

IAR EmbeddedWorkbench®
to
Atollic TrueSTUDIO®
migration guide

COPYRIGHT NOTICE

© Copyright 2010-2011 Atollic AB. All rights reserved. No part of this document may be reproduced or distributed without the prior written consent of Atollic AB. The software product described in this document is furnished under a license and may only be used or copied according to the terms of such a license.

TRADEMARK

Atollic, TrueSTUDIO, TrueINSPECTOR, TrueANALYZER, TrueVERIFIER and the Atollic logotype are trademarks or registered trademarks owned by Atollic. ARM, ARM7, ARM9 and Cortex are trademarks or registered trademarks of ARM Limited. ECLIPSE™ is a registered trademark of the Eclipse foundation. Microsoft, Windows, Word, Excel and PowerPoint are registered trademarks of Microsoft Corporation. Adobe and Acrobat are registered trademarks of Adobe Systems Incorporated. IAR Systems and IAR EmbeddedWorkbench are registered trademarks of IAR Systems. All other product names are trademarks or registered trademarks of their respective owners.

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

ASW-IAMG January 2011

REVISION

First version October 2010
Second version January 2011
Third version September 2011

Atollic AB

Science Park
Gjuterigatan 7
SE- 553 18 Jönköping
Sweden

+46 (0) 36 19 60 50

E-mail: sales@atollic.com

Web: www.atollic.com

Atollic Inc.

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

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

E-mail: sales.usa@atollic.com

Web: www.atollic.com

Contents

Introduction.....	1
Who Should Read This Guide	1
Document Conventions	1
Typographic Conventions	2
SECTION 1: Migration Decisions	1
Why migrate?	2
When to Migrate?	3
What to migrate and the implications of migration?.....	4
Project and build control.....	4
Infrastructure and work-flow	4
Application source and firmware.....	4
Third party O/S and libraries	5
Re-validation	5
How can migration be made easier?.....	7
Automated Project creation.....	7
CMSIS - Cortex® Microcontroller Software Interface Standard	7
Migration of Legacy Firmware	8
ABI Compliance	8
SECTION 2: Starting the Migration with Atollic TrueSTUDIO®. 10	
Before you start.....	11
Workspaces & projects	11
Creating a new project	13
Configuring the project	17
Building the project	21
Build	21
Rebuild all.....	21
Importing Source Files.....	23

Using files in an external location	23
Using directories in an external location	24
SECTION 3: Migrating Source Files	27
C/C++ Source changes	28
The Pre-processor	28
Language extensions	30
Inline assembler	31
Inline functions	31
RAM based functions	31
Interrupt and exception functions	32
Nested interrupt functions	32
Non-returning functions	32
ARM® specific functions	33
Weak functions/data	33
Root functions and unreferenced data	33
Packed Data	34
Alignment of data	34
Endian setting of data	34
Non-initialised data	35
Location control of data	35
Built-in functions	35
Assembler source changes	36
Startup code	38
SECTION 4: Detailed Project Build Control	40
Migrating Build files	41
Compiler setup and control	42
Optimization	42
Implementation Specific Options	43
Link management	48

Linker Script/Command Files	48
Library management	51
Standard Libraries	51
Library Creation and Management	53
Migrating 3rd Party files	55
Vendor supplied ports.....	55
Source level porting	55
Binary level porting	55
Creating a binary interface.....	56
Function call/return	56
Use the compiler to create an interface for you	57

Figures

No table of figures entries found.

Tables

Table 1 - IAR EmbeddedWorkbench® Specific Predefined Symbols.....	29
Table 2 - Cross-assembler differences	37
Table 3 - Startup Code symbols.....	39
Table 4 - Compiler option cross-reference	47
Table 5 - Standard Libraries	53

INTRODUCTION

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

WHO SHOULD READ THIS GUIDE

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 EmbeddedWorkbench® 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

TYPOGRAPHIC CONVENTIONS

This document has the following typographic conventions:





Style	Use
<code>Computer</code>	Keyboard commands or Source code.
Bold	Names of menus, menu commands, buttons and dialog boxes that appear on screen.
<i>italic</i>	A cross reference in this user guide or to another guide.
Atollic TrueStudio®	Atollic company products.
	Identifies instructions specific to the graphical user interface.
	Identifies instructions specific to the command line interface.
	Identifies help tips and programming hints.
	Identifies a caution.

Table 1 – Typographical 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 EmbeddedWorkbench® 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 EmbeddedWorkbench® 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 EmbeddedWorkbench® 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 EmbeddedWorkbench® 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 EmbeddedWorkbench® 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.

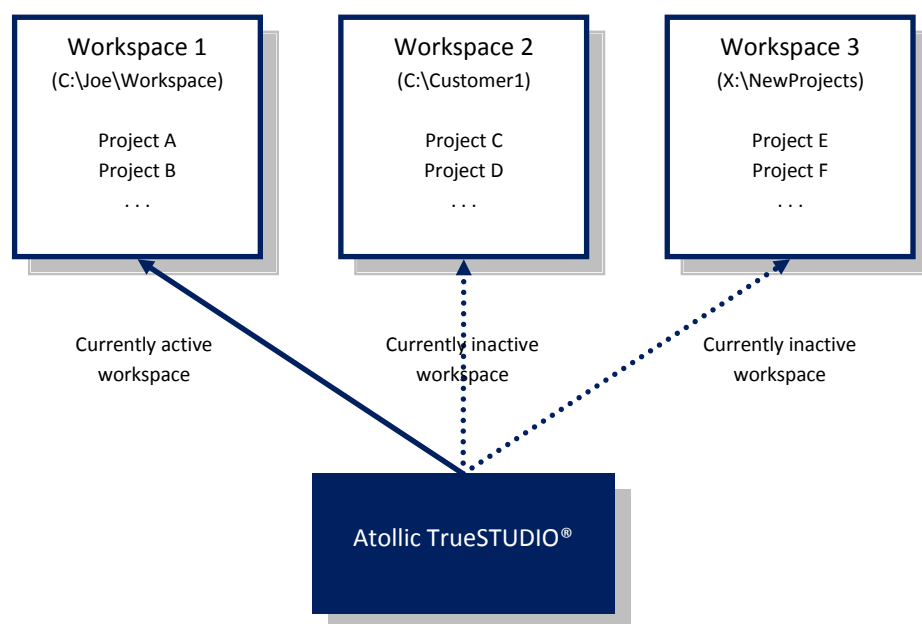


Figure 1 - Workspaces and projects

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.

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

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

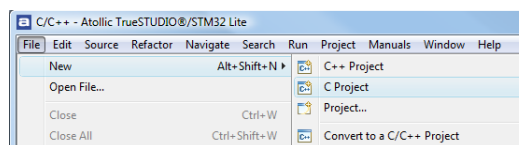


Figure 2 - Starting the project wizard

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

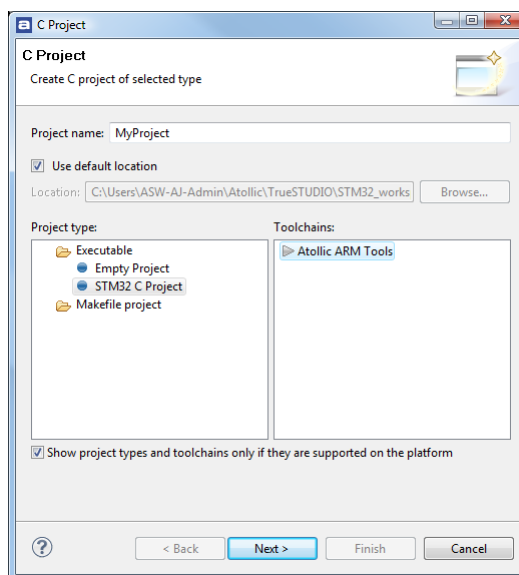


Figure 3 - C project

3. Enter a **Project name** (such as “MyProject”) and select your embedded target as the **Project type**, and **Atollic ARM Tools** as the **Toolchain**. Then click the **Next** button to display the **TrueSTUDIO® Build Settings** page.

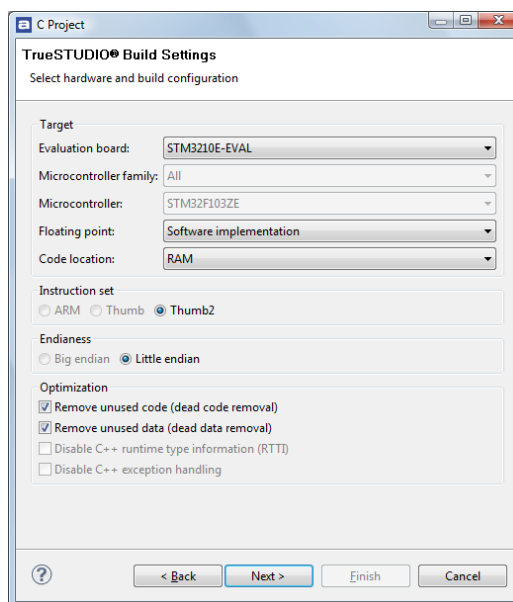


Figure 4 - TrueSTUDIO® Build Settings

4. In the **TrueSTUDIO® Build Settings** page, configure the hardware settings according to your Evaluation board or custom board design. Please note that your evaluation board may have hardware switches for configuration of RAM or FLASH mode. This setting must be the same in both the project wizard and on the board. Finally, click the **Next** button to display the **TrueSTUDIO® Debug Settings** page.

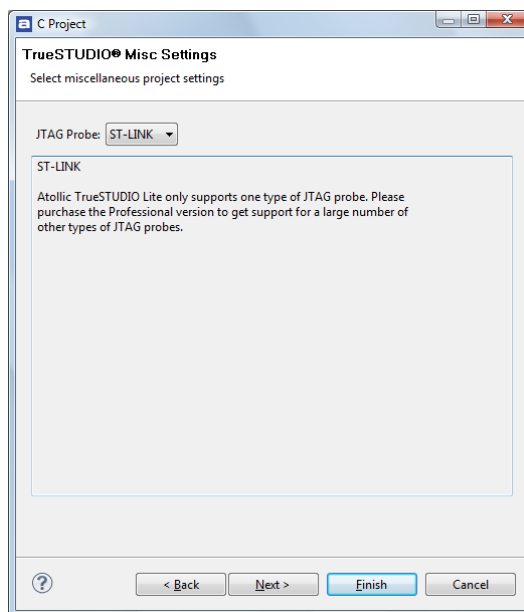


Figure 5 - TrueSTUDIO® Debug Settings

5. Select the JTAG probe you use. Click the **Next** button to display the **Select Configurations** page.

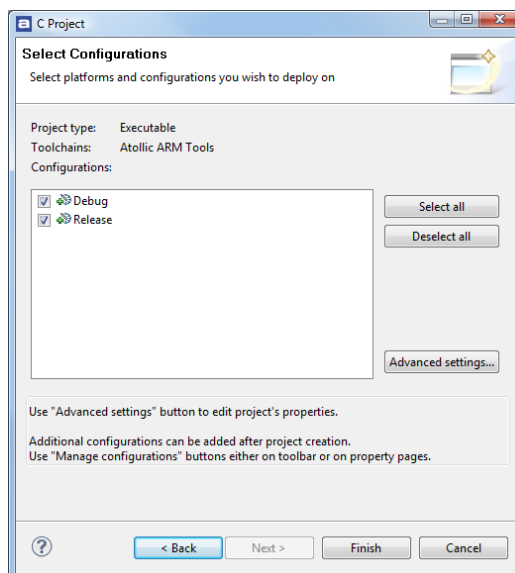


Figure 6 - Select Configurations

6. In the **Select Configurations** page, click on the **Finish** button to generate a new C project.
7. A new managed mode C project is now created. **Atollic TrueSTUDIO®** generates target specific sample files in the project folder to simplify development.
8. Expand the project folder (such as “MyProject” in the example above) and the **src** folder in the **Project Explorer** docking view.

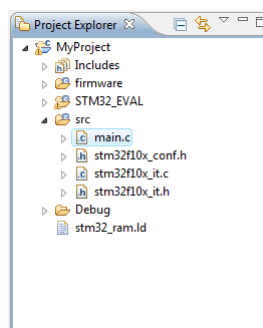


Figure 7 - Project Explorer view

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. Select the **Project, Properties** menu command.

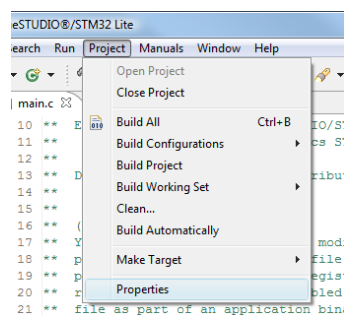


Figure 8 - Project properties menu command

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

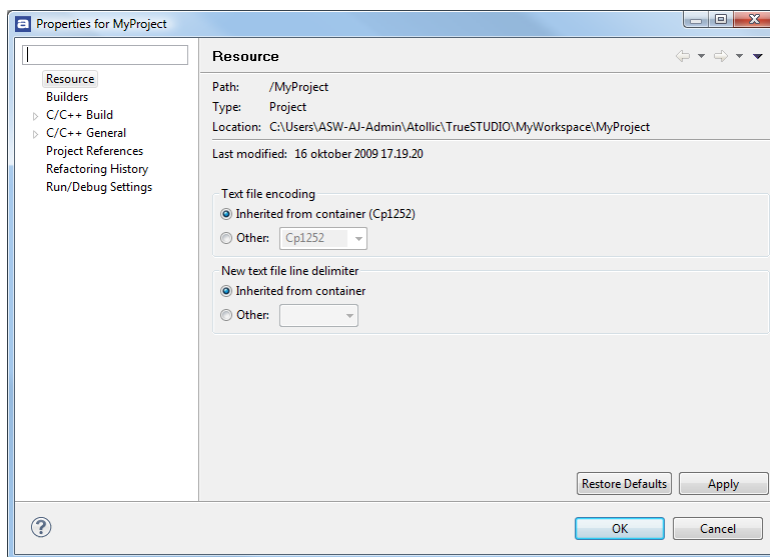


Figure 9 - Project properties dialog box

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

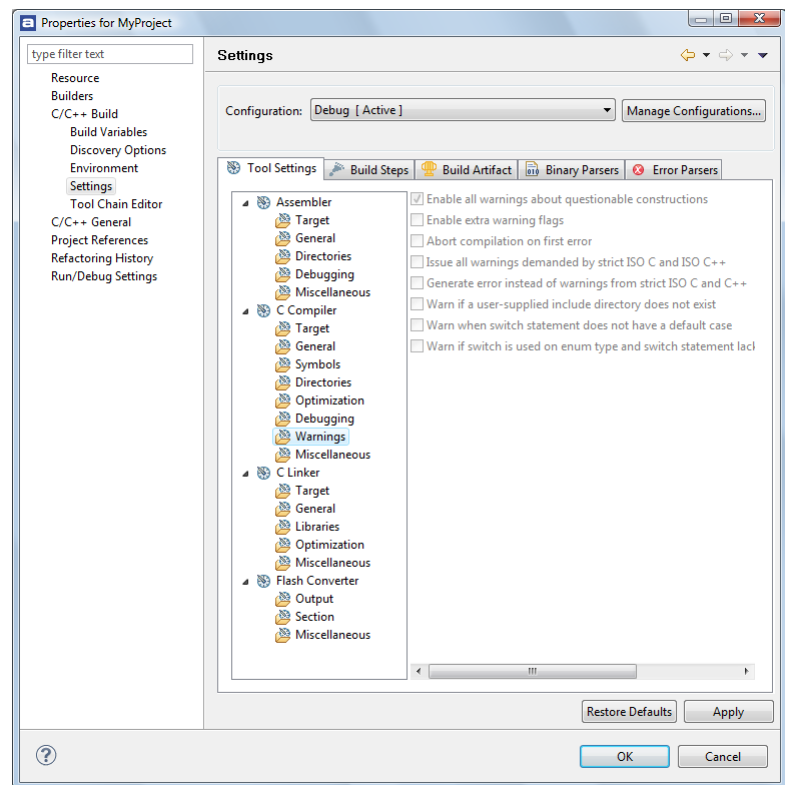


Figure 10 - Project properties dialog box

4. Select panels as desired and configure the command line tool options using the GUI controls. Advanced users may want to enter command line options manually, and this can be done in the **Miscellaneous** panel for any tool.

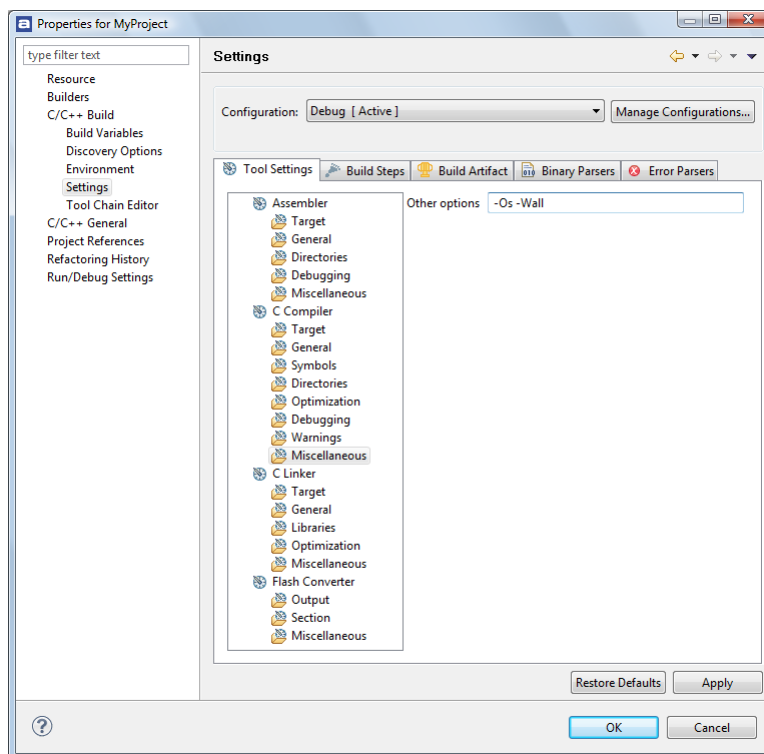


Figure 11 - Project properties dialog box

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

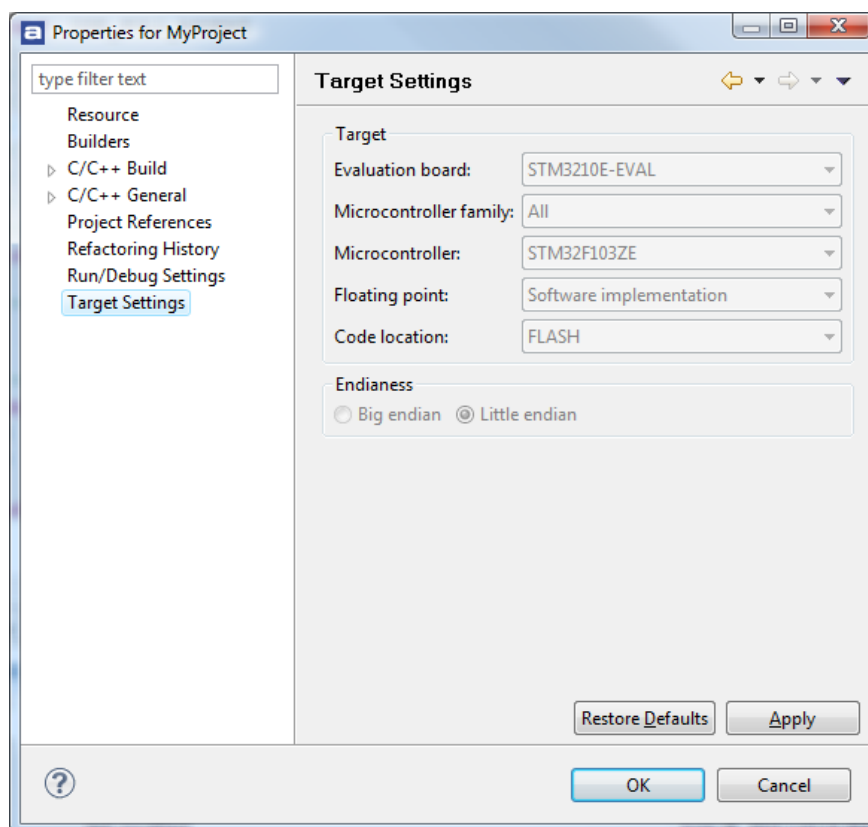


Figure 12 - Project properties dialog box

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

BUILDING THE PROJECT

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

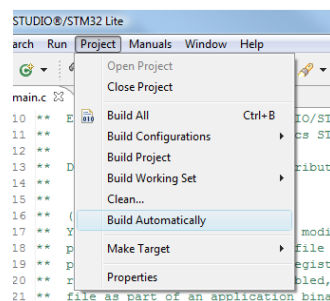


Figure 13 - Build automatically menu command

BUILD

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



Figure 14 - Build toolbar button

REBUILD ALL

To force a “rebuild all”, perform the following steps:

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

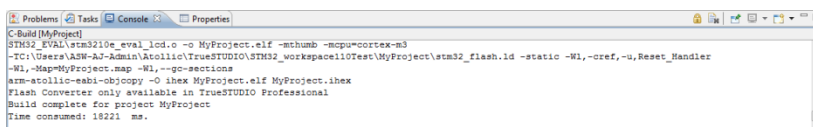


Figure 15 - Build console

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

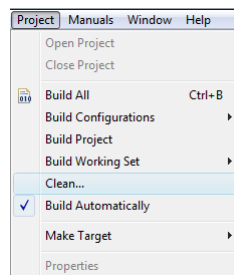


Figure 16 - Clean project

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

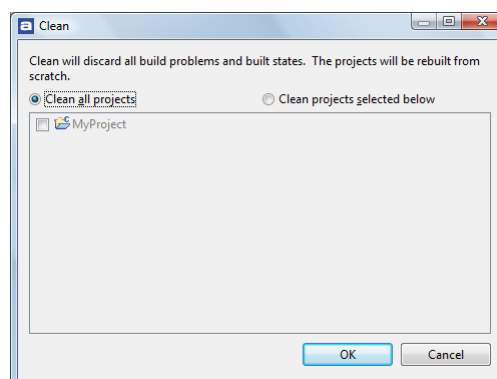


Figure 17 - Clean project dialog box

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

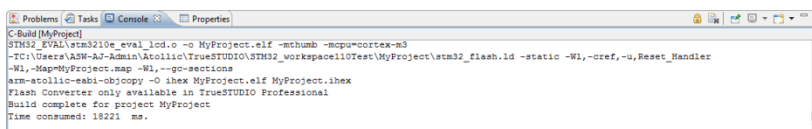


Figure 18 - Build console

IMPORTING SOURCE FILES

Importing the 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.

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 FILES IN AN EXTERNAL LOCATION

To make the project compile files located outside of the project directory, you need to create a link to it in the project. This is done by performing the following steps:

1. Right-click on the source directory in the **Project Explorer** view. Select the **New**, **File** menu command:

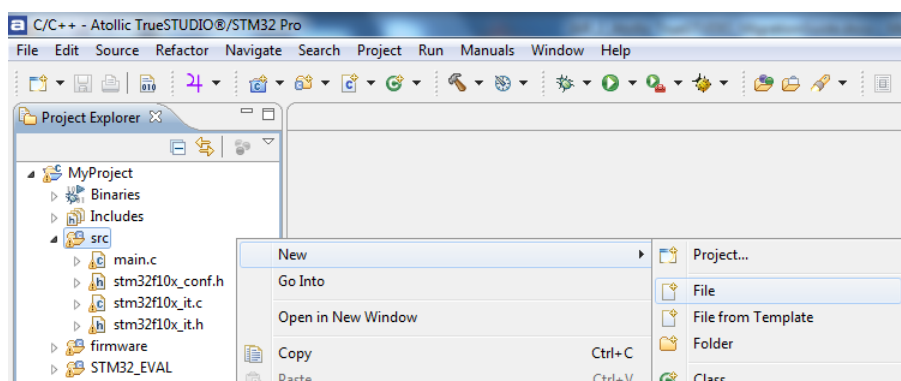


Figure 19 - Creating a link to an external file

2. Click on the **Advanced** button and select the **Link to file in the file system** checkbox:

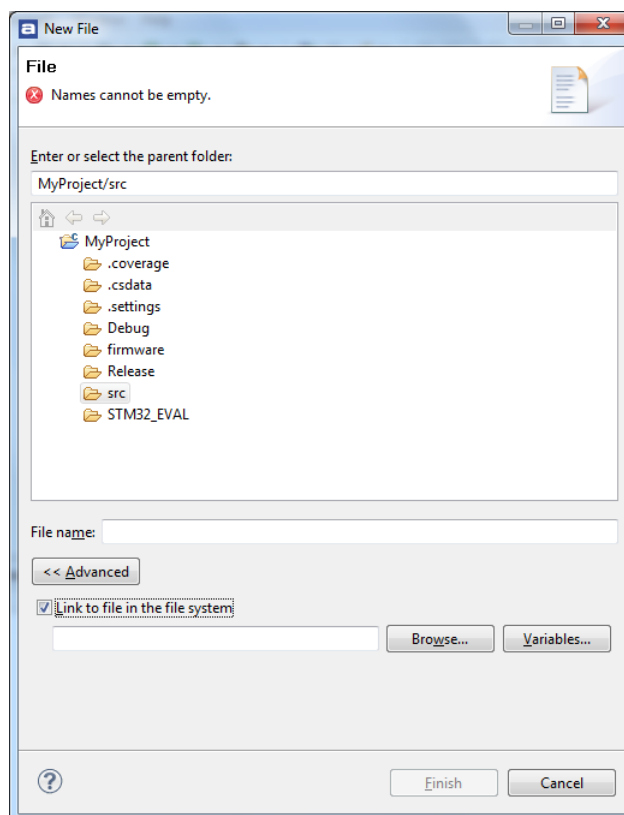


Figure 20 - Creating a link to an external file

3. Click the **Browse** button and point out the external file you want to compile as part of the project. A link to the file will be created in the project file tree, and the selected file will be compiled as part of a project build, even though the file is not physically stored in the project directory.

USING DIRECTORIES IN AN EXTERNAL LOCATION

To make the project compile many files located in a directory outside of the project directory, you need to create a link to it in the project. This is done by performing the following steps:

1. Right-click on the source directory in the **Project Explorer** view. Select the **New, Folder** menu command:

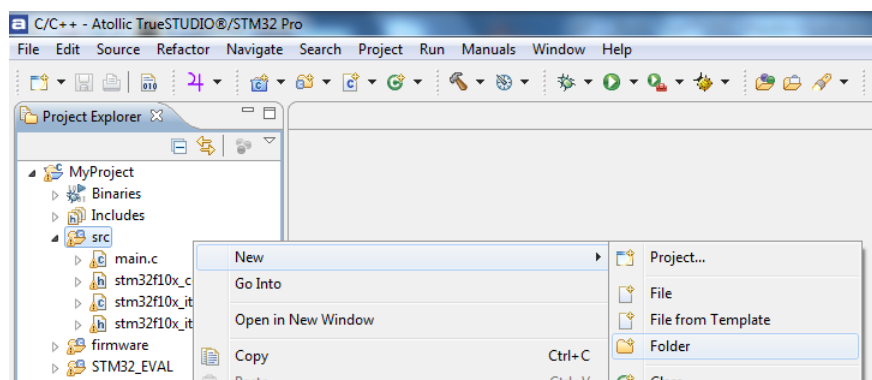


Figure 21 - Creating a link to an external folder

2. Click on the **Advanced** button and select the **Link to folder in the file system** checkbox:

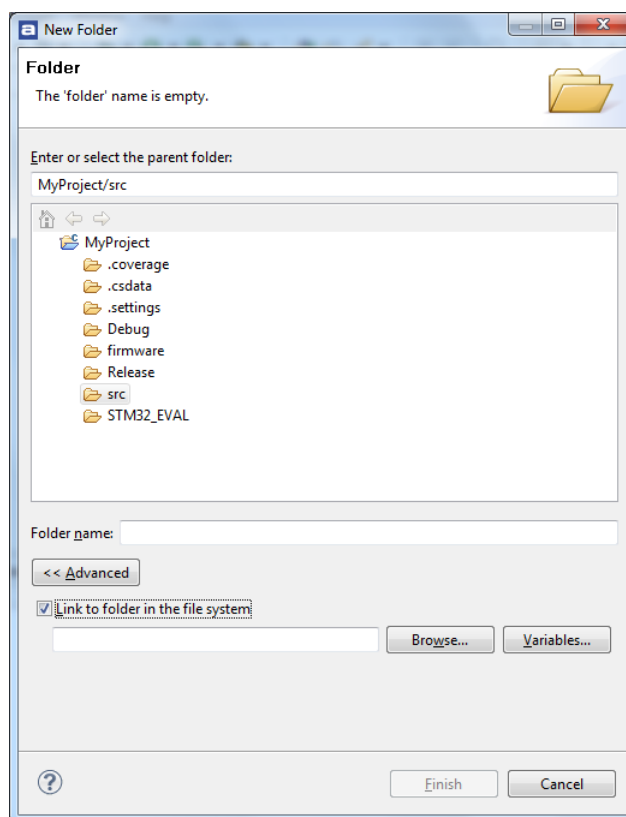


Figure 22 - Creating a link to an external folder

3. Click the **Browse** button and point out the external folder you want to compile as part of the project. A link to the folder will be created in the project file tree, and

the files in the selected folder will be compiled as part of a project build, even though the files are not physically stored in the project directory.



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 has 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 EmbeddedWorkbench® 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 1 - IAR EmbeddedWorkbench® 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).

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 EmbeddedWorkbench® toolchain, and discusses how they may be supported in a file to be compiled using the GNU toolchain.

INLINE ASSEMBLER

	IAR	GCC
Keyword:	<code>__asm</code>	<code>__asm</code>
Pragma:		
Syntax:	<code>__asm("<assembler>")</code>	<code>__asm("<assembler>")</code>
Comment:	GCC supports a superset of the IAR EmbeddedWorkbench® 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 EmbeddedWorkbench® toolchain uses the newline character '\n'.	

INLINE FUNCTIONS

	IAR	GCC
Keyword:	<code>inline</code>	<code>inline</code>
Pragma:	<code>inline</code>	<code>inline</code>
Syntax:	<code>#pragma inline</code> <code>void foo (void)</code> <code>{...}</code> <code>inline void foo2 (void)</code> <code>{...}</code>	<code>inline void foo2 (void)</code> <code>{...}</code>
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:	<code>__ramfunc</code>	<code>__attribute__((section("NAME")))</code>
Pragma:		
Syntax:	<code>__ramfunc void foo (void)</code> <code>{...}</code>	<code>void foo (void) __attribute__((section("RAMFNS")))</code> <code>{...}</code>
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 ("RAMFNS" in the example), and modifying the startup code and linker script accordingly.

INTERRUPT AND EXCEPTION FUNCTIONS

	IAR	GCC
Keyword:	<code>__swi</code> <code>__fiq</code> <code>__irq</code>	<code>__attribute__((interrupt("SWI")))</code> <code>__attribute__((interrupt("FIQ")))</code> <code>__attribute__((interrupt("IRQ")))</code>
Pragma:	<code>swi_number=NN</code>	
Syntax:	<code>__irq void foo (void)</code> <code>{...}</code>	<code>void foo (void) __attribute__((interrupt("IRQ")))</code> <code>{...}</code>
Comment:	The IAR EmbeddedWorkbench® "#pragma swi_number=N" is required in addition to the <code>__swi</code> keyword to define the SWI number used.	

NESTED INTERRUPT FUNCTIONS

	IAR	GCC
Keyword:	<code>__nested</code>	<code>__attribute__((nesting))</code>
Pragma:		
Syntax:	<code>__nested __irq</code> <code>void foo (void)</code> <code>{...}</code>	<code>void foo (void) __attribute__((nesting))</code> <code>{...}</code>
Comment:	The tagged function is marked as non-returning.	

NON-RETURNING FUNCTIONS

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

ARM® SPECIFIC FUNCTIONS

	IAR	GCC
Keyword:	<code>__arm</code> <code>__thumb</code> <code>__interwork</code>	<code>__attribute__((short_call))</code> <code>__attribute__((long_call))</code>
Pragma:		<code>long_calls</code>
Syntax:	<code>__arm void foo (void)</code> <code>{...}</code>	
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:	<code>__weak</code>	<code>__attribute__((weak))</code>
Pragma:	<code>weak</code>	<code>Weak</code>
Syntax:	<code>__weak int i;</code> <code>__weak void foo (void)</code> <code>{...}</code>	<code>int i __attribute__((weak));</code> <code>void foo (void) __attribute__((weak));</code> <code>{...}</code>
Comment:	The function/variable is defined as weak, so will be overridden at link time by any non-weak function with the same name. If no non-weak function is defined, it will be included in the linked image.	

ROOT FUNCTIONS AND UNREFERENCED DATA

	IAR	GCC
Keyword:	<code>__root</code>	<code>__attribute__((used))</code>
Pragma:	<code>required</code>	
Syntax:	<code>#pragma required=i</code> <code>int i;</code> <code>__root void foo (void)</code> <code>{...}</code>	<code>int i __attribute__((used));</code> <code>void foo (void) __attribute__((used));</code> <code>{...}</code>
Comment:	IAR EmbeddedWorkbench® 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:	<code>__packed</code>	<code>__attribute__((packed))</code>
Pragma:	<code>pack</code>	<code>Pack</code>
Syntax:	<pre>#pragma pack(4) struct tag1 { char a; int b; } mystruct1; #pragma pack() __packed struct tag1 { char a; int b; } mystruct1;</pre>	<pre>#pragma pack(4) struct tag { char a; int b; } mystruct; #pragma pack() struct tag1 { char a; int b __attribute__((packed)); } mystruct1;</pre>
Comment:	GCC supports extension using the ' <code>__attribute__</code> ' keyword, and allows packing on individual elements of a structure.	

ALIGNMENT OF DATA

	IAR	GCC
Keyword:		<code>__attribute__((aligned(N)))</code>
Pragma:	<code>data_alignment</code>	
Syntax:	<pre>#pragma data_alignment(8) int i;</pre>	<pre>int i __attribute__((aligned(8)));</pre>
Comment:	The IAR EmbeddedWorkbench® toolchain only supports a <code>#pragma</code> , whereas GCC supports an <code>__attribute__</code>	

ENDIAN SETTING OF DATA

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

NON-INITIALISED DATA

	IAR	GCC
Keyword:	<code>__no_init</code>	<code>__attribute__((section("no_init")))</code>
Pragma:		
Syntax:	<code>__no_init int i;</code>	<code>int i __attribute__((section("no_init")));</code>
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:	<code>@</code>	<code>__attribute__((section("NAME")))</code>
Pragma:	<code>location</code>	
Syntax:	<code>#pragma location i=0x0100 int i; __no_init int j @ 0x0104;</code>	<code>#define i (*(int *)0x0100)</code>
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.	

BUILT-IN FUNCTIONS

In addition to built-in functions to access processor instructions, the IAR EmbeddedWorkbench® 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.

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
Directive naming	.<name>	<NAME>
Multiline comment	/* ... */	/* ... */
Single line comment	# or @	// or ;
Statement delimiter	; or newline	Newline only
Binary Constants	0 nnnn 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 nnnn h or 0x nnn or h' nnn

Table 2 - 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 EmbeddedWorkbench® and the GNU toolchains for startup code.

Property	GNU	IAR
Startup file	startup_xxx_cl.s (e.g. startup_stm32f10x_cl.s)	cstartup_M.s
Entry Point	ResetHandler	__iar_program_start
Hardware Initialisation	SystemInit	__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 3 - 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.

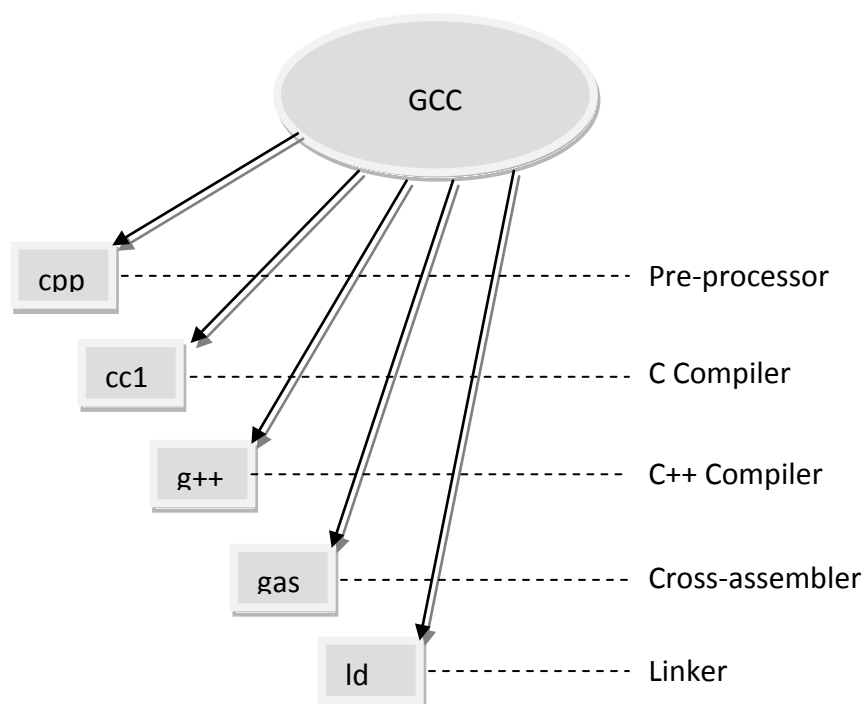
This section covers the above topics in detail, allowing the final control and customization of your migrated project.

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.



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 EmbeddedWorkbench® 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 are available to provide fine grained control.

IAR Option	GNU Option	GCC manual Section	Comment
------------	------------	-----------------------	---------

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 EmbeddedWorkbench®), 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.
--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
--only_stdout	N/A		Uses standard output only – this can be achieved by the shell used to invoke the

IAR Option	GNU Option	GCC manual Section	Comment
			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_separators	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 4 - 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)

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

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

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

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

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

    .sidata = .;

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

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

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

```

Define the entry point of the application as 'ResetHandler'

Define symbols with constant values

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

Start allocation of sections to memory

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

Define the symbol '**.sidata**' to be at the current location in memory (just after the vector table). Using the 'AT' directive, place the **contents** of the '.data*' sections at the address 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-volatile memory, and copied at startup into the runtime memory image. Symbols **_sidata** and **_edata** are defined for the start and end addresses of the collection of '.data' sections. Thus initialisation copies from **.sidata** to **_sidata**.

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)

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

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 EmbeddedWorkbench® 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 5 - 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 EmbeddedWorkbench® 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 EmbeddedWorkbench® 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 EmbeddedWorkbench® 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 EmbeddedWorkbench® archive tool to extract all modules from the IAR EmbeddedWorkbench® 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 -cr 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: <ul style="list-style-type: none">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 EmbeddedWorkbench® 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.

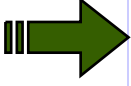
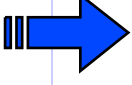
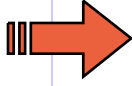
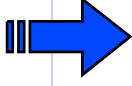
Looking at the two files, it is possible to see the parts of the code responsible for managing the function 'prolog' and 'epilog' – i.e. setting up the stack frame, managing incoming parameters and setting the return value, plus the parts of the code responsible for calling the library function 'foo'.



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		GNU		Merged
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		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	  	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 EmbeddedWorkbench® 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 EmbeddedWorkbench® 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.