Application Note

IAR to CCS Project Porting Guide

ABSTRACT

Migrating embedded projects between IDEs requires careful planning due to differences in toolchains, project structures, and configurations. This document is a structured guide to transition projects from IAR Embedded Workbench® to Texas Instruments' Code Composer Studio™ (CCS).

Table of Contents

List of Figures

List of Tables

Trademarks

1 Introduction

This application note is designed to assist developers in migrating code and projects written with IAR Embedded Workbench to Code Composer Studio (CCS). IAR and CCS are two widely used embedded development environments provided by IAR Systems and Texas Instruments, respectively. There are differences in functionality, interface design, and usage habits between the two, so special attention needs to be paid to compatibility and configuration issues during the migration process. This document provides detailed steps and considerations for migrating from IAR to CCS, helping developers complete the code migration smoothly. Although there are differences between the two in some aspects, careful inspection and adjustments can achieve a seamless transition. For new users, TI recommends to be familiar with the operation methods and interface layout of CCS to adapt more quickly.

2 Pre-Migration Preparation

2.1 CCS Version Comparison

During the preparation of this document, Texas Instruments released Code Composer Studio™ (CCS) v20, a significant architectural overhaul transitioning from the legacy Eclipse-based framework to the modern Theia IDE platform. While this update introduces enhanced toolchain integration and a streamlined user interface, the technical analysis and methodologies presented herein remain primarily grounded in CCS v12.8 and earlier iterations. The migration to CCS v20 has minimal bearing on the core content of this article; however, to make sure of clarity for readers utilizing the latest environment, TI provide a concise comparison of critical differences between v12.8 and v20 in the following section.

Table 2-1. Comparison between CCS v20 and CCS v12.8

	CCS v12.8 and Earlier	CCS v20
Architecture	Eclipse Rich Client Platform	Eclipse Theia
Strengths	Mature and stable, good for deeply customizable plugins and toolchains in embedded development.	Modern architecture supporting cloud or desktop hybrid workflows, native compatibility with VS Code extensions, and seamless DevOps integration.
Weaknesses	Relies on legacy technology, limited support for modern web standards, and higher memory and resource usage.	Smaller community ecosystem compared to Eclipse; some advanced plugins require third-party adaptation.
User Experience	Classic multi-window layout with nested menus and a steep learning curve.	VS Code-like interface with drag-and-drop panel customization (for example, terminal, memory views).

2.2 Pre-Migration Preparation

Before starting the migration, be familiar with the differences between IAR Embedded Workbench (EW) and Code Composer Studio that are in the toolchains, project management, and ecosystem integration. A concise breakdown of the differences are shown below.

1. Toolchain and Compiler: IAR uses a proprietary compiler, while CCS typically uses TI's compiler (based on GCC or Clang) or other supported compilers.

Table 2-2. Toolchain and Compiler Differences

IAR EW	CCS
Uses proprietary compiler of IAR (ICCARM for ARM).	Uses TI Arm Clang (based on LLVM/Clang) for TI devices.
Flags like --debug, -Oh, -D for defines.	Flags differ (for example, -g for debug, --define=NAME for macros).
Strict adherence to IAR-specific syntax (for example, #pragma vector).	Requires TI-compatible syntax (for example, __attribute__((interrupt))).

2. Project Structure: IAR and CCS have different project file structures and configurations.

Table 2-3. Project Structure Differences

IAR EW	CCS
Proprietary project format (.ewp, .eww).	Eclipse-based project (.cproject, .project).
Manages settings via GUI or .icf linker files.	Uses linker command files (.cmd) and Eclipse-style configuration menus.
Limited plugin ecosystem.	Extensible via Eclipse plugins (for example, TI Resource Explorer, GIT integration).

3. Debugging Tools and Hardware Support: CCS integrates TI-specific debugging tools, which can differ from the debugging environment of the IAR.

Table 2-4. Debugging and Hardware Support Differences

IAR EW	CCS
Broad third-party debug probes.	Support TI debug probes (XDS110 and so forth.) and third party debug probes
Requires manual HAL setup.	Pre-integrated TI libraries (for example, TI-RTOS, FreeRTOS).
Limited RTOS integration.	Native support for TI-RTOS and real-time debugging tools.

4. Ecosystem and Integration: CCS is free to use and supports a variety of tools to help users design projects.

Table 2-5. Ecosystem & Integration Differences

IAR EW	CCS
Paid license with limited free features.	Free tier with optional paid upgrades.
Minimal vendor-specific tools.	Tight integration with TI tools (for example, UniFlash, SysConfig).
Community support via IAR forums.	Strong TI community (E2E forums, detailed app notes).

5. Build and Optimization: CCS supports a variety of optimization level to meet different requirements.

Table 2-6. Build and Optimization Differences

IAR EW	CCS
Known for highly optimized code.	Balances optimization with TI-specific tuning.
Custom build steps via GUI.	Flexible build customization using Eclipse or Makefile.
Static memory allocation via .icf.	Dynamic linker configuration (.cmd files).

3 Porting Code to CCS

3.1 Prepare for Porting

First, understand the IAR project structure. Document the project hierarchy, source files, and dependencies. Note the target microcontroller (MCU) and the variant (for example, TI MSP430, Arm® Cortex®-M). Identify compiler and linker flags, memory configurations (.icf/.xcl files), and preprocessor symbols.

Second, backup the project. Create a copy of the original IAR project for reference and rollback.

Third, review code for toolchain-specific features. Issues like, check for IAR-specific pragmas, intrinsics (for example, __no_init), or inline assembly. Identify dependencies on IAR libraries or runtime files (for example, low_level_init.c). Note down important settings such as: compiler flags, linker configuration, memory layout (for example, linker script or ICF file), preprocessor definitions and include paths.

Finally, identify project dependencies. List all external libraries, drivers, and middleware used in the IAR project.

3.2 Set Up CCS Environment

1. Install CCS:
   a. Download and install the latest version of Code Composer Studio from TI's website. Make sure that the required device support packages (Cores, Compilers, and Debuggers) are installed in CCS.
   b. Install device-specific SDKs (for example, MSP432, C2000, or SimpleLink SDKs) by TI Resource Explorer or TI website.
   c. Install required libraries. If the project uses TI-specific libraries (for example, DriverLib, TivaWare, or MSPWare), then download and install them.
2. Create a New CCS Project:
   a. File → New → CCS Project.
   b. Select the target MCU, compiler (TI Arm Clang), and project template (for example, *Empty Project*).
   c. Make sure the output format (for example, ELF) matches the target.

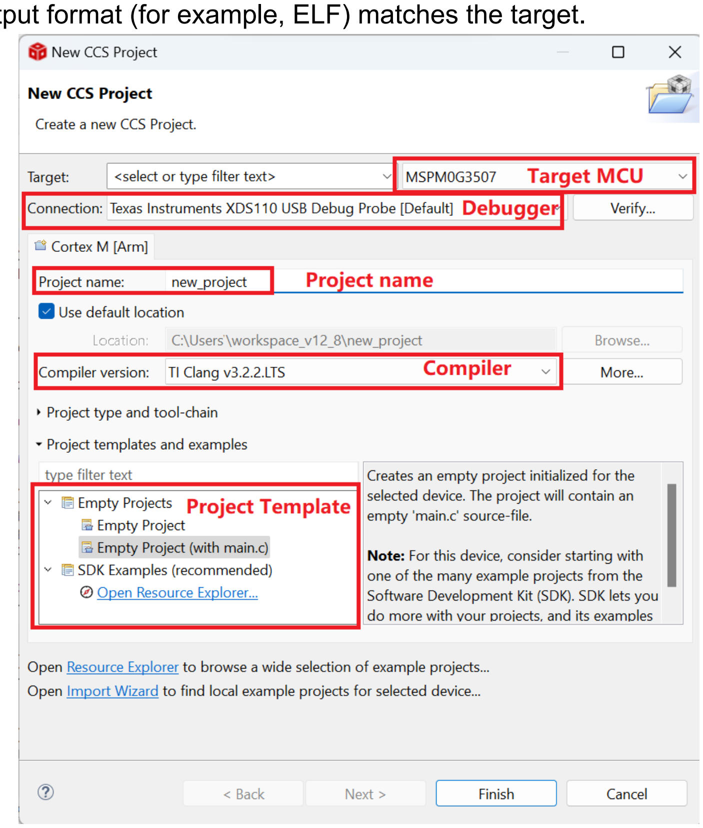

Figure 3-1. Create a New CCS Project

3.3 Import Source Code and Files in CCS

1. Copy source files:
   a. Copy the source files (`.c`, `.h`, `.asm`) from the IAR project to the CCS project directory.
   b. Use Project Explorer → Right-click *Import* → *File System* to add files.
2. Include paths and preprocessor symbols:
   a. Add the necessary include paths in the project properties (Right-click *Project* > Properties > Build > Include Options). Shown in figure below.
   b. Under *Predefined Symbols*, define macros if necessary.

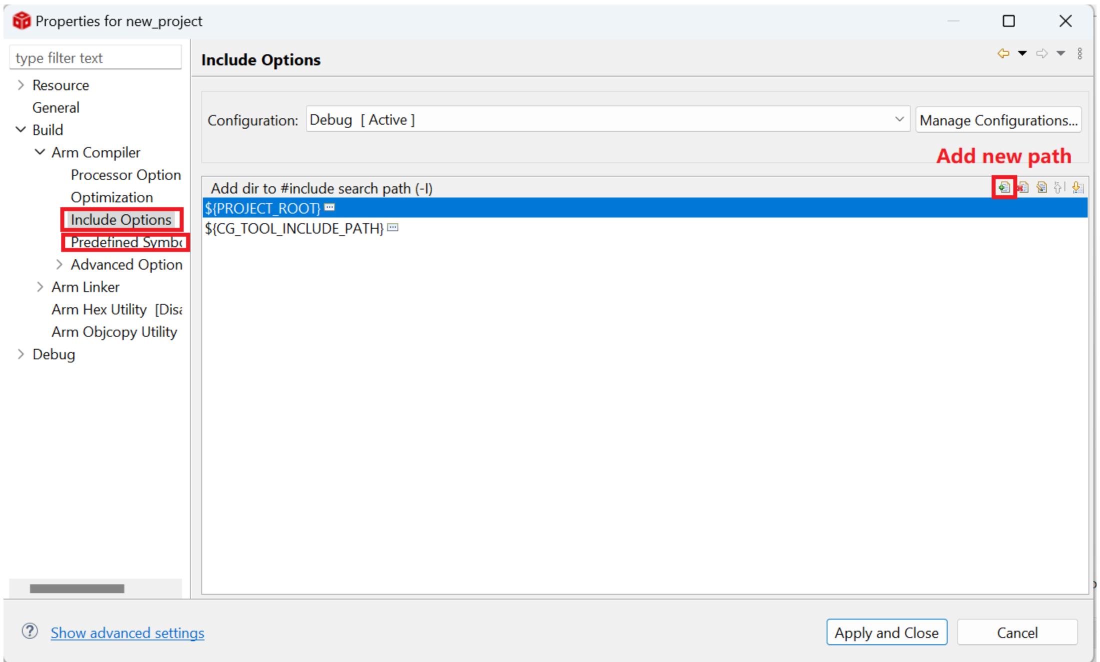

Figure 3-2. CCS Project Properties

3. Linker configuration:
   a. Replace IAR .icf/.xcl with a TI linker command file (.cmd).
   b. Configure memory regions (for example, FLASH, RAM) in the .cmd file. Users not familiar with .cmd file need to refer to the TI Linker Command File Primer for basic explanation of the code, which typically appears in most TI linker command files.

A typical IAR .icf file

```
******************************************************************************/
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_4.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x0000000;

/*-Memory Regions-*/
define symbol __ICFEDIT_region_IROM1_start__ = 0x00000000;
define symbol __ICFEDIT_region_IROM1_end__   = 0x0001FFFF;
define symbol __ICFEDIT_region_IROM2_start__ = 0x0;
define symbol __ICFEDIT_region_IROM2_end__   = 0x0;
define symbol __ICFEDIT_region_EROM1_start__ = 0x0;
define symbol __ICFEDIT_region_EROM1_end__   = 0x0;
define symbol __ICFEDIT_region_EROM2_start__ = 0x0;
define symbol __ICFEDIT_region_EROM2_end__   = 0x0;
define symbol __ICFEDIT_region_EROM3_start__ = 0x0;
define symbol __ICFEDIT_region_EROM3_end__   = 0x0;
define symbol __ICFEDIT_region_IRAM1_start__ = 0x20200000;
define symbol __ICFEDIT_region_IRAM1_end__   = 0x20207FFF;
define symbol __ICFEDIT_region_IRAM2_start__ = 0x0;
define symbol __ICFEDIT_region_IRAM2_end__   = 0x0;
define symbol __ICFEDIT_region_ERAM1_start__ = 0x0;
define symbol __ICFEDIT_region_ERAM1_end__   = 0x0;
define symbol __ICFEDIT_region_ERAM2_start__ = 0x0;
define symbol __ICFEDIT_region_ERAM2_end__   = 0x0;
define symbol __ICFEDIT_region_ERAM3_start__ = 0x0;
define symbol __ICFEDIT_region_ERAM3_end__   = 0x0;
define symbol __ICFEDIT_region_NON_MAIN_BCR_start__ = 0x41C00000;
define symbol __ICFEDIT_region_NON_MAIN_BCR_end__   = 0x41C0007F;
define symbol __ICFEDIT_region_NON_MAIN_BSL_start__ = 0x41C00100;
define symbol __ICFEDIT_region_NON_MAIN_BSL_end__   = 0x41C0017F;
/*-Sizes-*/
define symbol __ICFEDIT_size_proc_stack__ = 0x0000;
```

A typical CCS .cmd file

```
-uinterruptVectors
--stack_size=256


MEMORY
{
    FLASH           (RX)  : origin = 0x00000000, length = 0x00040000
    SRAM            (RWX) : origin = 0x20200000, length = 0x00008000
    BCR_CONFIG      (R)   : origin = 0x41C00000, length = 0x000000FF
    BSL_CONFIG      (R)   : origin = 0x41C00100, length = 0x00000080
}

SECTIONS
{
    .intvecs:   > 0x00000000
    .text   : palign(8) {} > FLASH
    .const  : palign(8) {} > FLASH
    .cinit  : palign(8) {} > FLASH
    .pinit  : palign(8) {} > FLASH
    .rodata : palign(8) {} > FLASH
    .ARM.exidx    : palign(8) {} > FLASH
    .init_array   : palign(8) {} > FLASH
    .binit        : palign(8) {} > FLASH
    .TI.ramfunc   : load = FLASH, palign(8), run=SRAM, table(BINIT)

    .vtable :   > SRAM
    .args   :   > SRAM
    .data   :   > SRAM
    .bss    :   > SRAM
    .sysmem :   > SRAM
    .stack  :   > SRAM (HIGH)

    .BCRConfig  : {} > BCR_CONFIG
    .BSLConfig  : {} > BSL_CONFIG
}
```

Figure 3-3. Linker File Comparison

4. Translate compiler and linker flags:
   a. Set stack or heap size in the linker file or by Project Properties → Build → ARM Linker → Basic Options.
   b. Map IAR flags to TI Arm Clang equivalents:

Table 3-1. A Comparsion Table of Common Used IAR Flag vs CCS Equivalent

IAR Flag	CCS Equivalent (TI Arm Clang)	Purpose
--debug	-g	Debug symbols
-Oh	-O3	High optimization
-DNAME	--define=NAME	Define preprocessor macro
--cpu=cortex-m4	-mcpu=cortex-m4	Target CPU
-I<path>	-I<path>	Include directory
--data_model medium	Not needed (configure in .cmd)	Memory model

3.4 Handle Device-Specific Code

1. Replace IAR Startup Code:
   a. Use TI-provided startup files (for example, startup_<device>.c from the SDK) instead of IAR's startup_<device>.s.
   b. Update interrupt vector tables to match TI's syntax (for example, #pragma DATA_SECTION for vectors).
2. Adapt Hardware Abstraction:
   a. Replace IAR-specific HAL functions with TI DriverLib or register-based code.
   b. Example: use MAP_GPIO_setAsOutputPin() instead of the GPIO library of the IAR.
3. Update Inline Assembly and Pragmas:
   a. Rewrite IAR-specific pragmas (for example, __packed becomes __attribute__((packed))).
   b. Convert inline assembly to TI Clang syntax.

Table 3-2. Comparison of Assembly Code Example in IAR and CCS

IAR EW	CCS
#pragma asm MOV R0, #0x10 #pragma endasm	__asm(" MOV R0, #0x10 ");

3.5 Adapt Code for CCS

1. Fix compiler-specific code:
    a. IAR and CCS compilers can have different syntax or behavior for certain constructs (for example, inline assembly, pragmas).
    b. Update any compiler-specific code to be compatible with CCS.
2. Replace IAR-specific functions:
    a. Replace IAR-specific functions (for example, `__enable_interrupt()`) with equivalent CCS or TI-specific functions.
3. Update debugging code:
    a. If the project uses IAR-specific debugging macros or functions, then replace those macros or functions with CCS-compatible alternatives.

3.6 Build and Debug

1. Build and validate the project:
    a. Resolve build errors if there is any. Check the error report in the Problems console first. Then locate the error and fix. Common issues are missing includes, undefined macros, and syntax mismatches.
    b. Adjust compiler optimization levels and other settings for performance or size. Please refer to TI ARM Clang Compiler User Manual for a more detailed description of the optimization options.
2. Create a debug configuration:
    a. Run → Debug Configurations → New Launch Configuration.
    b. Select the target connection (XDS110, JTAG, SWD).
    c. Use a .ccxml file to define the debug probe and MCU.
3. Run on hardware and debug:
    a. Connect the target microcontroller to CCS and load the program.
    b. Load the program and verify breakpoints, register views, and memory inspection. Verify that the program runs correctly and performs as expected.
    c. Compare behavior with the original IAR project.

4 Post-Migration Optimization

1. Document the migration:
   a. Document all changes made during the migration process for future reference.
   b. Update the project documentation to reflect the new CCS environment.
2. Common concerns:
   a. Interrupt handlers: Make sure ISRs use #pragma WEAK or are correctly named (for example, void TIMER0_A0_ISR(void)).
   b. Memory alignment: TI Clang can enforce stricter alignment than IAR. Use __attribute__((aligned(8))) if needed.
   c. Linker errors: Verify .cmd file addresses match the MCU memory map.

5 Summary

By following this application note, users can systematically transition projects to CCS while addressing toolchain-specific nuances. Test incrementally and leverage TI's robust debugging tools to streamline validation.

6 References

- Texas Instruments, ARM-CGT, webpage
- IAR Systems AB, IAR Embedded Workbench, webpage
- Texas Instruments, CCSTUDIO, webpage