizes the functions and symbols used by the IAR Embedded Workbench® and the GNU toolchains for startup code.

The standard (CMSIS) startup sequence for Cortex-M devices is that ResetHander in cstartup_<device>.s is the starting point when the device is reset. The function ResetHandler will call SystemInit just after C/C++ has been initialized.

There are some differences between how ***Atollic TrueSTUDIO®*** and EWARM behave during startup and we recommended that you use the starup file that comes with the project created by TrueSTUDIO. ***Atollic TrueSTUDIO®*** follow the CMSIS standard and IAR might have some extensions to the startup process. IAR have a symbol called __iar_program_start that by default is the starting point for the linker and it can be the starting point for the application (reset address). IAR also have a special function called __low_level_init that is called at the very beginning of the application. Usually it returns an int that has no meaning for anyone except IAR libraries and you can in that case safely ignore that function. It is however possible to create your own version of __low_level_init and if your application has that function defined then you should bring that function

definition over to ***Atollic TrueSTUDIO®*** and call __low_level_init at the beginning of SystemInit.

Property	GNU	IAR
Startup file	startup_<device>.s (e.g. startup_stm32f10x_cl.s)	startup_<device>.s or cstartup_M.s
Entry point	ResetHandler	ResetHandler or __iar_program_start
Hardware Initialization	SystemInit	SystemInit and/or __low_level_init
Interrupt Service routines	`xxHandler` `(e.g. USART2_IRQHandler)`	`xxHandler`
Default IRQ Handler	`Default_Handler`	`Infinite loop for each xxHandler`
Top of Stack	`_estack`	`sfe(CSTACK)`
Bottom of Heap	`_end`	`sfb(HEAP)`
Start of BSS	`_sbss`	
End of BSS	`_ebss`	
Start of Init Values for Data	`_sidata`	
Start of Data	`_sdata`	
End of Data	`_edata`	

Table 6 - Startup Code symbols

Note the IAR toolchain uses special directives (**sfb,sfe**) to determine the start/end of sections.

Section 4. DETAILED PROJECT BUILD CONTROL

Having determined the source level changes required to make your application build in the desired manner with the new compiler, you may find that the default compiler settings don't generate a working system, or perhaps generate a system that doesn't match your performance or size requirements.

At this stage knowledge of the underlying tools is important, as it allows you to take the best possible advantage of the features and functionality offered.

The placement of code and data on your target are often of critical importance. Linker script files offer the ability to control the placement of all parts of your application, but are compiler toolchain specific. Therefore at some point it may become necessary to edit a linker script file.

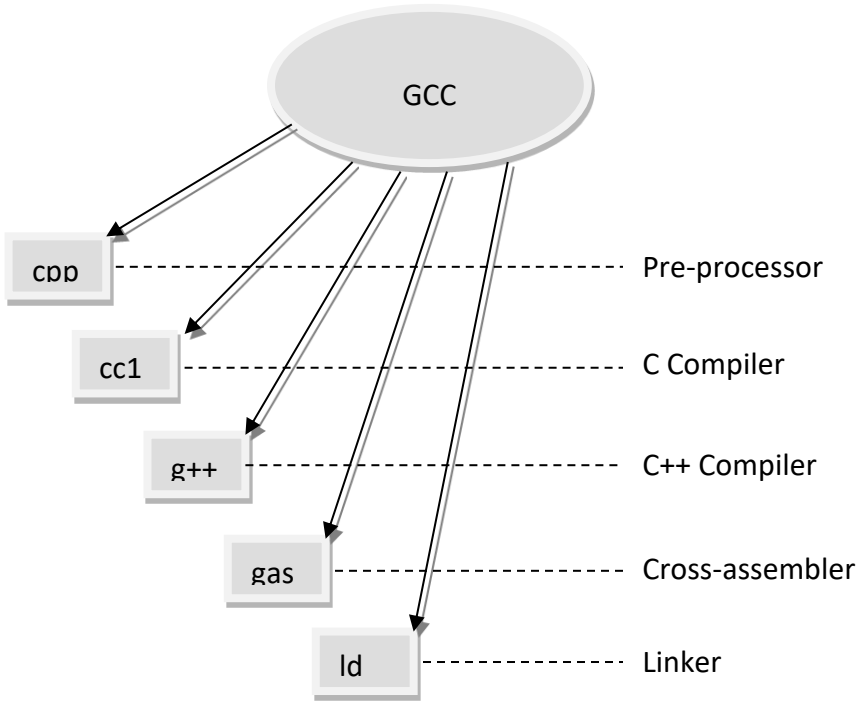
In addition, in more complicated migrations, you may need to migrate binary files, and/or libraries. This requires understanding of how libraries are constructed.

This section covers the above topics in detail, allowing the final control and customization of your migrated project.

MIGRATING BUILD FILES

The gcc program is in fact a wrapper around the underlying tools (pre-processor, C and C++ compilers and linker). This means that unless explicitly instructed to do otherwise, it will attempt to create an executable binary from the provided input files, selecting the correct compilation engine and performing linkage using a default linker control script.

In most embedded applications the user requires to have precise control over the location of code and data, so a two-stage compile and link process is required. Thus the compiler and cross-assembler are used to generate relocatable object files, and the linker is used to combine the selected relocatable object files into a single 'resolved' executable file. This is shown below.



COMPILER SETUP AND CONTROL

The GNU C and C++ compilers offer a huge range of options allowing the user to tailor the compiler according to the target architecture, the required level of optimization, the use of language extensions and/or the generation of warnings and errors.

This document does not cover every option in detail; users should refer to the GCC manual. The options discussed here are those required to perform a basic compilation in addition to those directly associated with the underlying target.

Simple compilation of a C file can be achieved using the below command line

```
gcc -c myfile.c -o myfile.o
```

This results in a relocatable object file being created (defined by using the '-c' option) from the compilation of **myfile.c** with the resulting file being called **myfile.o**.

Debugging information (Dwarf2 format) can be generated by using the '-g' switch

```
gcc -c myfile.c -g -o myfile.o
```

Section 3.9 of the GCC manual details the additional control over the generation of debugging information.

OPTIMIZATION

Optimization can be selected by using one of the 'collection' optimization switches -O1, -O2, -O3 or -Os which correspond to increasing levels of optimization for performance in the case of the numeric options, or optimization for code size in the case of -Os. The option -O0 is the default which ensures that the code generated can be debugged and minimizes compilation time.

Use -Og to further optimize debugging. -Og enables optimizations that do not interfere with debugging. It offers a reasonable level of optimization while maintaining fast compilation and a good debugging experience. In many cases, it is the best option for debugging.

It should be noted that these options represent a collection of sub-options which control individual optimization passes of the compiler.

The user may add additional sub-options to the command line to fine tune the optimization performed. The individual optimization controls are detailed in the GCC manual and are usually prefixed by '-f''.

For example '-fomit-frame-pointer' is commonly used in production code as frame pointers are useful for debugging, but may not otherwise be required.

The higher optimization levels will target performance over code size, which can result in inlining of code and loop unrolling. Where the code size increase needs to be tightly controlled additional command line switches are available to control the amount of inlining and unrolling. An example command line using such switches is given below.

```
gcc -c myfile.c -O3 -fomit-frame-pointer -finline-
limit=16 -o myfile.o
```

Details of the optimization settings available can be found in section 3.10 of the GCC manual.

IMPLEMENTATION SPECIFIC OPTIONS

There are a large number of options to control the GNU compiler toolchain, which allow for fine grained control of all aspects of the build process. Many of which are used to fine-tune optimization, or provide more information to other tools, or the user.

However particular notice should be taken of any options which enable, disable or otherwise alter the way in which the compiler generates code for source that complies to the C or C++ language standard – this 'transparent' behavior can cause problems with migration of code that either assumed a different behavior, or when linked against libraries that were built with non-compatible options.

The areas which should be double checked include:

- Size of standard types (integers, floats, wide characters)
- Default signed or unsigned for 'char'
- Size of enumerated types
- Structure packing
- Bit-field layout

The documentation for both compilers should be checked to determine any conflicts in default behavior, along with the build files for the original project to check if any specific controls have been applied to modify the default behavior.

A summary of the command line options provided by the IAR Embedded Workbench® toolchain with exact or similar GCC options is provided below. Where the GCC option shown includes an asterisk '*' this implies that more than one option is available to provide fine grained control.

IAR Option	GNU Option	GCC manual section	Comment
--aapcs	-mabi=aapcs	3.17.2	Specifies the calling convention
--aeabi	-mabi=aapcs	3.17.2	Enables AEABI-compliant code generation
--align_sp_on_irq	N/A		The Stack is aligned by default.
--arm	(default behaviour)	N/A	Sets the default function mode to ARM®
--char_is_signed	-fsigned-char	3.17.2	Treats char as signed
--cpu	-mcpu -mtune -march	3.17.2	Specifies a processor variant
--cpu_mode	--mthumb --mthumb-interwork	3.17.2	Selects the default mode for functions
-D	-D	3.11	Define preprocessor symbol
--debug	-g*	3.9	Generate debug information
--dependencies	-M*	3.11	Lists file dependencies
--diag_error	N/A	3.7	Diagnostic (compiler debug and analysis) information can be controlled using the GCC options -d*. This allows very fine grained control.
--diag_remark	N/A		
--diag_suppress	N/A		
--diag_warning	N/A		
--diagnostics_tables	N/A		
--discard_unused_publics	--discard-all --discard-locals	2.1 LD manual	Discards unused public symbols (IAR Embedded Workbench®), GCC allows control over various types of symbols to discard.
--dlib_config	N/A		Determines the library configuration file
-e	-std=gnu90 -std=gnu99 -std=gnu++98	3.4	Enables language extensions based on C90, C99 and C++98 respectively.
--ec++,--eec++	-fno-*	3.5	Disable certain features of C++
--enable_hardware_workaround	N/A		Enables a specific hardware workaround
--enable_multibytes	N/A		Enables support for multibyte characters in source files. Default behaviour of GCC.
--endian	-mlittle-endian -mbig-endian	3.17.2	Specifies the byte order of the generated code and data
--enum_is_int	-fshort-enums	3.18	Specify the size of an enumerated type
--error_limit	N/A		Specifies the allowed number of errors before compilation stops
-f	N/A		Extends the command line

IAR Option	GNU Option	GCC manual section	Comment
--fpu	-mfloat-abi -mfpu	3.17.2	Selects the type of floating-point unit
--header_context	-M*	3.11	Lists all referred source files and header files
-I	-I	3.11	Specifies include file path
--interwork	-mthumb-interwork	3.17.2	Generates interworking code
-l	(objdump) -S	Binutils manual	Creates a list file (file must have been built with debug)
--legacy	N/A		Generates object code linkable with older tool chains
--mfc	-fwhole-program	3.1	Whole program optimisation
--migration_preprocessor... _extensions	N/A		This is an inter-IAR compiler migration option and is not relevant for ***Atollic TrueSTUDIO®*** project migrations.
--misrac1998 , --misrac_	N/A		Misra is not supported by GCC
--no_clustering	N/A		Locality of access is enabled by default in GCC along with support for profile based optimisation.
--no_code_motion	-fno-sched-interblock	3.1	GCC has a number of scheduling options.
--no_const_align	N/A		
--no_cse	-fno-gcse	3.1	Disables common sub-expression elimination
--no_fragments	-fno-reorder-functions	3.1	Disables section fragment handling
--no_guard_calls	N/A		Disables guard calls for static initializers
--no_inline	-fno-inline	3.1	Disables function inlining
--no_path_in_file_macros	N/A		Removes the path from the symbols __FILE__ and __BASE_FILE__
--no_scheduling	-fno-schedule-insns	3.1	Disables the instruction scheduler
--no_tbaa	-fno-strict-aliasing	3.1	Disables type-based alias analysis
--no_typedefs_in_diagnostics	N/A		Disables the use of typedef names in diagnostics
--no_unaligned_access	-fno-align-*	3.1	Avoids unaligned accesses
--no_unroll	-fno-unroll-loops	3.1	Disables loop unrolling
--no_warnings	-w	3.8	Disables all warnings
--no_wrap_diagnostics	-fmessage-length	3.7	Disables wrapping of diagnostic messages
-O*	-O*	3.1	Sets the optimization level
-o	-o	3.2	Sets the output file name

IAR Option	GNU Option	GCC manual section	Comment
--only_stdout	N/A		Uses standard output only – this can be achieved by the shell used to invoke the compiler.
--output	-o	3.2	Sets the object filename
--predef_macros	-E -dM	3.11	Lists the predefined symbols
--preinclude	-include	3.11	Includes an include file before reading the source file
--preprocess	-E	3.11	Generates preprocessor output
--public_equ	--def-sym	2.1 LD manual	Defines a global named assembler label
-r	-g*	3.9	Generates debug information
--remarks	N/A		Enables remarks
--require_prototypes	-Wmissing-prototype	3.8	Verifies that functions are declared before they are defined
--section	--unique	2.1 LD manual	Changes a section name
--separate_cluster_for... _initialized_variables	N/A		Separates initialized and non-initialized variables
--silent	N/A		Sets silent operation
--strict_ansi	-ansi -pedantic	3.4 & 3.8	Checks for strict compliance with ISO/ANSI C
--thumb	-mthumb	3.17.2	Sets default function mode to Thumb
-- use_unix_directory_separator s	N/A		Uses / as directory separator in paths. This is the default behaviour in GCC.
--warnings_affect_exit_code	N/A		Warnings affects exit code. This is the default behaviour in GCC
--warnings_are_errors	-Werror	3.8	Warnings are treated as errors

Table 7 - Compiler option cross-reference

LINK MANAGEMENT

All modern compiler toolchains include a linker which is used to combine the output of the compiler, the cross-assembler and any libraries into an executable file. The linker therefore can have 3 types of input:

- Required application relocatable object files
- Library files used to provide library modules required by the application
- A linker control file

This section examines the linker control file usage and format, but before going into detail it's worth understanding how the linker may be invoked. As the GCC wrapper application supports automatic invocation of the correct underlying tool, it is possible to run the linker directly, or indirectly.

```
gcc myfile.o -lc -o myfile.elf
```
is equivalent to

```
ld myfile.o -lc -o myfile.elf
```

Both of the above commands will take the relocatable object file **myfile.o**, link it with the standard C library file **libc.a** and produce an executable (ELF) format file called **myfile.elf**.

The '-lc' option uses a shorthand notation where '-l' tells the linker to link a library which it will find using default search paths, the environment or other command line hints, and the 'c' suffix will be expanded to '**libc.a**' or '**libc.so**' (static and dynamic libraries respectively). Many embedded systems will not be running with a port of Linux, so only static libraries are available.

It is possible to perform compilation and linkage at the same time, using the below command line:

```
gcc myfile.c -lc -o myfile.elf
```

The only problem with using the GCC wrapper application is that there are situations where similar options are available for the compiler and the linker. Where there is a need to specify a linker option, while still using GCC, those commands may be prefixed by the -Wl switch as below

```
gcc myfile.o -lc -Wl,--entry=ResetHandler -o myfile.elf
```

A separate manual is provided which details how the linker is used, called **ld.pdf** in the Atollic distribution.

LINKER SCRIPT/COMMAND FILES

Linker script files are used to control what gets linked, where it gets placed in memory and to define any symbols which may be used to initialize the system on startup.

Most modern linkers provide similar functionality to one-another, so in migration, the main task is to understand how to convert the syntax of one linker command format to that of the other.

Some projects never require changing the default behavior of the linker away from that either provided by the compiler toolchain, or more likely by the IDE which created the project.

When starting a migration it is strongly recommended to use the ***Atollic TrueSTUDIO®*** IDE to create a project along with the associated build files which may then be tailored according to need.

The wizard provided by the IDE will allow the user to select the target processor and even development board, which will greatly simplify the process. It should also be remembered that all embedded systems have the same basic requirements:

- On reset/power-up, startup code should be executed to initialize the runtime system (C and or C++), this includes initialization of the stack, heap and data sections, along with the execution of global constructors for C++ applications. This functionality is automatically supported by the standard C runtime provided with every toolchain.
- A vector table should be set up to connect reset, exception and interrupt table entries to the respective handlers.
- Other processor specific initialization should be performed (e.g. clock and memory subsystem initialization).

The GNU linker LD supports a high-level command syntax to enable placement of code and data. An extract of example file is provided below along with explanatory comments to highlight the main features.

Note that the '.' directive defines the current location pointer.

```
ENTRY(Reset_Handler)
```

Define the entry point of the application as 'ResetHandler'

```
_estack = 0x20010000;
_Min_Heap_Size = 0;
_Min_Stack_Size = 0x200;
```

Define symbols with constant values

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH =
256K

  RAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 64K

  MEMORY_B1 (rx) : ORIGIN = 0x60000000, LENGTH =
0K
}
```

Define memory regions:

- 'FLASH' with read / execute permissions, size 256kB, start address 0x8000000
- 'RAM' with read / write / execute permissions, size 64kB, start address 0x20000000
- 'MEMORY_B1' with read/execute permissions, size 0kB, start address 0x60000000

```
SECTIONS
{
```

Start allocation of sections to memory

```
 .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector))
    . = ALIGN(4);
  } >FLASH
```

Allocate the '.isr_vector' section to FLASH. 4-byte alignment is maintained to ensure correct placement.

```
 .sidata = .;
```

Define the symbol '.**sidata**' to be at the current location in memory (just after the vector table).

```
 .data : AT ( _sidata )
  {
    . = ALIGN(4);
    _sdata = .;
    *(.data)
    *(.data*)
    . = ALIGN(4);
    _edata = .;
  } >RAM
```

Using the'AT' directive, place the **contents** of the .data* sections at the addres specified by .sidata, but link **references** to the data as if it exists in the RAM section. This allows for initial values to be placed in non-volatilve memory, and copied at startup into the runtime memory image. Symbols **_sdata** and **_edata** are defined for the start and end addresses of the collection of '.data' sections. Thus initialisation copies from .**sidata** to **_sdata.**

```
  PROVIDE ( end = _ebss );
  PROVIDE ( _end = _ebss );
```

Provide the symbols **end** and **_end** only if they are referenced, but not defined anywhere in the included files (to ensure a linker symbol clash doesn't occur)

```
 ._user_heap_stack :
  {
    . = ALIGN(4);
    . = . + _Min_Heap_Size;
    . = . + _Min_Stack_Size;
    . = ALIGN(4);
  } >RAM
```

Allocate an area in the RAM memory which is at least as big as the minimum heap and stack sizes defined earlier in the file.

```
}
```

Complete the section allocation

If we have a look at the IAR linker script file from our example project we will see something like the code below. Here we have highlighted different parts of the linker script file and we will have a look at how the corresponding GCC linker script file would do the same thing.

```
1  /*###ICF### Section handled by ICF editor, don't touch! ****/
2  /*-Editor annotation file-*/
3  /* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
4  /*-Specials-*/
5  define symbol __ICFEDIT_intvec_start__ = 0x08000000;
6  /*-Memory Regions-*/
7  define symbol __ICFEDIT_region_ROM_start__   = 0x080000EC ;
8  define symbol __ICFEDIT_region_ROM_end__     = 0x0801FFFF;
9  define symbol __ICFEDIT_region_RAM_start__   = 0x20000000;
10 define symbol __ICFEDIT_region_RAM_end__     = 0x20004FFF;
11 /*-Sizes-*/
12 define symbol __ICFEDIT_size_cstack__   = 0x800;
13 define symbol __ICFEDIT_size_heap__     = 0x400;
14 /**** End of ICF editor section. ###ICF###*/
15
16
17 define memory mem with size = 4G;
18 define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
19 define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];
20
21 define symbol __region_USB_PKG_RAM_start__   = 0x40006000;
22 define symbol __region_USB_PKG_RAM_end__     = 0x400063FF;
23 define region USB_PKG_RAM_region = mem:[from __region_USB_PKG_RAM_start__ to __region_USB_PKG_RAM_end__];
24
25
26
27 define block CSTACK    with alignment = 4, size = __ICFEDIT_size_cstack__   { };
28 define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };
29
30 initialize by copy { readwrite };
31 do not initialize  { section .noinit };
32 do not initialize  { section USB_PACKET_MEMORY };
33
34 place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };
35 place in USB_PKG_RAM_region
36                      { readwrite data section USB_PACKET_MEMORY  };
37 place in ROM_region   { readonly };
38 place in RAM_region   { readwrite,
39                         block CSTACK, block HEAP };
```

Figure 40 - EWARM linker script file

Line 1 to 14 is defining symbols that later are used in the script. In our GCC script we would define symbols the same way we define variables in C, like this.

```
36 /* Highest address of the user mode stack */
37 _estack = 0x20005000;    /* end of 20K RAM */
38
39 /* Generate a link error if heap and stack don't fit into RAM */
40 _Min_Heap_Size = 0;      /* required amount of heap  */
41 _Min_Stack_Size = 0x100; /* required amount of stack */
```

Figure 41 - Linker script, defining symbols

Line 17 to 23 defines the memory space and the different regions inside this memory. The corresponding way to do this in GCC is to use the MEMORY command and define the different regions inside this MEMORY command. Our example project defines our memory and regions like this.

```
43 /* Specify the memory areas */
44 MEMORY
45 {
46   FLASH (rx)       : ORIGIN = 0x08000000, LENGTH = 128K
47   RAM (xrw)        : ORIGIN = 0x20000000, LENGTH = 20K
48   MEMORY_B1 (rx)   : ORIGIN = 0x60000000, LENGTH = 0K
49 }
```

Figure 42 - Linker script, defining memory and regions

Line 27 to 28 is defines the stack and heap blocks. In our GCC linker script we would do the same thing inside the SECTIONS command and the screenshot below shows an example of this.

```
147  /* User_heap_stack section, used to check that there is enough RAM left */
148  ._user_heap_stack :
149  {
150    . = ALIGN(4);
151    PROVIDE ( end = . );
152    PROVIDE ( _end = . );
153    . = . + _Min_Heap_Size;
154    . = . + _Min_Stack_Size;
155    . = ALIGN(4);
156  } >RAM
```

Figure 43 - Linker script, adding stack and heap

Here we align the stack and heap to 4, provide the symbols “end” and “_end” as addresses to the end of the heap block and then locates the heap and stack after each other. The “>RAM” makes sure that all of this is located in the RAM memory region.

Line 30 to 32 is specifying what to initialize or not. We do the same thing in our script file when we locate our sections inside the SECTIONS command. In the example above for the stack and heap sections we used “>RAM” at the end to specify that this is none initialized sections. A section that needs to be initialized would in our example use “RAM AT> FLASH” to tell the linker to store this section at FLASH and copy the content to RAM at startup. Below is an example of this.

```
119  /* Initialized data sections goes into RAM, load LMA copy after code */
120  .data :
121  {
122    . = ALIGN(4);
123    _sdata = .;        /* create a global symbol at data start */
124    *(.data)           /* .data sections */
125    *(.data*)          /* .data* sections */
126
127    . = ALIGN(4);
128    _edata = .;        /* define a global symbol at data end */
129  } >RAM AT> FLASH
```

Figure 44 - Linker script, initialized data

Line 34 to 39 is locating the different sections into memory regions. This is what we already have done inside the SECTIONS command for the stack, heap and initialized data. Our GCC linker script contains many more commands inside the SECTION command in order to make sure that all sections in our example application gets located correctly.

Line 34 in the IAR linker script file contains a “place at’ directive in order to make sure that the interrupt vector table is located at the beginning of our Flash memory. In our GCC script we can accomplish the same thing if we, in the SECTIONS command, start be locating our vector table in the FLASH memory region. Anything else that is located in FLASH will be located after our vector table.

We see that we are missing the USB_PKG_RAM_region and that we have a memory region call MEMORY_B1that we really don't need for our application after we compare the two linker script files. We can also see in the IAR linker script that USB_PKG_RAM_region should not be initialized. The rest is behaving basically the same and we can keep the ***Atollic TrueSTUDIO®*** linker script file as it is for those other parts.

To create a linker script file that will, in all essential parts, locate our application and initialize memory in the same way as the original project we need to do the following

Replace the memory region MEMORY_B1 with USB_PKG_RAM_region.

Remove the code that locates sections into MEMORY_B1.

Add section placement for the sections that goes into USB_PKG_RAM_region, making sure that this section should not be initialized.

Comparing the memory start addresses and sizes with IAR we see that ROM/FLASH and RAM matches up, but we need to modify the start address and size for our USB memory. We modify the MEMORY command in our linker script file to this.

```
40 /* Specify the memory areas */
41 MEMORY
42 {
43   FLASH (rx)                : ORIGIN = 0x08000000, LENGTH = 128K
44   RAM (xrw)                 : ORIGIN = 0x20000000, LENGTH = 20K
45   USB_PKG_RAM_region (rwx)  : ORIGIN = 0x40006000, LENGTH = 1K
46 }
```

Figure 45 - Linker script, modifying memory regions

We also remove the code locating code and data into MEMORY_B1 to avoid getting a warning from the linker saying none declared memory region.

```
172
173   /* MEMORY_bank1 section, code must be located here explicitly             */
174   /* Example: extern int foo(void) __attribute__ ((section (".mb1text"))); */
175   .memory_b1_text :
176   {
177     *(.mb1text)         /* .mb1text sections (code) */
178     *(.mb1text*)        /* .mb1text* sections (code) */
179     *(.mb1rodata)       /* read-only data (constants) */
180     *(.mb1rodata*)
181   } >MEMORY_B1
182
```

Figure 46 - Linker script, removing section placements

After this code is removed we add code to locate our USB_PACKET_MEMORY into uninitialized USB_PKG_RAM_region memory region. We can do this with the code below.

```
  /* Uninitialized data section */
  . = ALIGN(4);
  .usb_packet :
  {
    *(USB_PACKET_MEMORY)
    *(USB_PACKET_MEMORY*)
  } >USB_PKG_RAM_region

  . = ALIGN(4);
  .bss :
  {
    /* This is used by the startup in order to initialize the .bss secion */
    _sbss = .;         /* define a global symbol at bss start */
    __bss_start__ = _sbss;
    *(.bss)
    *(.bss*)
    *(COMMON)
```

Figure 47 - Linker script, place sections in regions

In order to locate variables into the USB_PACKET_MEMORY we can use the __attribute keyword “__attribute((section("USB_PACKET_MEMORY")))”.

LIBRARY MANAGEMENT

Libraries are collections of relocatable object files that are commonly used by applications. The collection, or archive, may be used by the linker to resolve (i.e. find a match for) a function or data item which has been referenced by the user's application code.

Every compiler ships with a number of libraries such as the standard C library, the C mathematics library, the C++ runtime library etc. Such 'system' libraries usually include default startup code and intrinsic libraries which implement the low-level functionality required to make an application work.

Startup code ensures that the processor is initialized correctly on reset and that the runtime environment is set up correctly (global variables get their correct values, the heap is initialized etc).

Intrinsic libraries implement low level operations which the compiler requires to support high level language types and features, which are not easily implemented using one or a small number of assembler instructions. Examples of intrinsic functions may include 64-bit arithmetic on 32-bit machines, floating point operations on machines with no floating point unit and function prolog/epilog code used when optimizing for size.

Each relocatable module contained within a library will only be included in the generated executable file if one or more of the symbols it exports (which correspond to functions or data items) is referenced, either directly from application code, or indirectly from another library which itself was directly referenced. This enables the generated executable file to only include the required functions and data, rather than always including the complete C or C++ runtime.

There are two types of library, static and dynamic. The library code contained within static libraries will form part of the executable (if referenced) – i.e. all references are resolved once ('statically') at build time.

As most embedded systems are single applications, this is the normal model. Dynamic libraries allow the library code to be linked to the application at runtime, which leads to smaller application binaries, but requires a runtime linker/loader such as that provided by Linux. This document covers static libraries only, as systems migrating from the IAR Embedded Workbench® toolchain will not be Linux based.

STANDARD LIBRARIES

The C and C++ standards don't just define the language, but also specify a set of runtime libraries that need to be supported.

The 'standard libraries provide facilities to manipulate data at a level not supported directly by the language (e.g. strings), to manage memory allocation dynamically (malloc and free for C, or new and delete for C++), and to interact with the underlying system (file input/output, time management etc).

In addition to the standard libraries, compilers typically ship with what are called 'intrinsic' libraries.

These are compiler specific and usually provide low level routines which the compiler invokes automatically as part of the build process. Intrinsic functions are usually written to implement a language feature which the underlying processor cannot support simply.

For example 64-bit arithmetic on a 16 or 32-bit CPU, or floating-point arithmetic on any CPU without a dedicated floating-point unit. Programmers don't call intrinsic libraries directly (they don't need to) and there is no guarantee that the functions comply to the normal ABI.

The intrinsic functions are usually hand optimized, written in assembler, and undocumented. The programmer simply needs to ensure that the intrinsic, or 'language support' library is included in the build to enable the correct low-level functions to be included in the final application.

A further type of library file usually shipped with a compiler is used to support systems where 'objects' need initialization at startup. This includes calling constructors for global objects in C++ applications; along with the corresponding destructors should the application ever exit. Special initialization and finalization sections are included in the executable file which contains code to iterate through a list of supplied constructors/destructors.

In order to aid migration, the below table provides an insight into the libraries shipped with the GNU C/C++ compiler:

Library	Language	Usage
Libc.a	C	This is the standard C runtime library (not including maths)
Libg.a	C	This is a debug build of the standard C runtime library
Libm.a	C	This is the C standard mathematic library
Libgcc.a	C	This is the intrinsic 'compiler support' library required to support C applications
Libiberty.a	C	a collection of subroutines used by various GNU programs including getopt, obstack, strerror, strtol and strtoul.
Libgcov.a	C	GCC supports automatic instrumentation of code, by including gcov, it is possible to analyze programs to help create more efficient, faster running code through optimization
Libsupc++.a	C++	This library provides support for the C++ programming language (among other things, libsupc++ contains routines for exception handling). This library can be used where the full C++ standard library is not required.
Libstdc++.a	C++	The C++ standard library. It is used by C++ programs and contains functions that are frequently used in C++ programs. This includes the Standard Template Library (STL).
crtbegin.o & crtend.o	C++	Constructors and destructor support files

Table 8 - Standard Libraries

Library Creation and Management

Static libraries are simply a collection of compiled source modules, they can be created and managed using a standard 'archiving' tool, in the case of the GNU C/C++ compiler, and this tool is called 'ar'.

To create an archive from one or more relocatable object files, use the command line below, which will create a new archive called **libmyfiles.a**, or update it if it already exists, replacing old versions of modules **file1.o** and **file2.o** if they exist.

```
ar cru libmyfiles.a file1.o file2.o
```

By convention static libraries have the '.a' extension.

The IAR Embedded Workbench® library management tool (iarchive) is similar in concept and capability to the GNU one, which makes migration of library build files relatively simple.

In the case where legacy IAR Embedded Workbench® binaries are to be used in the migrated application, it is useful to extract those modules from the libraries they currently exist in, in order to create a 'migration library' which may also include any ABI wrapper code required.

It is not recommended to simply link against original IAR Embedded Workbench® libraries, as they may (will) include modules which define functions or data items which are also defined in the GNU libraries – this will create link time errors due to duplicate symbols, or in the worst case override weak symbols in the GNU libraries resulting in a successfully linked application which has unexpected behavior.

Unless a specific third party library is used, in all other cases the recommended way to solve library dependencies is as below.

1. Link the application against the GNU libraries only, noting any 'unresolved external' linker errors.
2. Determine the source of each of the symbols in the legacy libraries. This can be done using a combination of the library management tool and an object file utility as shown in the following example.
3. Extract only the source modules that contain the required symbols from the legacy libraries creating/updating a migration library.
4. Repeat from step 1 above linking against the updated migration library until all unresolved externals are resolved.

Example commands to determine the location of a symbol in a library, to extract the containing relocatable module, and to include it into a migration library are shown below.

For iarchive to find and extract the relocatable file (module)

```
iarchive --symbols mylib.a          list the symbols in library
iarchive -x mylib.a module.o        extract module.o from the library
```

For GNU to create/update the migration library

```
ar cru libmigration.a module.o
```

MIGRATING 3RD PARTY FILES

If your existing project contains third party libraries, which you wish, or need, to include in the migration project, then the scope of the migration needs to be enlarged to encompass the effort to achieve this.

As the GNU Compiler toolchain port to the ARM® architecture is ubiquitous, it is extremely likely that a port has already been done to GCC. Depending on the way that the library is supplied, you may already have a port!

VENDOR SUPPLIED PORTS

Some vendors supply libraries in source form with build/configuration files. In such cases, it is likely that the source files already support GCC compilation through the use of conditional compilation. The documentation supplied with the library should provide information on building with the GNU tools.

Often the libraries are supplied in a binary format, in order to protect the intellectual property of the vendor. In such cases it is likely that a binary distribution for the GNU compiler toolchain is available, though it will be necessary to check licensing arrangements with the vendor.

SOURCE LEVEL PORTING

The method to port third party source libraries to GCC is essentially the same as for the rest of the project. You may however need to check on the terms of the license to check whether porting the source files to another toolchain is permitted.

In order to make the ported library available to other projects, it is worth creating a new project in the ***Atollic TrueSTUDIO®*** workspace to contain it. The workspace environment supports multiple projects, each of which may be an application or a library. In addition, dependencies between projects may be defined to ensure that the build order is correct (i.e. build the libraries first), and that if the library is changed, then the main application will be rebuilt too.

BINARY LEVEL PORTING

From a technical perspective this is the hardest of the porting exercises to achieve. The assumption here is that there is a binary library with associated header files, and the user is unable to convince the vendor to supply a version of the library which has been ported to the GNU compiler toolchain.

The requirement is to be able to call all of the library functions from your GCC based application, passing data into, and receiving data from them.

As detailed previously in this document, both the GNU and IAR compilers support the ARM procedure call standard, and have consistent object file formats, which may enable the library functions to be directly linked into your end application. It is worth checking this route first, before embarking on generating ABI wrappers.

The steps to follow for simple integration are:

1. Use the IAR Embedded Workbench® archive tool to extract all modules from the IAR Embedded Workbench® archive
2. Use the GNU archive tool to create a new archive, importing all of the library modules
3. Add the new library as file to the main application project

An example of this process is shown below:

```
iarchive -x tplib.a module.o      extract module.o from the library
ar -cru libtplib.a *.o            create a new library with all the modules
```

CREATING A BINARY INTERFACE

In a small number of cases it may be necessary to construct an interface library to the third party library. For C libraries, the process is relatively straight forward. For C++ libraries, the process may be much more complex; depending on the nature of the library it may be prudent to create a C wrapper around the underlying C++ library, effectively masking the differences in the ABI between compilers.

In some situations this may not be acceptable, in which case C++ wrappers will need to be constructed – this requires a high level of understanding of the low level implementation of C++ (how objects are constructed in memory, how polymorphism is supported using virtual function pointer tables, how exceptions are thrown and caught), and is beyond the scope of this document.

Here follows a series of methods which the user can employ to perform the majority of migration tasks.

FUNCTION CALL/RETURN

Before going into the methods used to construct binary interfaces, the first thing that needs to be understood is the function call/return mechanism. This defines the registers that get used in passing parameters and returning results when calling functions, in addition to rules on which registers may be modified by a called function without needing to be preserved (i.e. saved on entry and restored on exit).

The input parameters are stored first in registers, and then on the stack.

Item	Description
Input Parameters	r0 to r3 r0:r1 and r2:r3 pairs may be used for 64-bit parameters. The first parameter may be an address: • the 'this' pointer for C++ • the address of a large/composite return value
Return Value	r0, or r0:r1 for 64-bit values r0 may contain the address of the return data for large/composite data
Work registers	r0 to r3 and r12 may be modified by the called function without saving
Saved registers	r4 to r11 must be saved if modified and restored before returning
Special registers	r13 is the stack pointer r14 is the link register r15 is the program counter

Thus on return from a function, registers r4 to r11 must contain the same values as when it was called, as should the stack pointer and link register. Register r0 contains the return value. Registers r1 to r3 and r12 may contain any data.

USE THE COMPILER TO CREATE AN INTERFACE FOR YOU

The 'outgoing' compiler may be used to construct a call/return interface for use by the new compiler by simply using its ability to emit assembler source files from C/C++ level input. This relies on using the legacy toolchain to create a valid assembler interface, callable from C or C++ which may then be integrated with the new tools.

It should be noted that this procedure assumes that the two C/C++ compilers produce code with incompatible interfaces, if this is not the case, then please refer to previous sections in this manual for integrating compatible libraries into your new project.

As an example, assume the third party library has a function 'foo' with the below prototype.

```
typedef struct s
{
    int a;
    char *str;
} S;

S foo (int a, char *str);
```

The function takes two parameters, one of which is a pointer, and returns a structure. It is assumed that for whatever reason, the structure layout varies between the two compilers, which present the worst case scenario to deal with.

First of all, create a wrapper function in C as a basis for the interface and save in a file called `wrapper.c`.

```
S foo_w (int a, char *str)
{
    S s = foo (a,str);
    return (s);
}
```

The source file should be compiled twice, using both the IAR and GNU compilers, in both cases using the compiler's facility to emit assembler source files, and in both cases with optimization disabled. It is important to remove optimization, as otherwise the compiler may produce code which is extremely hard to adapt.

The aim is to produce function prolog/epilog code suitable for GCC, and a function body suitable for calling the underlying IAR Embedded Workbench® function.

To generate two assembler files using the two compilers:

```
gcc -S wrapper.c -o wrapper_g.s -O0 -fomit-frame-pointer

iccarm  wrapper.c -lb wrapper_i.s -On --interwork
```

This will cause files `wrapper_g.s` and `wrapper_i.s` to be generated, both of which have compiled the input code and generated an assembler source file.

Looking at the two files, it is possible to see the parts of the code responsible for managing the function 'prolog' and 'epilog' – i.e. setting up the stack frame, managing incoming parameters and setting the return value, plus the parts of the code responsible for calling the library function 'foo'.

As defined earlier, the procedure call standard states that if the return value is a composite type (structure) which is larger than 4 bytes, then an extra argument is used to specify the location the return value is to be stored. Thus r0 is used to pass in the address of the memory to contain the return value. In the case of the IAR generated code, the memory is put at the top of the stack, whereas for the GNU generated code, the memory is located at an 8-byte offset into the stack.

The table below shows how the generated code can be partitioned and merged to form a new function. In this simple example, as both compilers conform to the procedure call standard, there was no need to create the wrapper file, but it enables the mechanism to be shown. The prolog and epilog code is shown in blue, the function call code in green, and the return value processing in red.

IAR

```
foo_w:
        PUSH    {R2-R6,LR}
        MOVS    R4,R0
        MOVS    R5,R1
        MOVS    R6,R2
        MOVS    R2,R6
        MOVS    R1,R5
        MOVS    R0,SP
        BL      foo
        MOVS    R0,SP
        LDR     R2,[R0, #+0]
        LDR     R3,[R0, #+4]
        STM     R4,{R2,R3}
        POP     {R1,R2,R4-R6,LR}
        BX      LR
```

GNU

```
foo_w:
        stmfd   sp!, {r4, lr}
        sub     sp, sp, #16
        mov     r4, r0
        str     r1, [sp, #4]
        str     r2, [sp, #0]
        add     r3, sp, #8
        mov     r0, r3
        ldr     r1, [sp, #4]
        ldr     r2, [sp, #0]
        bl      foo
        mov     r3, r4
        add     r2, sp, #8
        ldmia   r2, {r0, r1}
        stmia   r3, {r0, r1}
        mov     r0, r4
        add     sp, sp, #16
        ldmfd   sp!, {r4, lr}
        bx      lr
```

Merged

```
foo_w:
        stmfd   sp!, {r4, lr}
        sub     sp, sp, #16
        mov     r4, r0
        str     r1, [sp, #4]
        str     r2, [sp, #0]
        add     r3, sp, #8
        MOVS    R2,R2 ;not needed
        MOVS    R1,R1 ;not needed
        MOVS    R0,R3
        BL      foo
        mov     r3, r4
        add     r2, sp, #8
        ldmia   r2, {r0, r1}
        stmia   r3, {r0, r1}
        mov     r0, r4
        add     sp, sp, #16
        ldmfd   sp!, {r4, lr}
        bx      lr
```

Once the prolog and epilog code has been identified, and as the resulting function is to be called by code generated by GCC, the prolog, epilog and return value processing code is copied from the GNU generated code. The function call code is then copied from the IAR Embedded Workbench® generated code, which ensures that the function which is to be called has the parameters set in the correct manner.

Note that if the location of the input data varied between compilers, this section of code would vary between the two assembler files. In such cases, the code highlighted in green would need to map the incoming data from the prolog to the format required for the underlying function.

However the function call code has to be modified to reflect the locations where the incoming values were stored by the GNU compiler. In fact, by inspection, it can be seen that the input values remain in registers r1 and r2, so the modified IAR Embedded Workbench® code as shown is really not required. Finally, the r0 parameter should point

to the memory which has been set aside by the GNU compiler for the return value, so the merged code is modified to use r3 rather than the stack pointer.

Having completed the merged file, it may be included as an assembler source file into the overall project and built accordingly.