ADuCM4x50 Device Family Pack for CCES Users Guide

ADuCM4x50 Device Family Pack for CCES Users Guide
Version 3.0.0, August 2021

http://www.analog.com

Contents

1 Introduction

1.1 Purpose

This document describes how to use the ADuCM4x50 Device Family Pack (DFP) with CrossCore Embedded Studio®. The ADuCM4x50 processor integrates an ARM Cortex-M4 microcontroller with various on-chip peripherals within a single package.

1.2 Scope of this Manual

This document describes how to install and work with the Analog Devices ADuCM4x50 DFP. This document explains what is included with the pack and how to configure the software to run the example applications that accompanies this package.

This document is intended for engineers who integrate ADI's device driver libraries with other software to build a system based on the ADuCM4x50 processor. This document assumes background in ADI's ADuCM4x50 processor.

1.3 Acronyms and Terms

ADI	Analog Devices, Inc.
API	Application Programming Interface
ARM	Advanced RISC Microcontroller
CCES	CrossCore Embedded Studio
CMSIS	Cortex Microcontroller Software Interface Standard
Cortex	A series of ARM microcontroller core designs
CRC	Cyclic Redundancy Check
DFP	Device Family Pack
HRM	Hardware Reference Manual
ISR	Interrupt Service Routine
IVT	Interrupt Vector Table
JTAG	Joint Test Action Group
NVIC	Nested Vectored Interrupt Controller
RISC	Reduced Instruction Set Computer
RTE	Run-Time Environment
RTOS	Real-Time Operating System
TRACE	Debugging with TRACE access port

1.4 Conventions

Throughout this document, we refer to several important installation locations: the ADuCM4x50 DFP, ARM CMSIS, and CCES toolchain installation root. Each of these packages can be installed in various places, which are referred to as follows:

- `<dfp_version>`
  - The version of the ADuCM4x50 Device Family Pack, e.g. 1.0.0 or 3.0.0.
- `<cces_root>`
  - The default CCES installer places the product at location **`C:/Analog Devices/CrossCore Embedded Studio 2.x.y`**, but the install location may vary depending on user preferences.
  - The default packs are placed at location `<cces_root>/ARM/packs/AnalogDevices`. There will be the following folder for ADuCM4x50 within that location called **`ADuCM4x50_DFP`**.
- **`<ADuCM4x50_root>`**
  - The directory `<cces_root>/ARM/packs/AnalogDevices/ADuCM4x50_DFP/<dfp_version>` which contains the content of the Analog Devices ADuCM4x50 DFP.
- **`<ARM_CMSIS_version>`**
  - The version of the ARM CMSIS pack, e.g. 4.5.0 or 5.0.1.
- **`<ARM_CMSIS_root>`**
  - The directory `<cces_root>/ARM/packs/ARM/CMSIS/<ARM_CMSIS_version>` which contains the content of the ARM CMSIS pack.

1.5 References

1. Analog Devices : `<ADuCM4x50_root>/Documents`
   a. ADuCM4x50_DFP_**`<dfp_version>`**_Release_Notes.pdf
   b. ADuCM4x50_DFP_Device_Drivers_UsersGuide.pdf
   c. ADuCM4x50_DFP_UsersGuide_CCES.pdf (this document)
   d. ADuCM4x50 Device Drivers API Reference Manual (`Docs/html` and hyperlinked)

2. For CrossCore Embedded Studio [http://www.analog.com]
    a. In CCES IDE, open **`Help->Help Contents`**: CCES on-line help.
        i. CrossCore Embedded Studio documentation
        ii. CMSIS C/C++ Development User's Guide
    b. **`<cces_root>/Docs`**: Release notes.
3. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*, Joseph Yiu, 3rd edition.
    - Every Cortex programmer’s bible; a must-have reference.
4. ICE-1000 or ICE-2000 Emulator [http://www.analog.com]

1.6 Additional Information

For more information on the latest ADI processors, silicon errata, code examples, development tools, system services and devices drivers, technical support and any other additional information, please visit our website at www.analog.com/processors.

1.6.1 Manual Contents

- Product Overview
- Installation Components
- ADuCM4x50 System Overview
- Build Configurations
- Examples
- Device Driver API Documentation

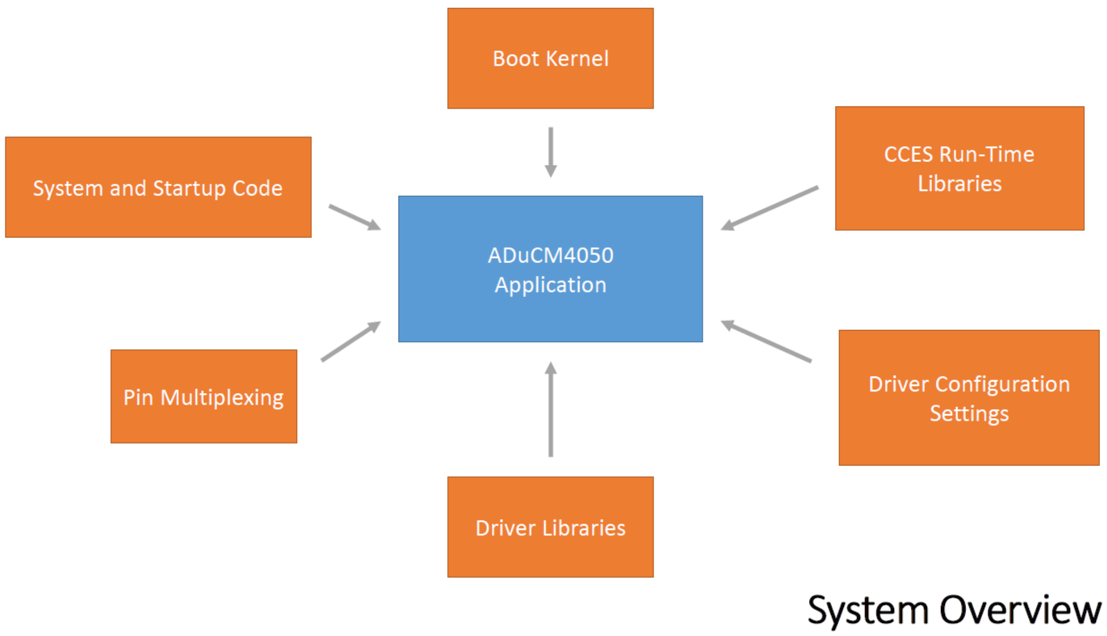
2 Product Overview

2.1 Software System Overview

The ADuCM4x50 Device Family Pack (DFP) provides files which are needed to write application software for the ADuCM4x50 processor. The product consists of a boot kernel, startup, system and driver source code, driver configuration settings, driver libraries, sample applications and associated documentation (see *Figure 1. Software Overview*).

The ADuCM4x50 DFP is designed to work with CrossCore Embedded Studio in CMSIS pack format for ARM.

Figure 1. Software Overview



2.2 Hardware System Overview

The examples provided with the ADuCM4x50 DFP run on the Analog Devices' ADuCM4x50-EZ-Kit evaluation board. The evaluation board is connected to the host computer using an ICE-1000 or ICE-2000 emulator over the evaluation board's debug port interface connectors. External I/O signals and system hardware are connected to the evaluation board connectors (see *Figure 3. ADuCM4x50 Evaluation Board with ICE-1000* and *Figure 5. ADuCM4x50 Evaluation Board with ICE-2000*).

If ICE-1000 is used, the emulator will physically block the RST, 3V, 5V, GND and VIN connections on the ADuCM4x50 board. To prevent this, an adapter can be used to connect ADuCM4x50 EZ-Kit and ICE-1000 (see *Figure 4. Adapter for ICE-1000*).

Figure 2. Hardware Overview

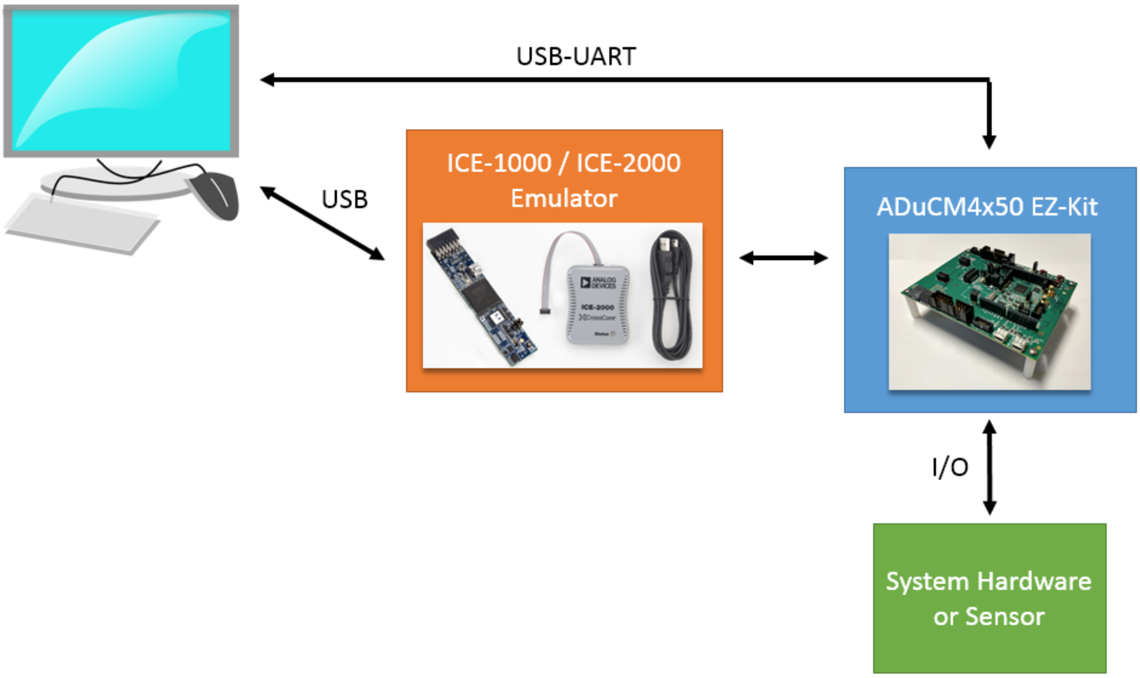


Figure 3. ADuCM4x50 Evaluation Board with ICE-1000

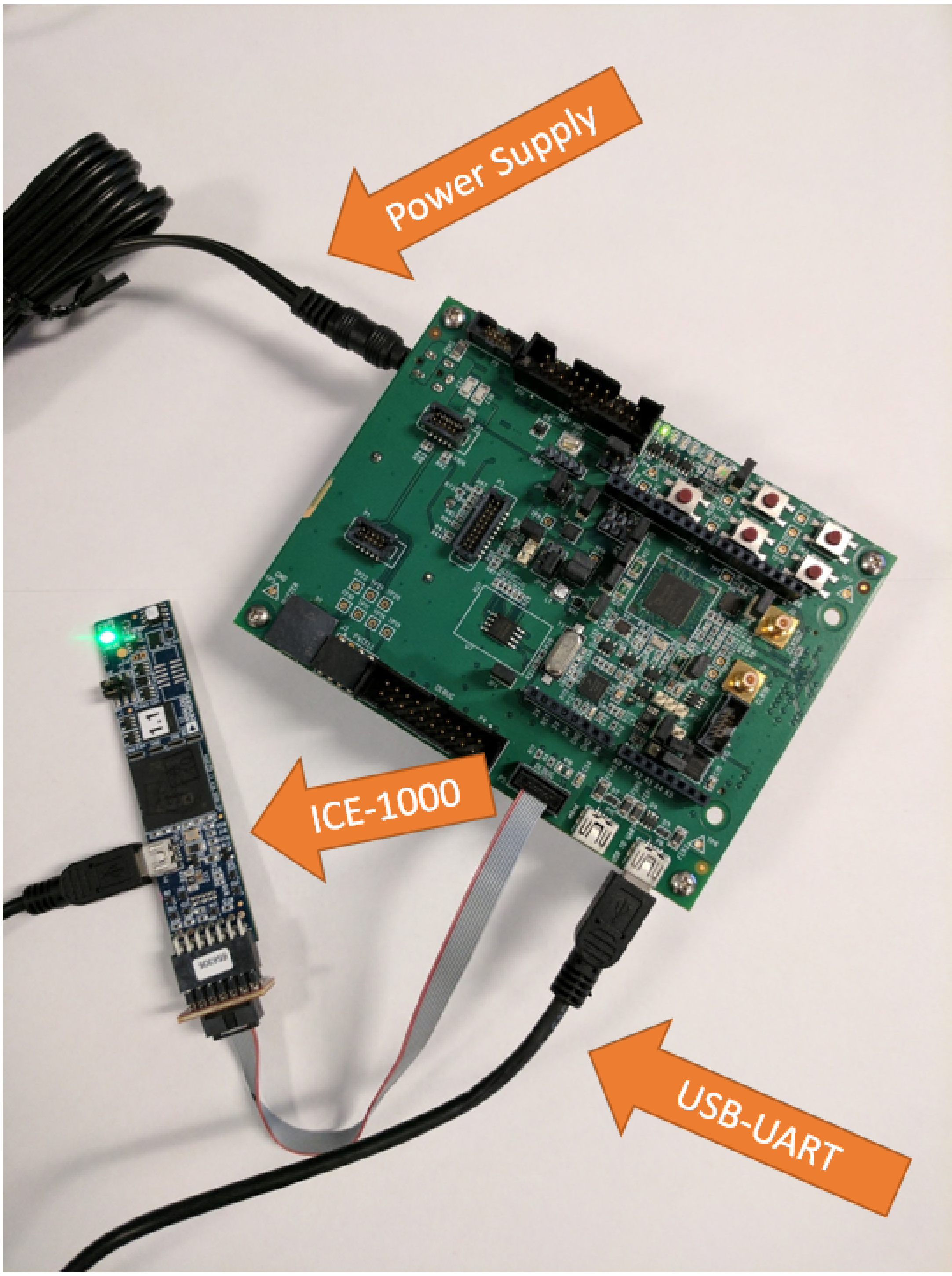


Figure 4. Adapter for ICE-1000

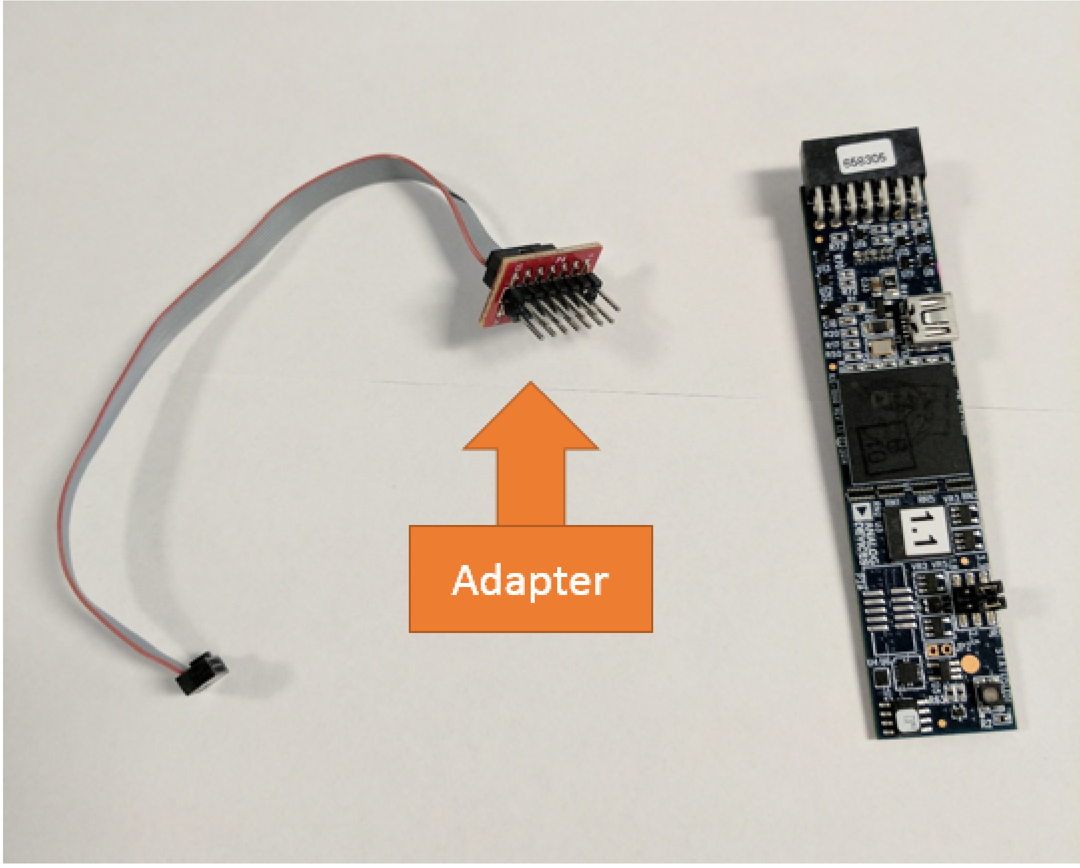
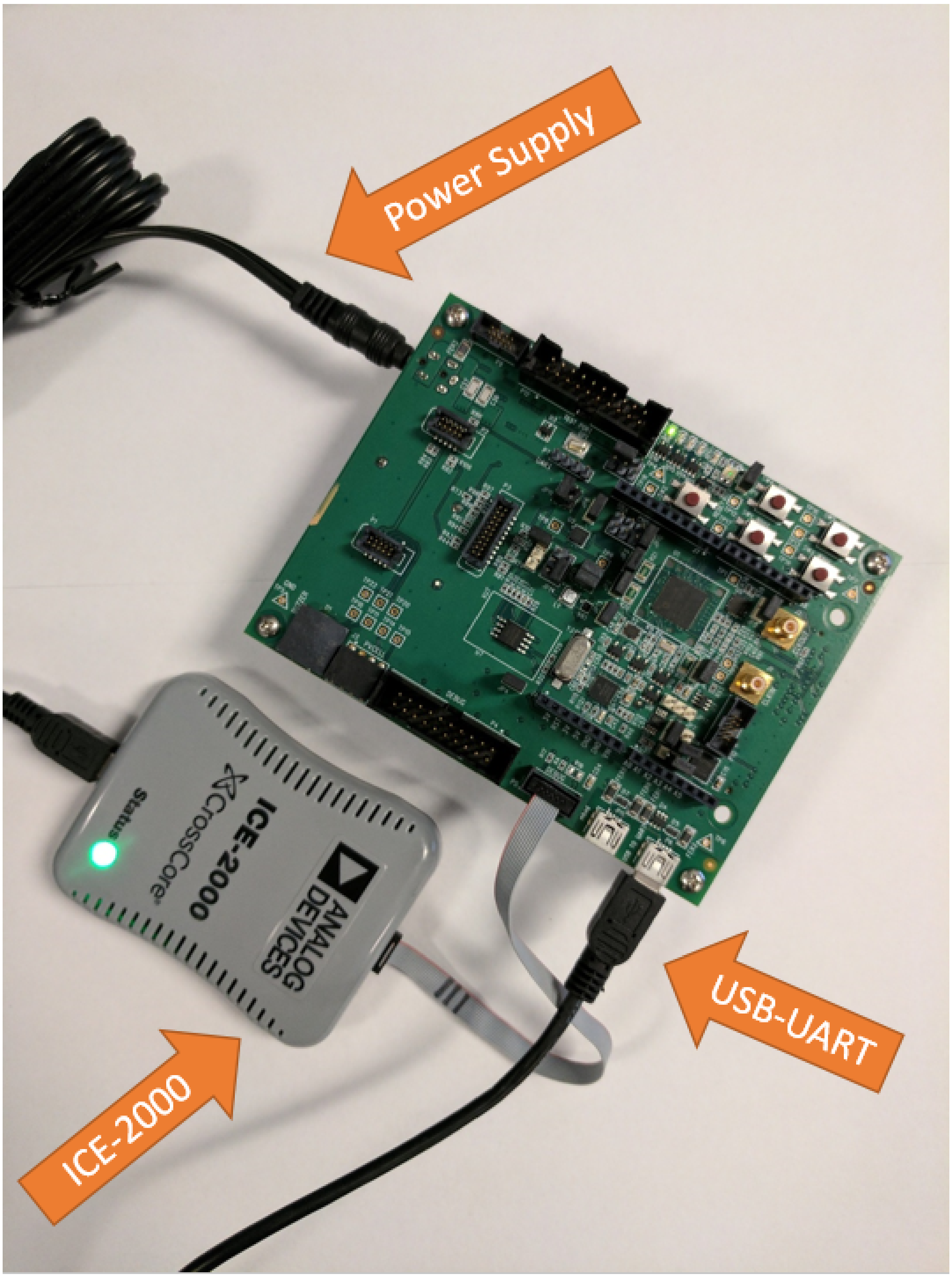


Figure 5. ADuCM4x50 Evaluation Board with ICE-2000



2.3 Documentation

The following documentation is provided for this release in the main product documentation directory:

- ADuCM4x50_DFP_<**dfp_version**>_Release_Notes.pdf

- ADuCM4x50_DFP_Device_Drivers_UsersGuide.pdf
- ADuCM4x50_DFP_UsersGuide_CCES.pdf (this document)

The *ADuCM4x50 Device Drivers API Reference Manual* is published in hyperlinked HTML format under the `html` directory.

Launch the "**<ADuCM4x50_root>**\Documents\html\index.html" file to browse the API documentation interactively.

The *ADuCM4x50 Device Drivers API Reference Manual* contains complete driver documentation, including API descriptions, data types, structures, parameters, return values, etc.

2.4 Technical or Customer Support

Submit your questions online at:

http://www.analog.com/support

E-mail your Processors and DSP applications and processor questions to:

processor.support@analog.com OR

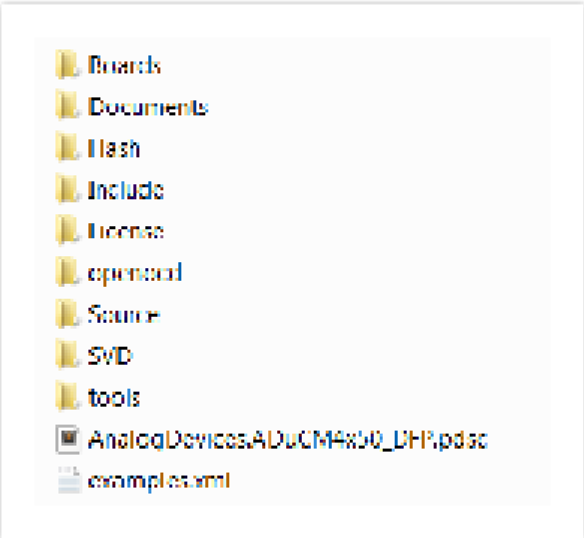
processor.china@analog.com (Greater China support)

3 Installation Components

CrossCore Embedded Studio 2.5.0 or later must be purchased and installed prior to installing the ADuCM4x50 Software Package. You can purchase CrossCore Embedded Studio from the Analog Devices website at http://www.analog.com/cces. Follow the instructions in the CrossCore Embedded Studio product installation procedure.

The ADuCM4x50 CMSIS pack contents (startup code, device drivers, libraries, examples, tools. documentation, etc.) are placed at **`<ADuCM4x50_root>`**.

Figure 6. Installation Directory Structure



3.1 CCES Project Support Files

This section documents the CCES-specific details of the ADuCM4x50 Device Family Pack. A working knowledge of the CCES toolchain and environment is assumed. See the CCES reference materials for details of installing, configuring and using the CCES tools. The following are the list of important files added by the ADuCM4x50 Device Family Pack, necessary to build and run applications in the CCES environment.

- OpenOCD scripts (.tcl & .cfg)
- Linker Scripts (.ld)

3.1.1 OpenOCD Scripts

OpenOCD supports the ADuCM4050 target via SWD (Serial Wire Debug) through the target config file `aducm4050.cfg`.

The target config file provides OpenOCD with the necessary definitions, including:

- Default values
- TAPs (Test Access Ports) to add to the scan chain
- CPU targets (including GDB support)
- CPU/Chip/CPU-Core specific features
- On-Chip flash definitions

For more information on using OpenOCD, see ARM Development Tools Documentation > OpenOCD User's Guide in the CCES On-line Help.

The **<ADuCM4x50_root>/openocd/scripts/target** folder contains *.tcl and *.cfg files.

- The `aducm4x50.tcl` file defines common routines for Analog Devices ADuCM4x50 targets.
- The `aducm4050.cfg` files defines target-specific information and includes `aducm4x50.tcl`.

3.1.2 Linker Scripts

The linker script maps sections from the input files into the output file, and controls the memory layout of the output file (See ARM Development Tools Documentation > Cortex-M > Binutils Linker Manual > Linker Scripts in CCES On-line Help).

The **<ADuCM4x50_root>/Source/GCC** folder contains *.ld files.

The `ADuCM4050.ld` file included in the ADuCM4x50 DFP configures:

- Memory regions
- Library group
- Sections and symbol values

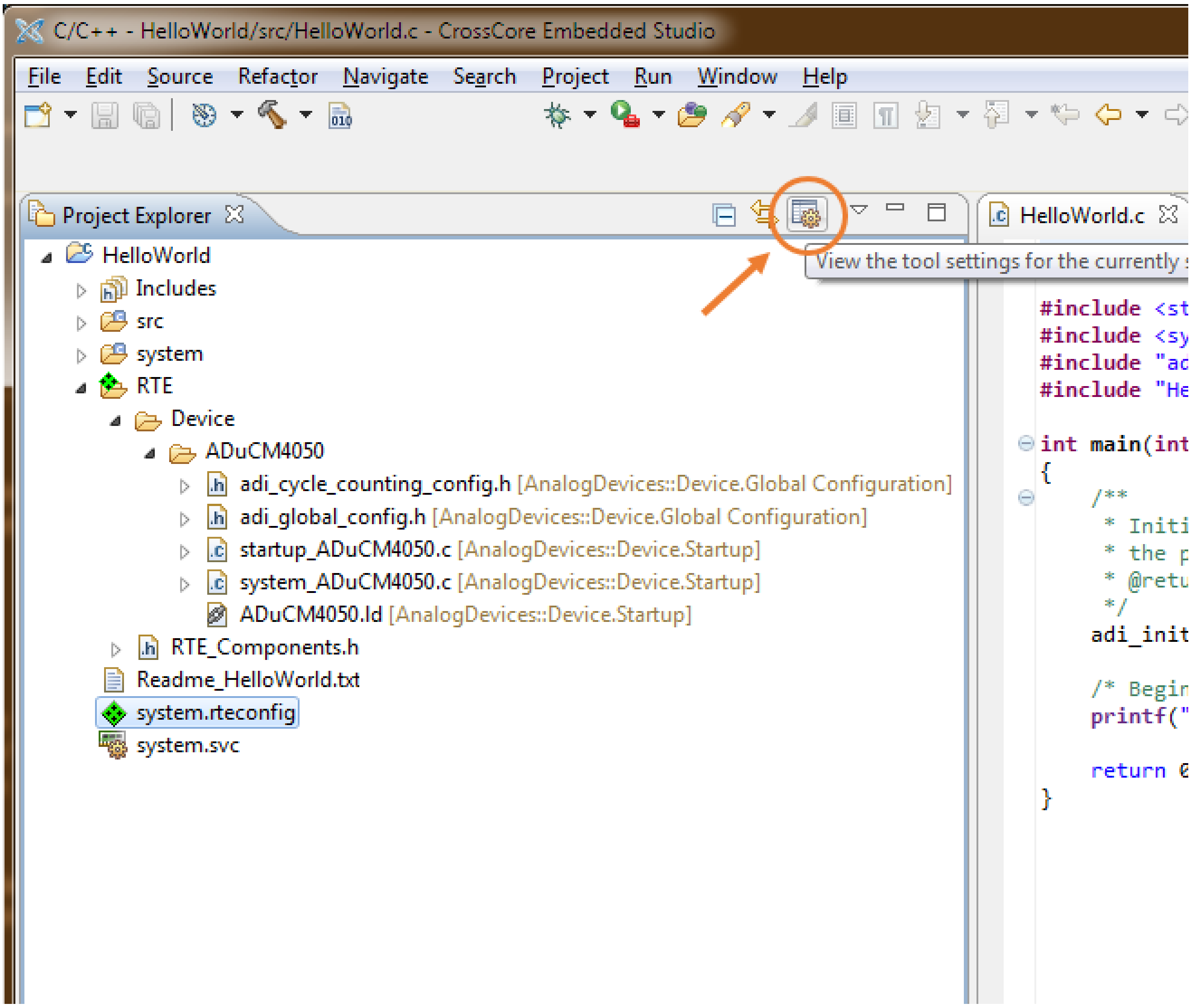
3.2 CCES Project Options

This section documents the CCES project options for ADuCM4x50 processors. A working knowledge of the CCES environment and CMSIS C/C++ development is assumed. See the CCES reference materials for details of installing, configuring and using the CCES tools as well as managing CMSIS packs.

3.2.1 Tool Settings

To access the project's tool settings, select the project in the Project Explorer and click the Tool Settings button shown in *Figure 7. Access Project Tool Settings* below.

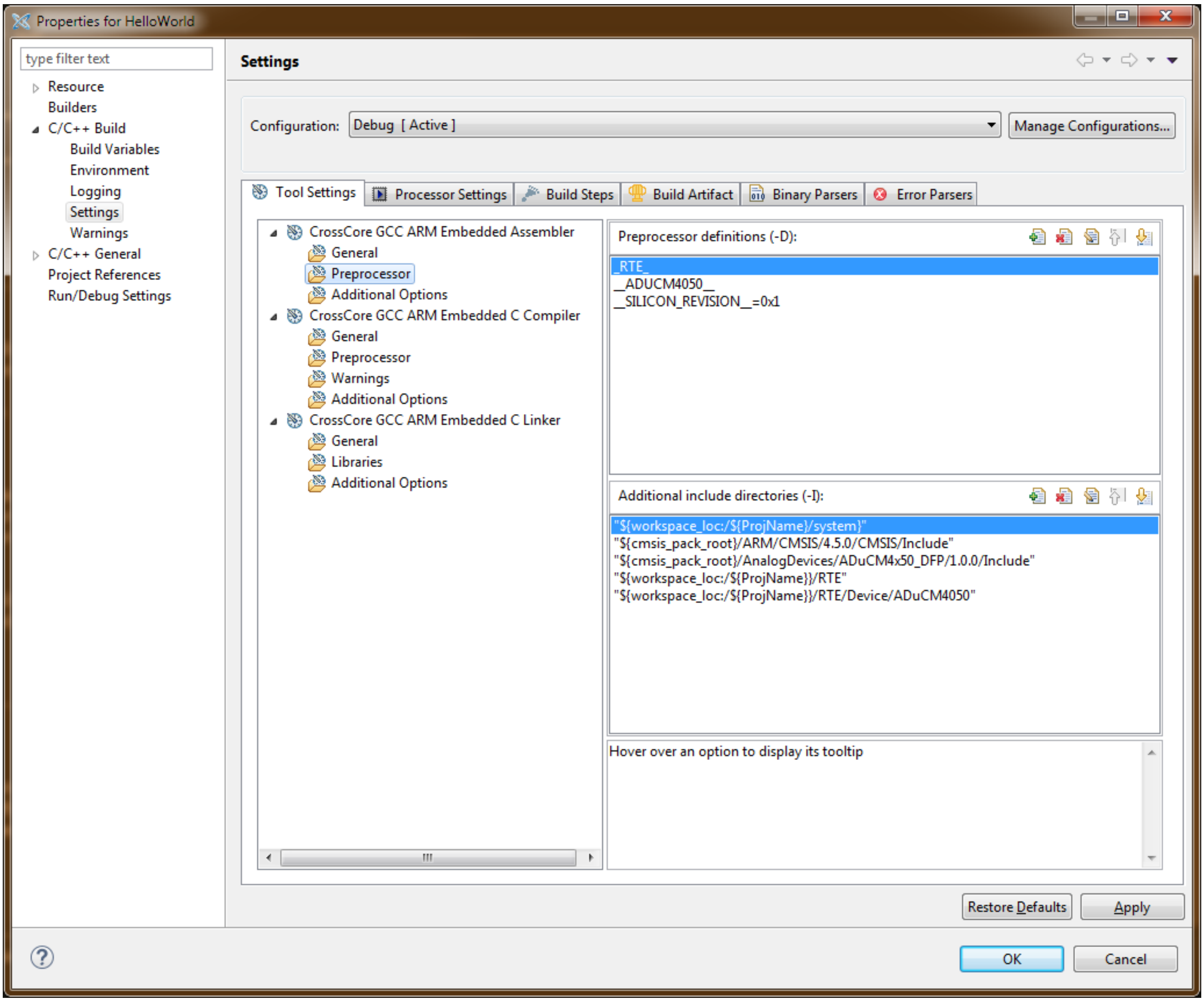
Figure 7. Access Project Tool Settings



CrossCore GCC ARM Embedded Assembler

This section allows the user to set any assembler flags, definitions and include search paths to be passed to the CrossCore GCC ARM Embedded Assembler during build. Definitions and include paths can be set from the Preprocessor sub-section as shown in *Figure 8. CrossCore GCC ARM Embedded Assembler - Preprocessor* below.

Figure 8. CrossCore GCC ARM Embedded Assembler - Preprocessor

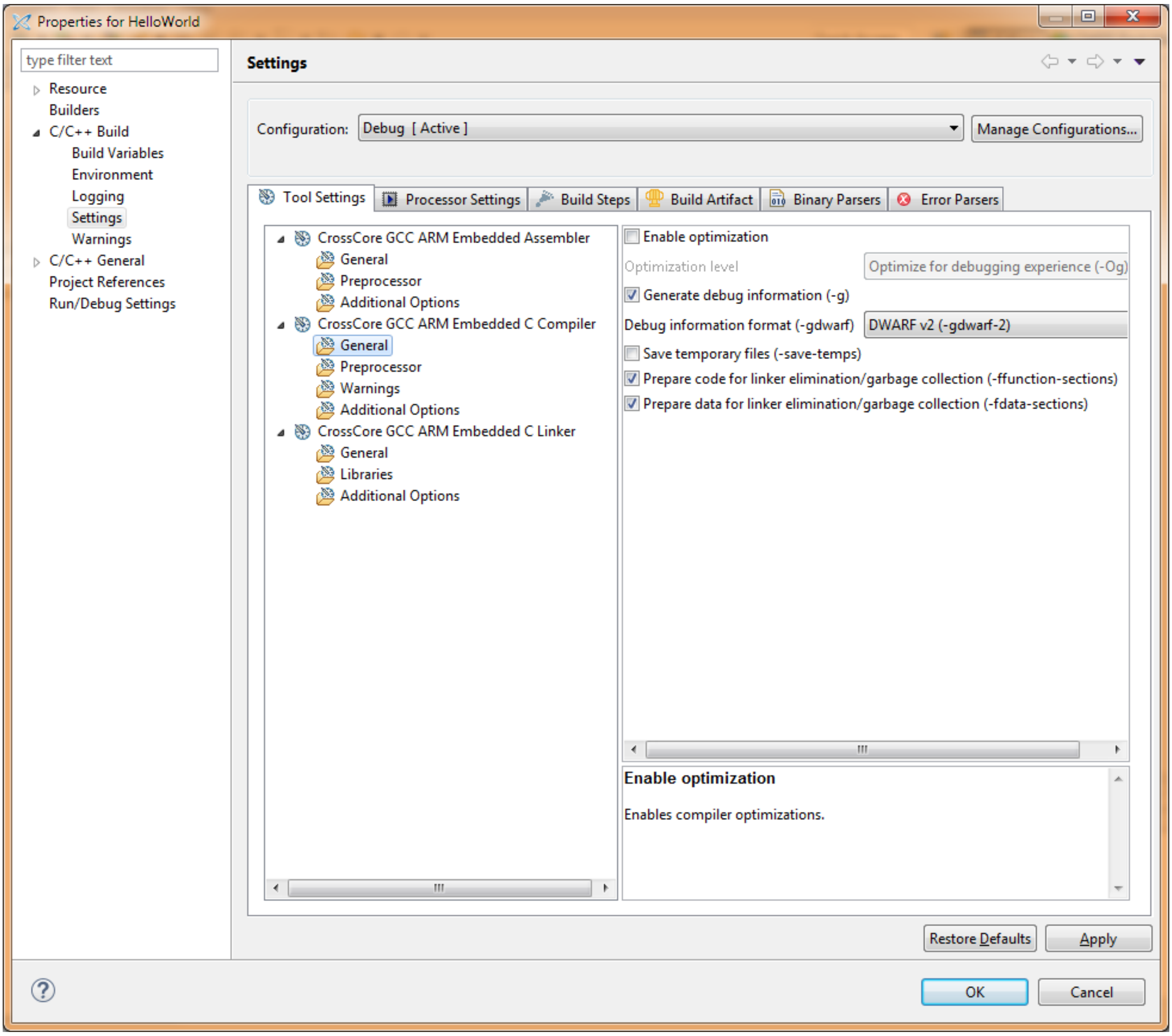


Additional assembler options can also be added from the Additional Options sub-section.

CrossCore GCC ARM Embedded C Compiler

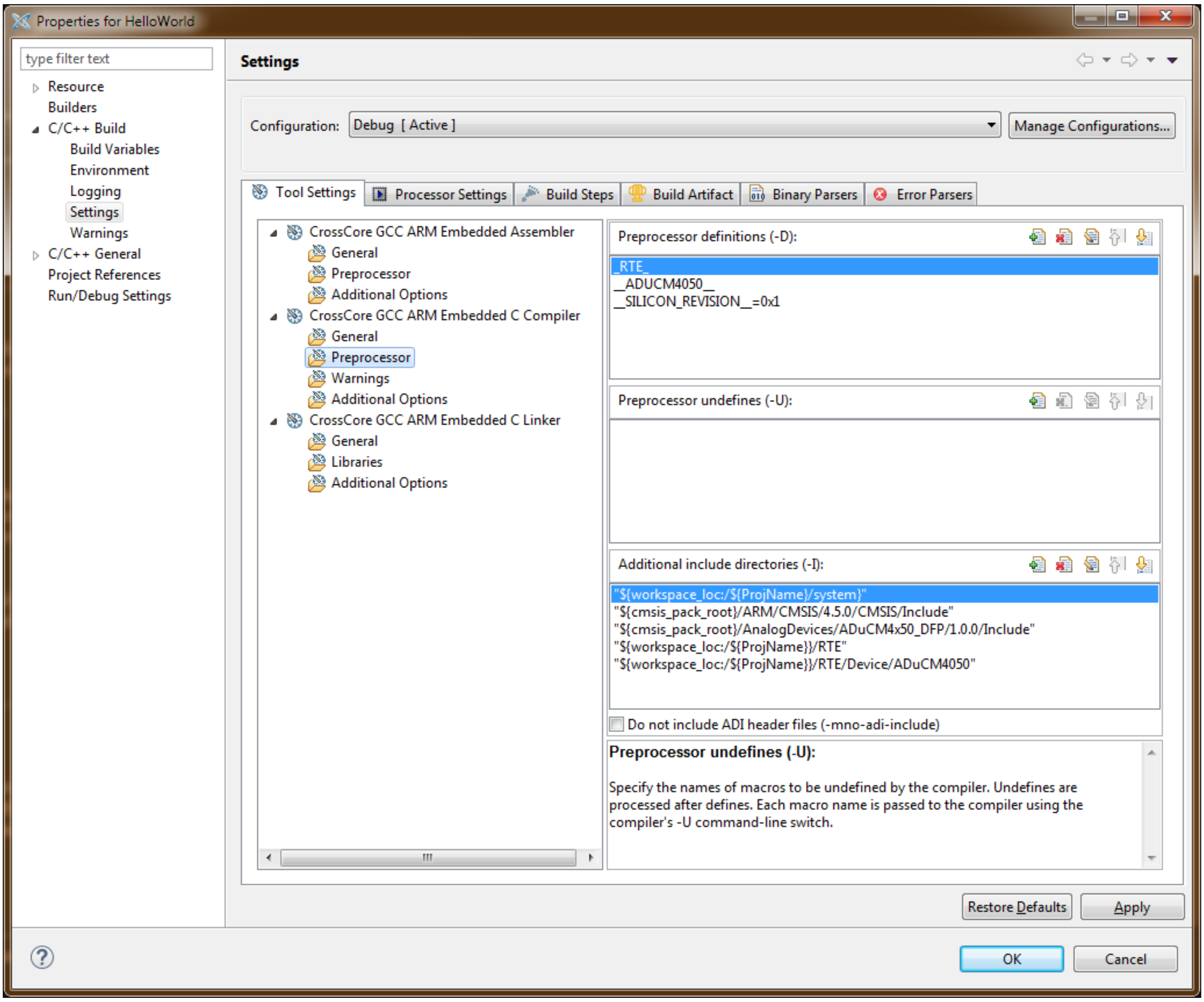
This section allows the user to set any compiler flags, definitions, include search paths and optimization to be passed to the CrossCore GCC ARM Embedded Compiler during build. Optimization level can be specified from the General sub-section as shown in *Figure 9. CrossCore GCC ARM Embedded C Compiler - General* below.

Figure 9. CrossCore GCC ARM Embedded C Compiler - General



Preprocessor definitions, undefines and include search paths can be set from the Preprocessor sub-section as shown in *Figure 10. CrossCore GCC ARM Embedded C Compiler - Preprocessor* below.

Figure 10. CrossCore GCC ARM Embedded C Compiler - Preprocessor

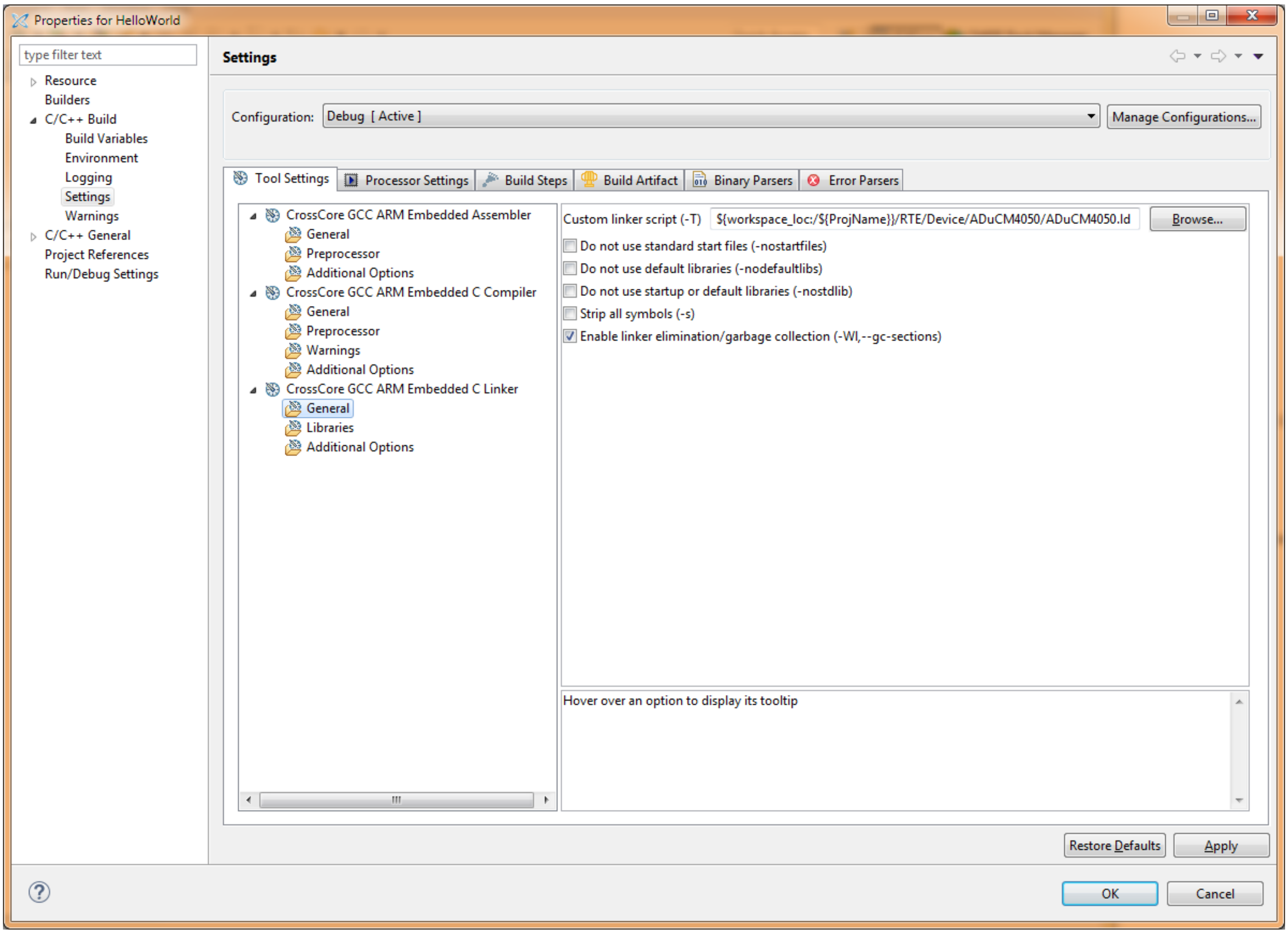


Additional compiler options can also be added from the Additional Options sub-section.

CrossCore GCC ARM Embedded C Linker

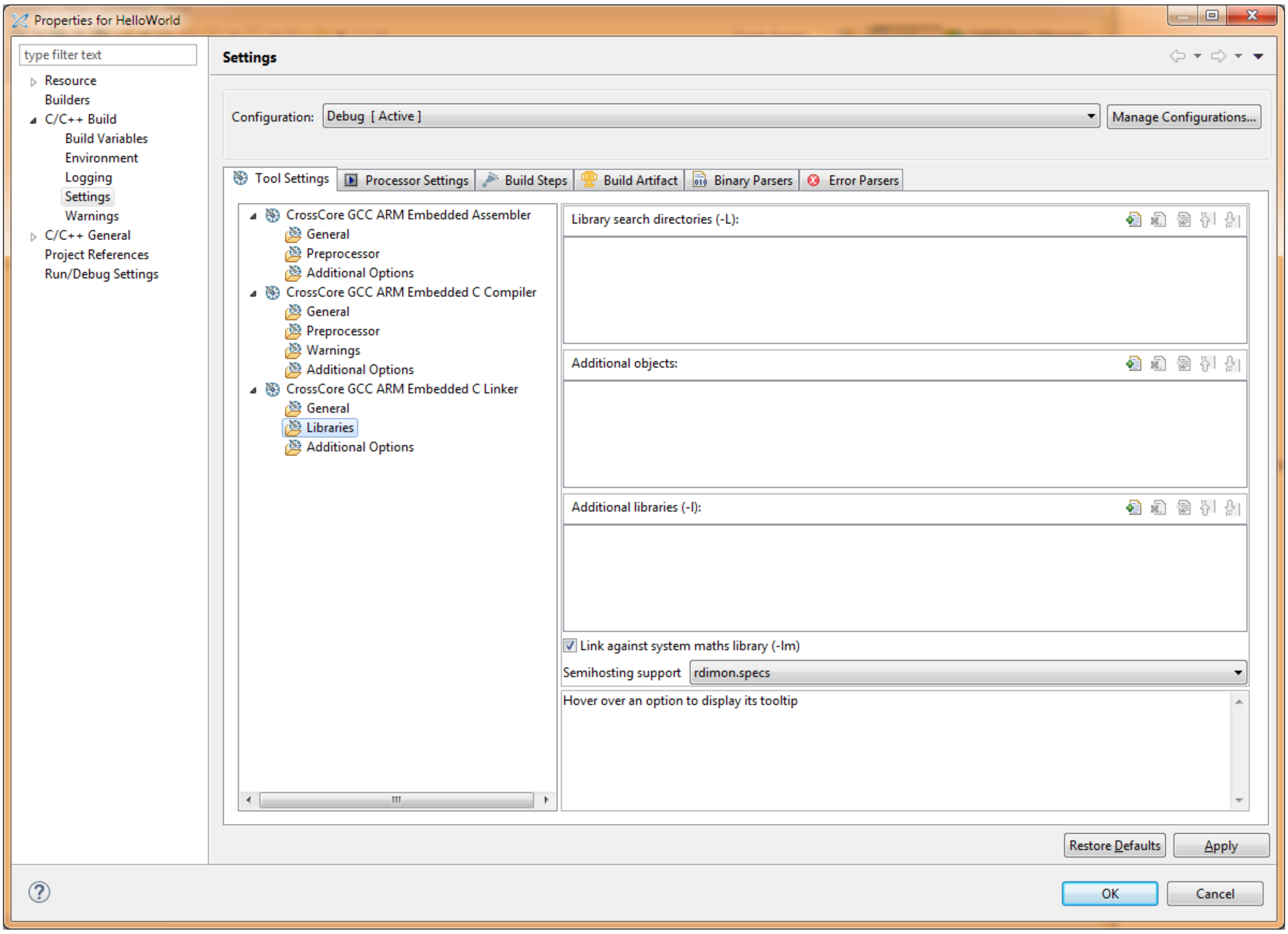
This section allows the user to set the path for the linker script file (.ld file), to set whether to use standard startup files or default libraries, and to specify whether to enable linker elimination by the CrossCore GCC ARM Embedded C Linker as shown in *Figure 11. CrossCore GCC ARM Embedded C Linker - General* below.

Figure 11. CrossCore GCC ARM Embedded C Linker - General



Library search directories, additional objects, additional libraries, system maths library and semi-hosting support (which supports I/O to the debugger console, by default it's set to rdimon.specs that enables functions in the C library, such as `printf()` and `scanf()`) can be set from Libraries sub-section as shown in *Figure 12. CrossCore GCC ARM Embedded C Linker - Libraries* below.

Figure 12. CrossCore GCC ARM Embedded C Linker - Libraries

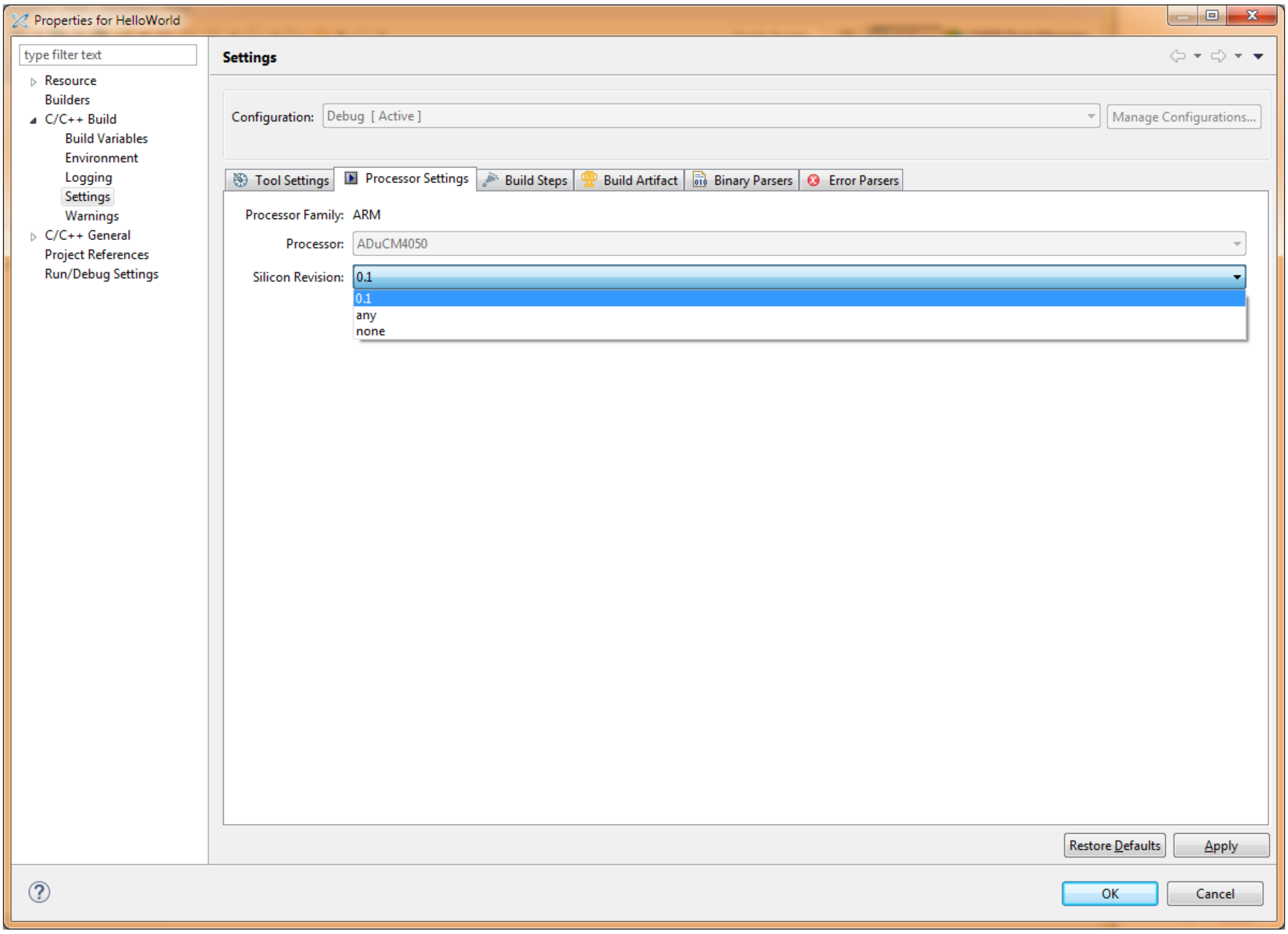


Additional linker options can also be added from the Additional Options sub-section.

3.2.2 Processor Settings

The Processor Settings tab displays the processor family, type, and silicon revision. The user can change the silicon revision of a project using the drop down list shown in *Figure 13. Processor Settings* below.

Figure 13. Processor Settings



Note that changing the silicon revision will regenerate source files such as *adi_initialize.{c|h}*.

The user cannot change the processor using this tab as you can with other ADI processors. Please refer to the *RTE Configuration - Device* section below on how to change the processor for CMSIS projects.

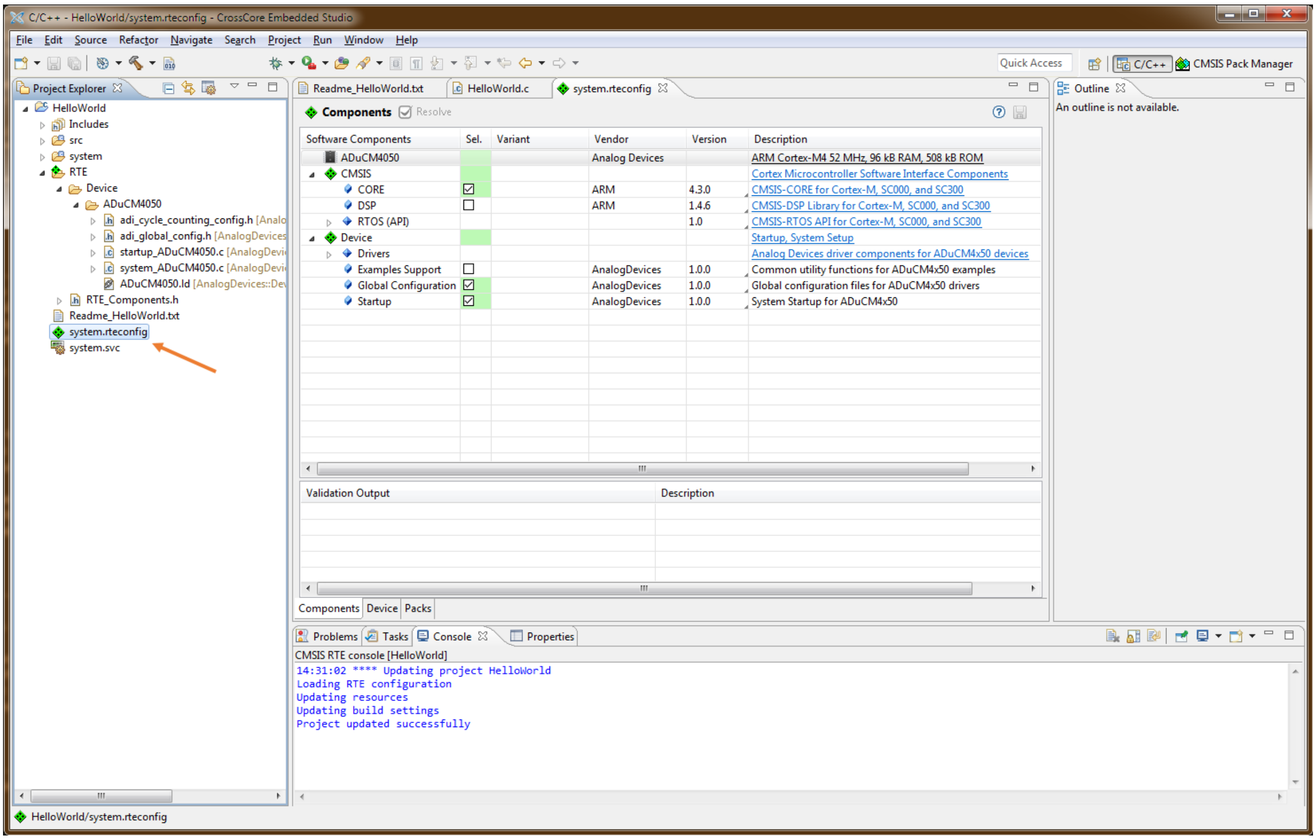
3.2.3 RTE Configuration

The Run-Time Environment (RTE) Configuration Editor allows the user to manage the software components used by the project, which device or processor the project targets, and let's the user specify which pack versions to use. You can open the RTE Configuration Editor by double-clicking the `system.rteconfig` file in the Project Explorer view. See the CMSIS C/C++ Development User's Guide in CCES On-line Help for further information.

`CMSIS` and `Device` components, including device drivers, are managed in `system.rteconfig` instead of `system.svc`. To add or remove a component, check or uncheck the "Sel." column next to the component in the `system.rteconfig` editor and save the file.

When creating a new project, be sure to add the `CMSIS-CORE` and `Startup` components manually as shown in *Figure 14. system.rteconfig* below. Both of these components are required to successfully build and run an application on ADuCM4x50 hardware.

Figure 14. system.rteconfig

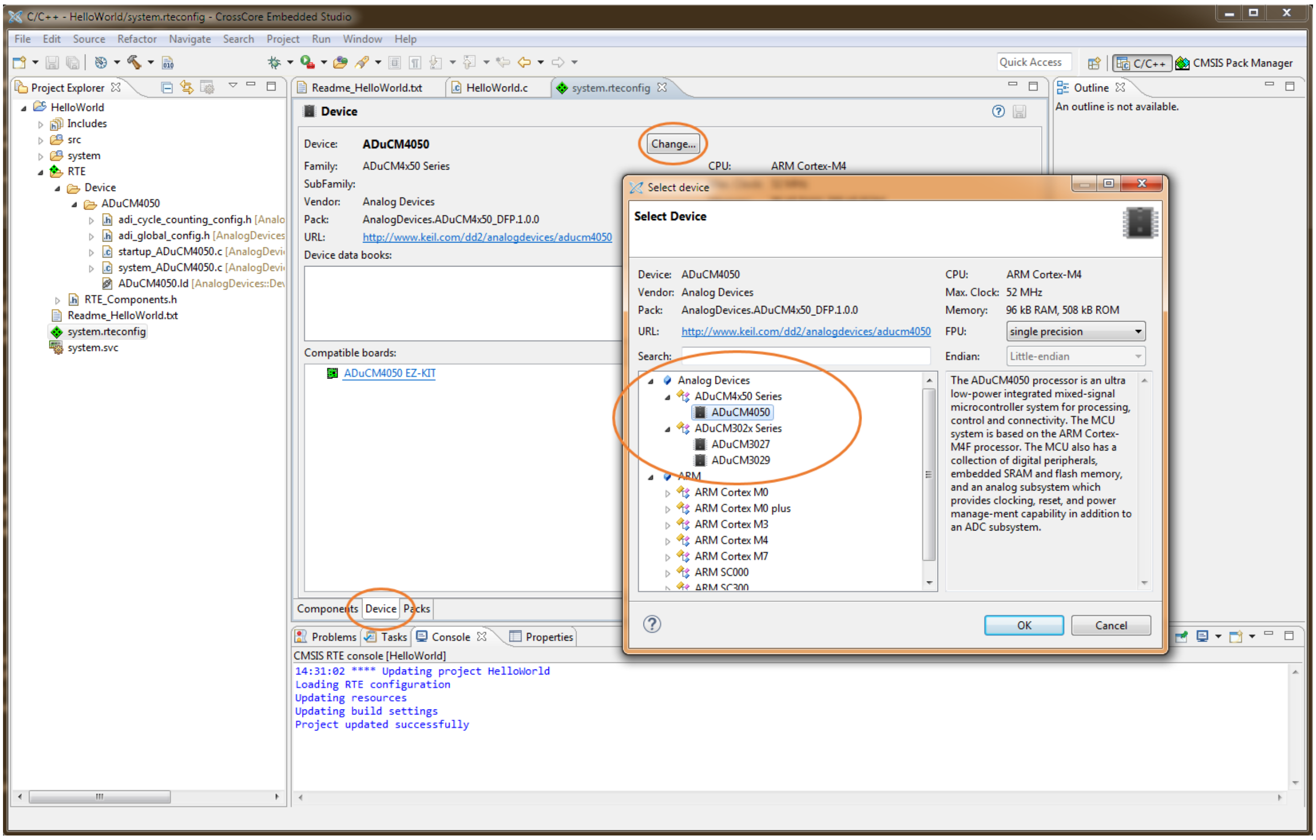


RTE Configuration - Device

The Device tab of the RTE Configuration Editor provides the following functionality:

- Shows information about the current selected device for the project.
- Access to the Software Pack URL and other documentation using hyperlinks.
- Provides `Change...` button to select a different device for the project.

Figure 15. RTE Configuration - Device



Note that changing the device will regenerate source files such as *adi_initialize.{c|h}* and will update the project's build settings.

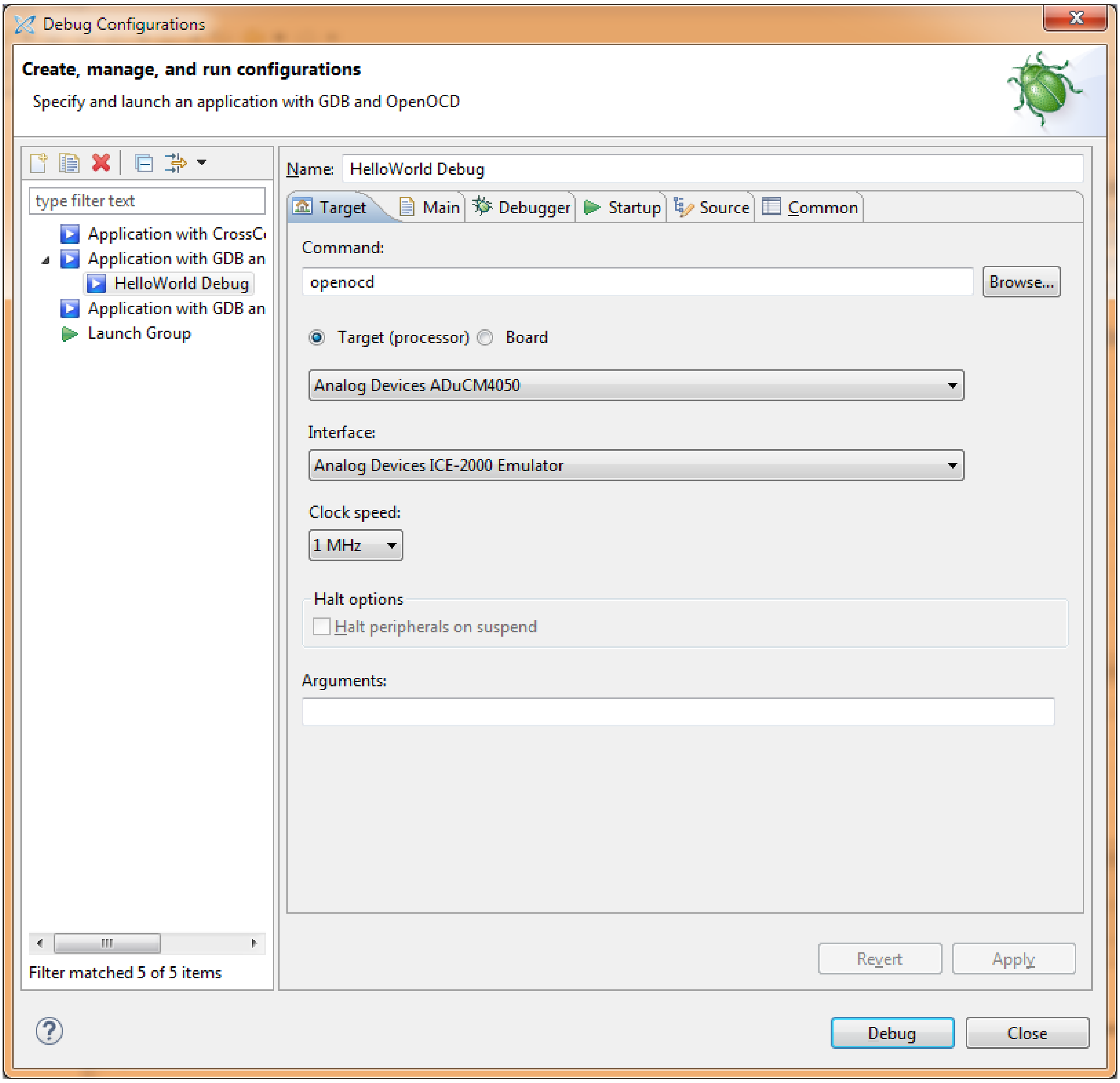
3.3 Debug Configurations

When creating a debug configuration for ADuCM4x50 processors, select "Application with GDB and OpenOCD". The Debug Configurations window allows you to select the Interface (ICE-1000 or ICE-2000), clock speed, to specify OpenOCD arguments or configure GDB.

3.3.1 Target

Create a debug configuration with "Application with GDB and OpenOCD". In the Target tab, select "Target (processor)", and select "Analog Devices ADuCM4050" in the drop-down list as shown in *Figure 16. Debug Configurations - Target* below.

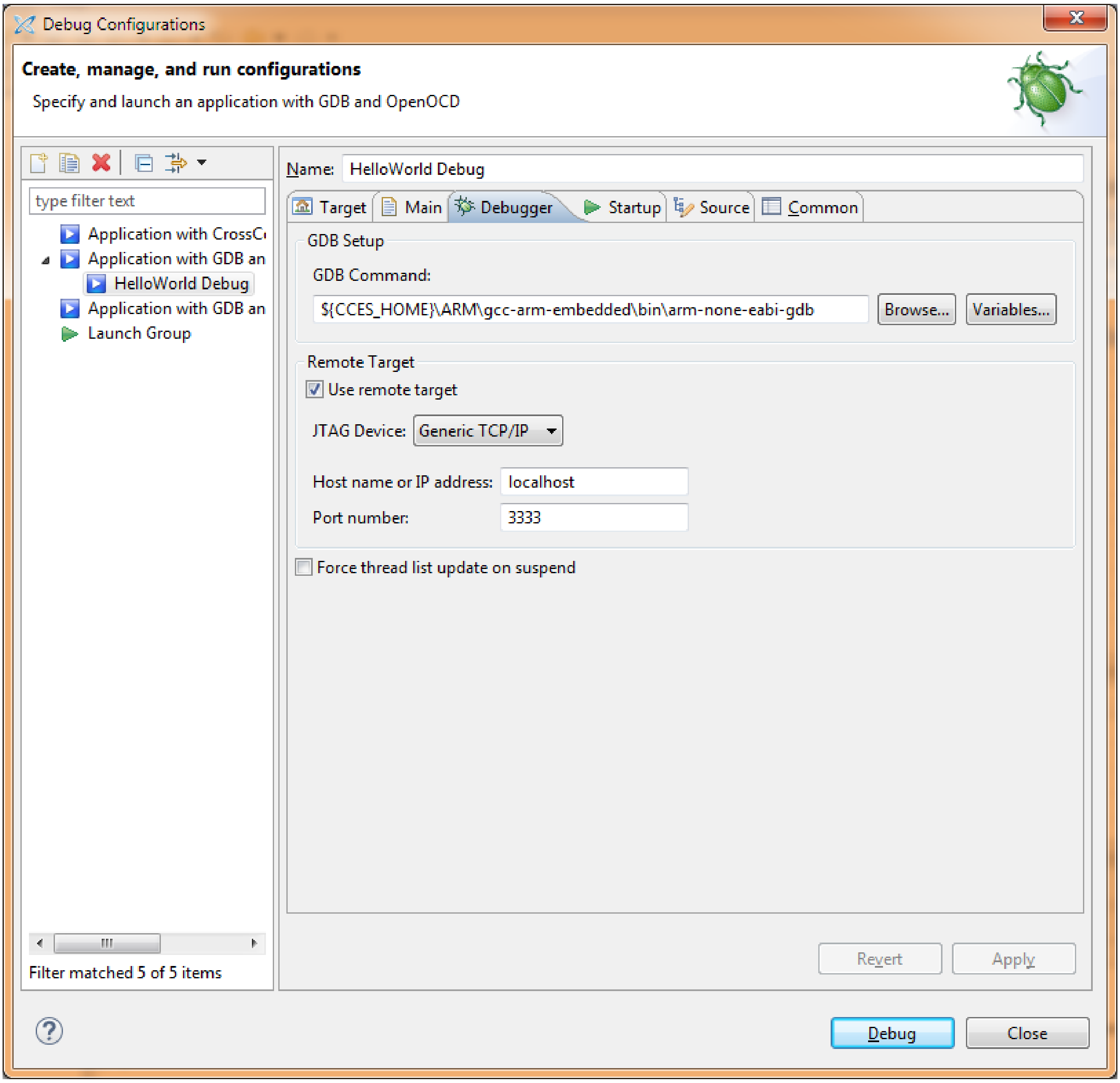
Figure 16. Debug Configurations - Target



3.3.2 Debugger

The Debugger tab allows you to configure GDB, as shown in *Figure 17. Debug Configurations - Debugger* below.

Figure 17. Debug Configurations - Debugger


Debug Configurations
Create, manage, and run configurations
Specify and launch an application with GDB and OpenOCD
type filter text
Application with CrossC
Application with GDB an
HelloWorld Debug
Application with GDB an
Launch Group
Filter matched 5 of 5 items
Name: HelloWorld Debug
Target
Main
Debugger
Startup
Source
Common
GDB Setup
GDB Command:
${CCES_HOME}\ARM\gcc-arm-embedded\bin\arm-none-eabi-gdb
Browse...
Variables...
Remote Target
Use remote target
JTAG Device: Generic TCP/IP
Host name or IP address: localhost
Port number: 3333
Force thread list update on suspend
Revert
Apply
Debug
Close

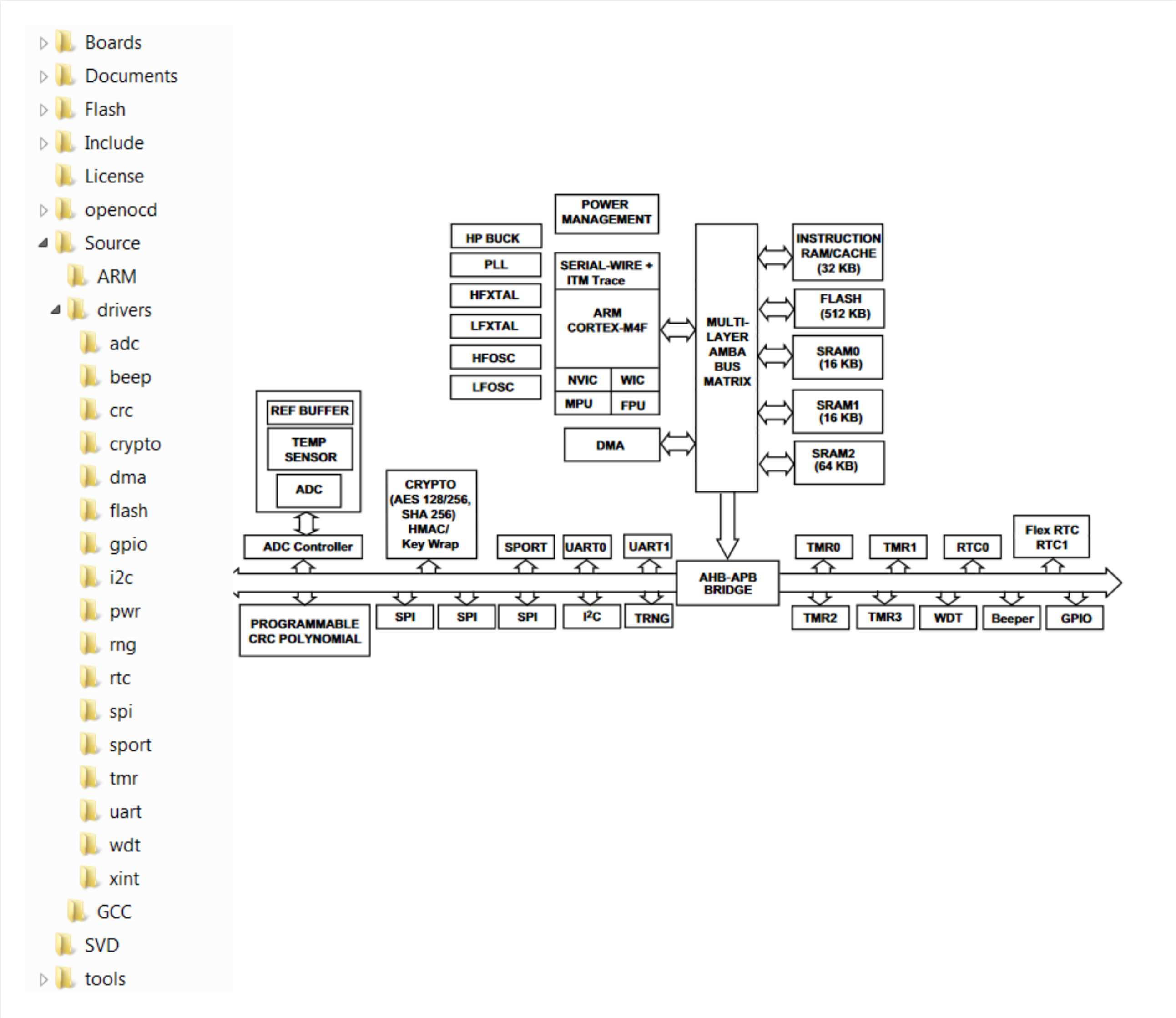
4 ADuCM4x50 System Overview

4.1 Block Diagram and Driver Layout

The Peripheral Device Drivers installed with this software package are used to configure and use various ADuCM4x50 on-chip peripherals. *Figure 18. Peripherals and Driver Source* illustrates the available peripherals and interconnects on the ADuCM4x50 processor and corresponding source files in the **<ADuCM4x50_root>**/Source directory.

In general, the driver sources are located in the **<ADuCM4x50_root>**/Source directory. The include file path **<ADuCM4x50_root>**/Include must also be specified as additional include directories in the project's compiler (see section `CrossCore GCC ARM Embedded C Compiler – Preprocessor` in `Installation Components` of this documentation).

Figure 18. Peripherals and Driver Source



4.2 Boot-Time CRC Validation

The ADuCM4x50 system reset interrupt vector is hard-coded on-chip to execute a built-in pre-boot kernel that performs a number of critical housekeeping tasks before executing the user provided reset vector. Some of those tasks include initializing the JTAG/Serial-Wire debug interface and validating flash integrity.

One of the primary tasks of the pre-boot kernel is to validate the integrity of the Flash0/Page0 region (first 2k of flash). This is done by comparing a pre-generated CRC code (embedded in the executable code image at build-time) against a boot-time-generated CRC value of Flash0/Page0 using the on-chip CRC hardware. The Page0 embedded CRC signature is stored at reserved location 0x000007FC.

ADuCM4x50 CCES support does not currently compute and implant the CRC signature into the executable during the target build process (as a post-link build command). Therefore at boot time, when the kernel computes a run-time Flash0/Page0 CRC value using the on-chip CRC hardware and compares it against the value at the last CRC page, by default applications are built to omit the CRC check.

5 Application Configuration

Application initialization and configuration will vary depending on the chosen operating mode. The modes of operation include:

- Non-RTOS
  - The application is built without an RTOS.
- RTOS
  - The application is built with an RTOS. In this mode of operation the drivers can be RTOS-Aware or RTOS-Unaware.
  - RTOS-Aware Drivers
    - In this C-Macro controlled mode of operation the driver's source code will include the following features:
      - Interrupt Service Routines (ISR) with RTOS API calls used to potentially cause a task context switch.
      - Semaphores control communication between task-level code and ISR level code.
      - Mutexes control access to access to shared resources.
  - RTOS-Unaware Drivers.
    - In this C-Macro controlled mode of operation the driver's source code will not include the features listed above.

5.1 Application Initialization

The function `adi_initComponents()` is used to initialize an application. `adi_initComponents()` is required to initialize the managed drivers that have been added to the project, in which `adi_initpinmux()` is required to initialize the peripheral pin multiplexing, if static pin multiplexing is used (see section `Static Pin Multiplexing` in `Build Configurations`). *Figure 19. Application Initialization* and *Figure 20. adi_initComponents()* below shows `adi_initComponents()` being called from the user application `main()` and `adi_initpinmux()` being called from `adi_initComponents()`.

Figure 19. Application Initialization

```
int main(int argc, char *argv[])
{
```

```
    /**
     * Initialize managed drivers and/or services that have been
added to
     * the project.
     * @return zero on success
     */
    adi_initComponents();

    /* Begin adding your custom code here */
    return 0;
}
```

Figure 20. adi_initComponents()

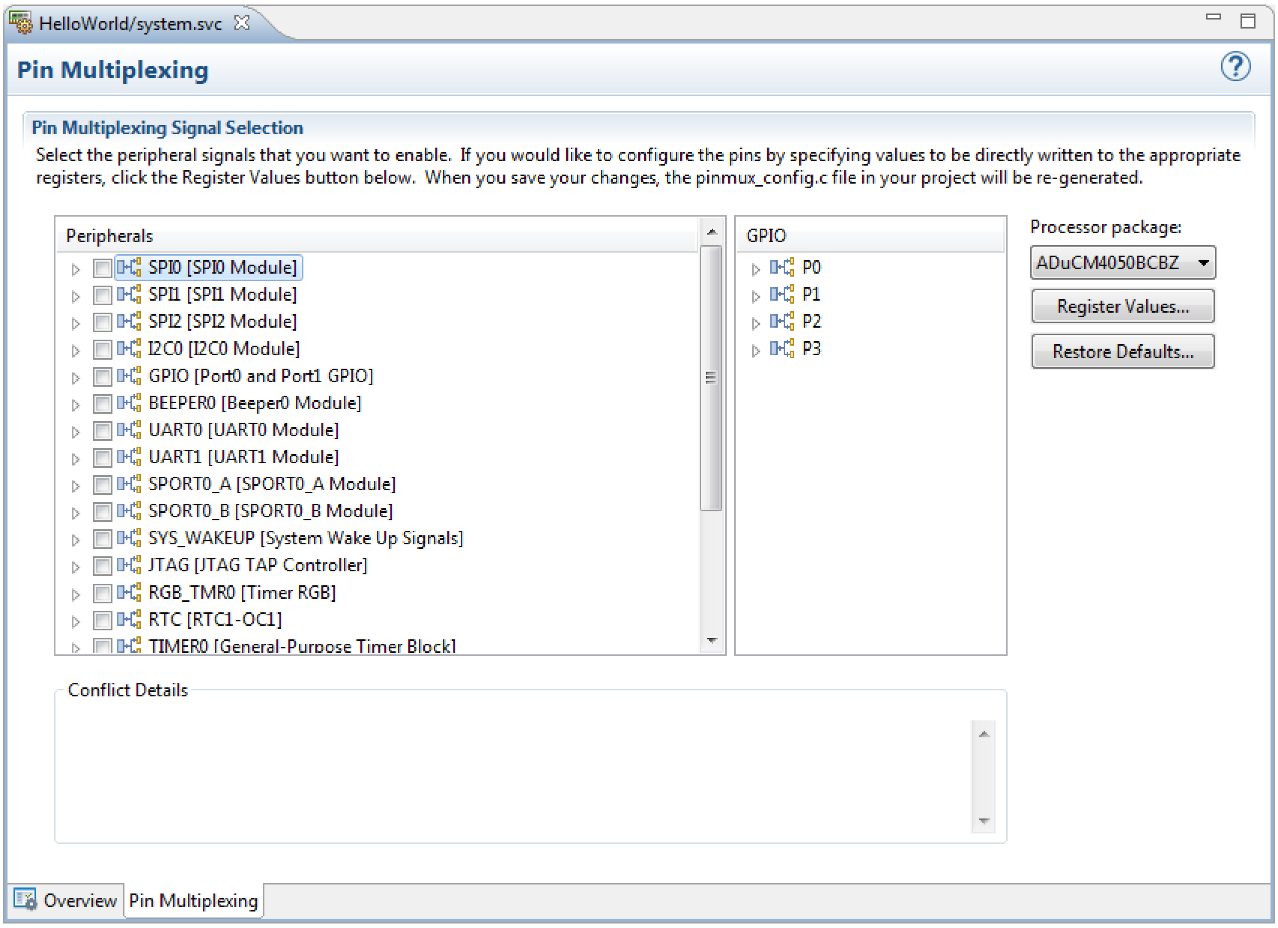
```
int32_t adi_initComponents(void)
{
    int32_t result = 0;

    if (result == 0) {
        result = adi_initpinmux(); /* auto-generated code (order:0)
*/
    }
    return result;
}
```

5.2 Static Pin Multiplexing

The Pin Multiplexing Add-in is recommended by default when creating a CCES project for ADuCM4x50; the Add-in is capable of generating code to set all the port MUX registers statically for all peripherals in a single call. It also can be added to the project from `system.svc`.

After adding the Pin Multiplexing Add-in, you can configure the add-in to generate C source by opening the system.svc file and selecting the `Pin Multiplexing` tab. The add-in will not allow conflicting peripherals to be selected. Saving the system.svc file after selecting signals will automatically generate a C source file that sets the GPIO port configuration registers based on the peripherals and functions selected. The C source file is automatically added to your project as `pinmux_config.c` under your project's `system/pinmux/GeneratedSources` directory. The C source file has a function `adi_initpinmux()` which can be called from `adi_initComponents()` (see *Figure 21. Pin Multiplexing Add-in*) to set up the port MUX registers.

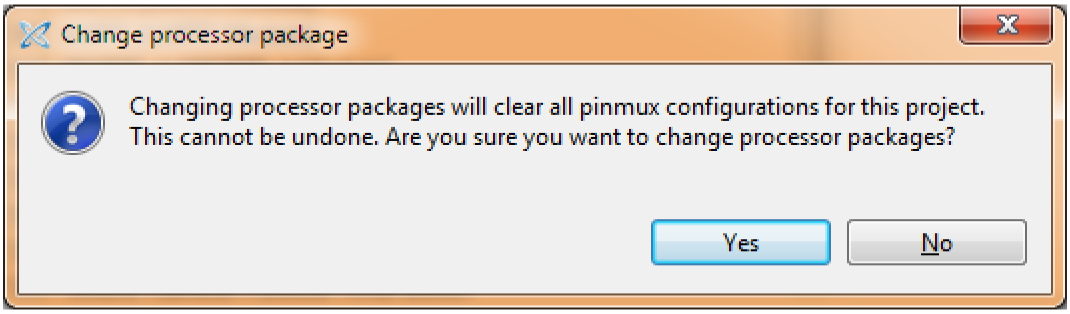
Figure 21. Pin Multiplexing Add-in



The ADuCM4x50 processors have two package variants: the BCBZ and BCPZ, each with different peripheral configurations. The "Processor package" drop down list lets you specify which package to use when generating the pinmux configuration file. Note that changing processor packages will clear the pinmux configuration, as shown in the dialog in *Figure 22. Change Processor Package Dialog* below.

Note: The pinmux code generator is the preferred method of configuring port multiplexing. It avoids multiple dynamic calls to each driver and allows all pin multiplexing to be done once, which reduces both footprint and run-time overhead.

Figure 22. Change Processor Package Dialog



5.3 UART Baud Rate Configuration Utility

Included with the ADuCM4x50 DFP is a `UartDivCalculator` utility which computes the baudrate configuration values for a specified clock. This helps you to statically configure the baudrate. The utility can also provide the baudrate configuration values for a set of baudrates.

The output can be used in the UART baudrate configuration API, as

```
ADI_UART_RESULT adi_uart_ConfigBaudRate( ADI_UART_HANDLE const
hDevice, uint16_t nDivC, uint8_t nDivM, uint16_t nDivN, uint8_t nOSR
);
```

Where:

- **hDevice** is the device handle to UART device obtained when an UART device opened successfully
- **nDivC** is `DIV-C` in the output of `UartDivCalculator` utility
- **nDivM** is `DIV-M` in the output of `UartDivCalculator` utility
- **nDivN** is `DIV-N` in the output of `UartDivCalculator` utility
- **nOSR** is `OSR` in the output of `UartDivCalculator` utility

To produce the baudrate configuration values for a specified clock and the whole set of baudrates:

```
UartDivCalculator.exe input_clock
```

To produce the baudrate configuration values for a specific clock and a particular baudrate:

```
UartDivCalculator.exe input_clock baudrate
```

For example, to get the baudrate configuration values for input clock 16 MHz and baudrate 9600, run the following command line:

```
UartDivCalculator.exe 16000000 9600
```

and you will get

```
+------------------------------------------------------------------+
| CALCULATING UART DIV REGISTER VALUE FOR THE  INPUT CLOCK: 16000000     |
|------------------------------------------------------------------|
|       BAUDRATE       DIV-C     DIV-M     DIV-N        OSR    DIFF      |
|------------------------------------------------------------------|
|       00009600        0014      0003      1475        0003    0000      |
|------------------------------------------------------------------|
```

5.4 Driver Include Files

In Tool Settings of the Properties for a project, the Additional include directories (-I) section of the CrossCore GCC ARM Embedded C Compiler > Preprocessor tab is used to define the additional include directories needed to build the project. The device drivers only require the following search paths

<ADuCM4x50_root>/Include

<ARM_CMSIS_root>/CMSIS/Include

to be added from the DFP installation. The required include directories, including the ones described above, are added to the Tool Settings automatically by the CMSIS components. See the *Installation Components - RTE Configuration* section in this document for more information. Applications may need to augment the preprocessor search path with their own requirements.

5.5 Driver Configuration

Most of the drivers are statically configurable. Their configuration is controlled via C/C++ pre-processor macros that are managed in a common area.

Static initialization is preferred, as it offers two advantages over dynamic (API) initializations:

1. It reduces the run-time driver startup time (and complexity) of initializing each driver through various driver configuration APIs.
2. It allows programmers to bypass most of the driver configuration APIs altogether, thereby allowing linker elimination to remove unused driver APIs, thereby reducing overall footprint.

5.5.1 Global Configuration

There is a single, global configuration file <ADuCM4x50_root>/Include/config /adi_global_config.h, that is added to your project's RTE directory automatically by adding the Global Configuration CMSIS component via the system.rteconfig file. We recommend using the same approach for overriding the driver-specific configuration files as described below to override the global feature set-up. For example, to overwrite the RTOS feature, set the corresponding macro in `adi_global_config.h` as shown in *Figure 23. Global Configuration File Contents* below:

Figure 23. Global Configuration File Contents

```
/*! Configure the RTOS required across the project.
    It can be configured to one of the following macros:
    - #ADI_CFG_RTOS_NO_OS
    - #ADI_CFG_RTOS_MICRIUM_II
    - #ADI_CFG_RTOS_MICRIUM_III
    - #ADI_CFG_RTOS_FREERTOS
 */
#define ADI_CFG_RTOS    ADI_CFG_RTOS_NO_OS
```

5.5.2 Configuration Defaults

Two distinct types of configurations are managed in the driver configuration files: feature/function enable/disable (such as removing unneeded code for slave-mode operation, DMA support, etc.) and default values for the peripheral control registers. Each device driver uses these macros to control feature inclusion and set controller registers during driver initialization.

Factory default driver configuration files (one per driver) are located in the <ADuCM4x50_root>/Include/config directory, which you can edit for global changes or override them within your project, for localized changes.

It is recommended that the default configuration files are not modified, but overridden by modifying the local copies within your project found in your project's RTE directory. These copies are added by the CMSIS device driver components.

5.5.3 Configuration Overrides

In summary, there are two ways to override the default static configuration values:

1. Locally by editing the configuration files in your project's RTE directory.
2. Dynamically by using the dynamic APIs to modify the configuration at run time. The configuration APIs may be called at run-time to alter a driver’s configuration. Static configuration is preferred, however, as it will save both footprint and run-time cycles.

Please note that a combination of static and dynamic configuration is possible.

5.5.4 IVT Table Location

The Cortex-M4 processor core allows the Interrupt Vector Table (IVT) to be relocated. In this release, we support a default placement of the IVT in ROM (FLASH) and allow it to be moved from ROM to RAM during system startup. The pre-processor macro RELOCATE_IVT is used enable IVT relocation.

The default, statically-linked IVT placement in ROM is preferred as it will avoid wasting RAM space and startup time to relocate the table. The static IVT cannot be used if the application needs to alter the IVT content.

Alternatively, the IVT may be dynamically relocated during system startup from ROM to RAM for applications that need to modify the IVT content. For example, by dynamically hooking/replacing interrupt handlers or running an RTOS that requires patching interrupt handlers through a common interrupt dispatcher. To support dynamic IVT relocation, add the RELOCATE_IVT macro to the compiler pre-processor option tab. Doing so causes the IVT to be relocated during system reset (see startup.c: ResetISR() handler).

The default static IVT is always present in ROM and is optionally copied to RAM under control of the RELOCATE_IVT macro. See relevant source code in system files startup.c and system.c (enclosed within the RELOCATE_IVT macro) for implementation details of the relocated IVT memory allocation, relocation address and alignment attributes, physically copying the IVT and updating the *interrupt vector table offset register* (VTOR) within the Cortex-M4 core System Control Block (SCB). Once the IVT is copied and VTOR is written with the new address, the relocated interrupt vectors are active and can then be modified dynamically.

5.5.5 Interrupt Callbacks

In general, the device drivers take ownership of the various device interrupt handlers in order to drive communication protocols, manage DMA data pumping, capture events, etc. Most device drivers also offer application-level interrupt callbacks by giving the application an opportunity to receive event notifications or perform some application-level task related to device interrupts.

Application callbacks are optional. They may be an integral component of an event-based system or they may just tell the application when something happened. Application callbacks are always made in response to device interrupts and are *executed in context of the originating interrupt*.

To receive interrupt callbacks, the application defines a callback handler function and registers it with the device driver. The callback registration tells the device driver what application function call to make as it processes device interrupts. Each driver has unique event notifications which are passed back with the callback and describe what caused the interrupt. Some device drivers support event filtering that allows the application to specify a subset of events upon which to receive callbacks.

To use callbacks, the application defines a callback handler with the following prototype:

```
void cbHandler (void *pcbParam, uint32_t Event, void *pArg);
```

Where:

- **pcbParam** is an application-defined parameter that is given to the device driver as part of the callback registration,
- **Event** is a device-specific identifier describing the context of the callback, and
- **pArg** is an optional device-specific argument further qualifying the callback context (if needed).

The application will then call into the device driver callback registration API to register the callback, as:

```
ADI_xxx_RESULT_TYPE adi_xxx_RegisterCallback (ADI_xxx_DEV_HANDLE cons
t hDevice, ADI_CALLBACK const pfCallback, void *const pcbParam);
```

Where:

- **xxx** is the particular device driver,
- **hDevice** is the device driver handle,
- **ADI_CALLBACK** is a typedef (see `adi_int.h`), describing the callback handler prototype (cbHandler, in this case),
- **pfCallback** is the function address of the application's callback handler (cbHandler), and
- **pcbParam** is an application-defined parameter that is passed back to the application when the application callback is dispatched. This parameter is used however the application dictates, it is simple passed back through the callback to the application by the device driver as-is. It may be used to differentiate device drivers (e.g., the device handle) if multiple drivers or driver instances are sharing a common application callback.

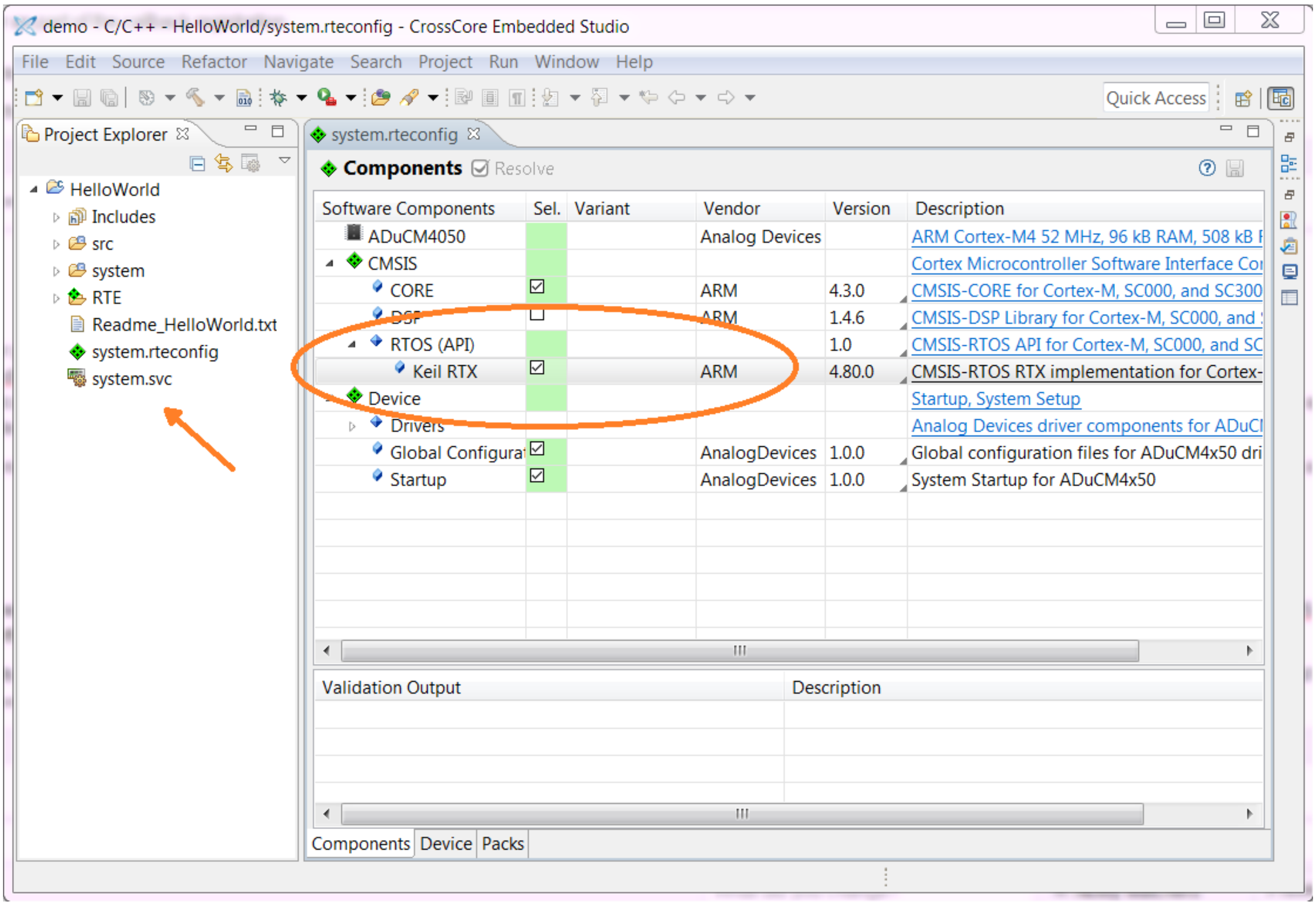
Note: Application callbacks occur in context of the originating device interrupt, so extended processing at the application level will impact interrupt dispatching. Typically, extended application-level processing is done by some task after the callback is returned and the interrupt handler has exited.

5.6 CMSIS-RTOS Keil RTX

CMSIS-RTOS Keil RTX is an implementation of the CMSIS-RTOS API, which is a generic RTOS interface for Cortex-M processor-based devices. The CMSIS-RTOS API provides a standardized interface for software components that require RTOS functionality. Refer to *Real-Time Operating System: API and RTX Reference Implementation* for further information.

The component can be added to a project from `system.rteconfig.` The components `CMSIS-CORE`, `Global configuration` and `Startup` also need to be added to the project. In order to build successfully, the following options need to be added to the `Additional Options` of the `assembler/compiler/linker`. Refer to `Installation Components > CCES Project Options > Tool Settings` in this documentation, regarding how to add additional options to tools.

```
-mfloat-abi=hard -mfpu=fpv4-sp-d16
```



6 Examples

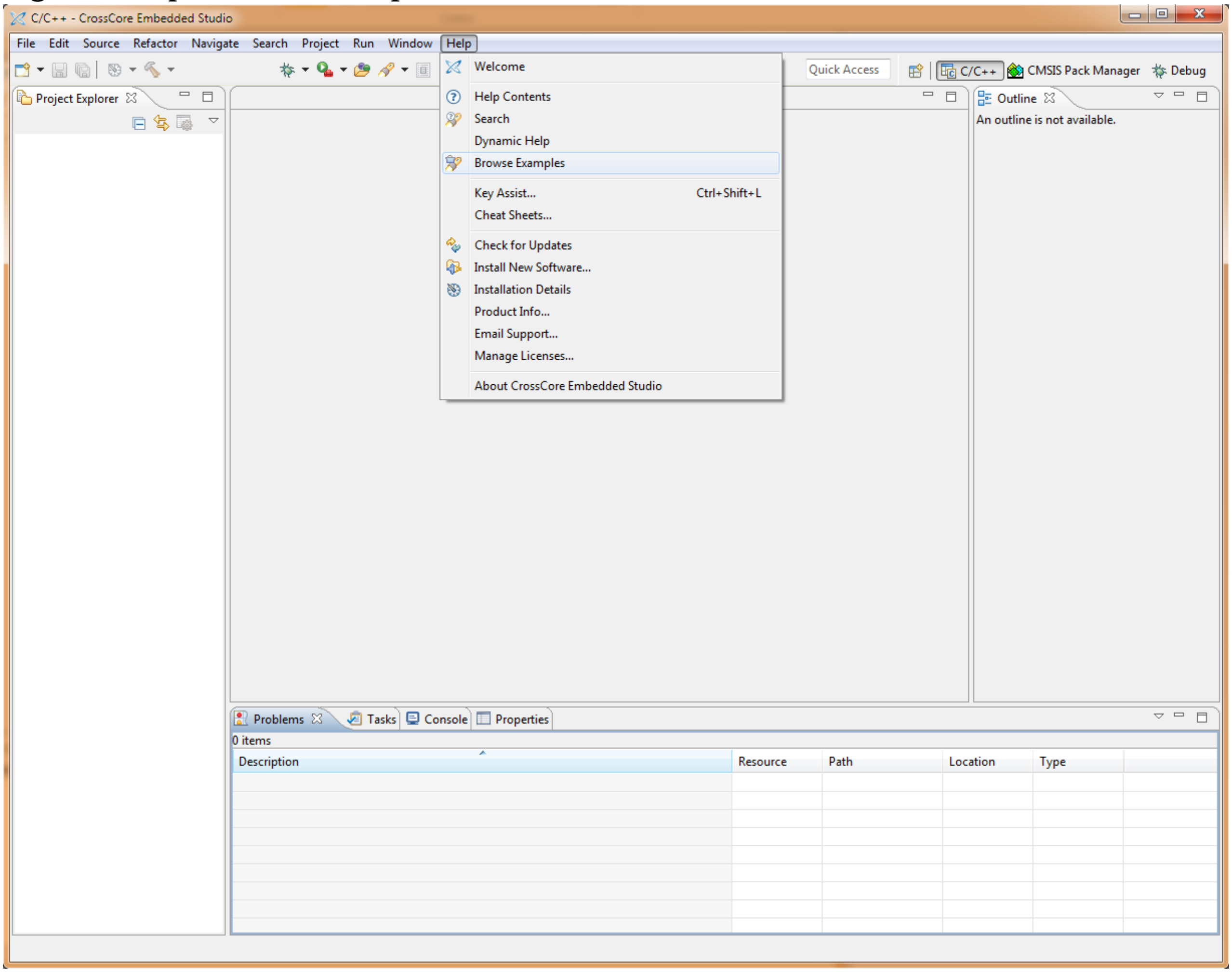
The ADuCM4x50 Device Family Pack contains example programs illustrating how to use the ADuCM4x50 device and software device drivers. Use these example programs to explore the hardware and recommended use of the device drivers. Each project has a Readme.txt file that describes any required switch settings, hardware setups, build/run steps, and expected outcome. Use the Console View to capture example output.

The examples are located in directory <ADuCM4x50_root>/Boards/ADuCM4050-EZ-KIT. The device driver sources, include files, example projects, etc., are all device-specific and ported to the specific evaluation board, which are located under the **<ADuCM4x50_root>** directory. The evaluation board is the only target on which the examples are supported.

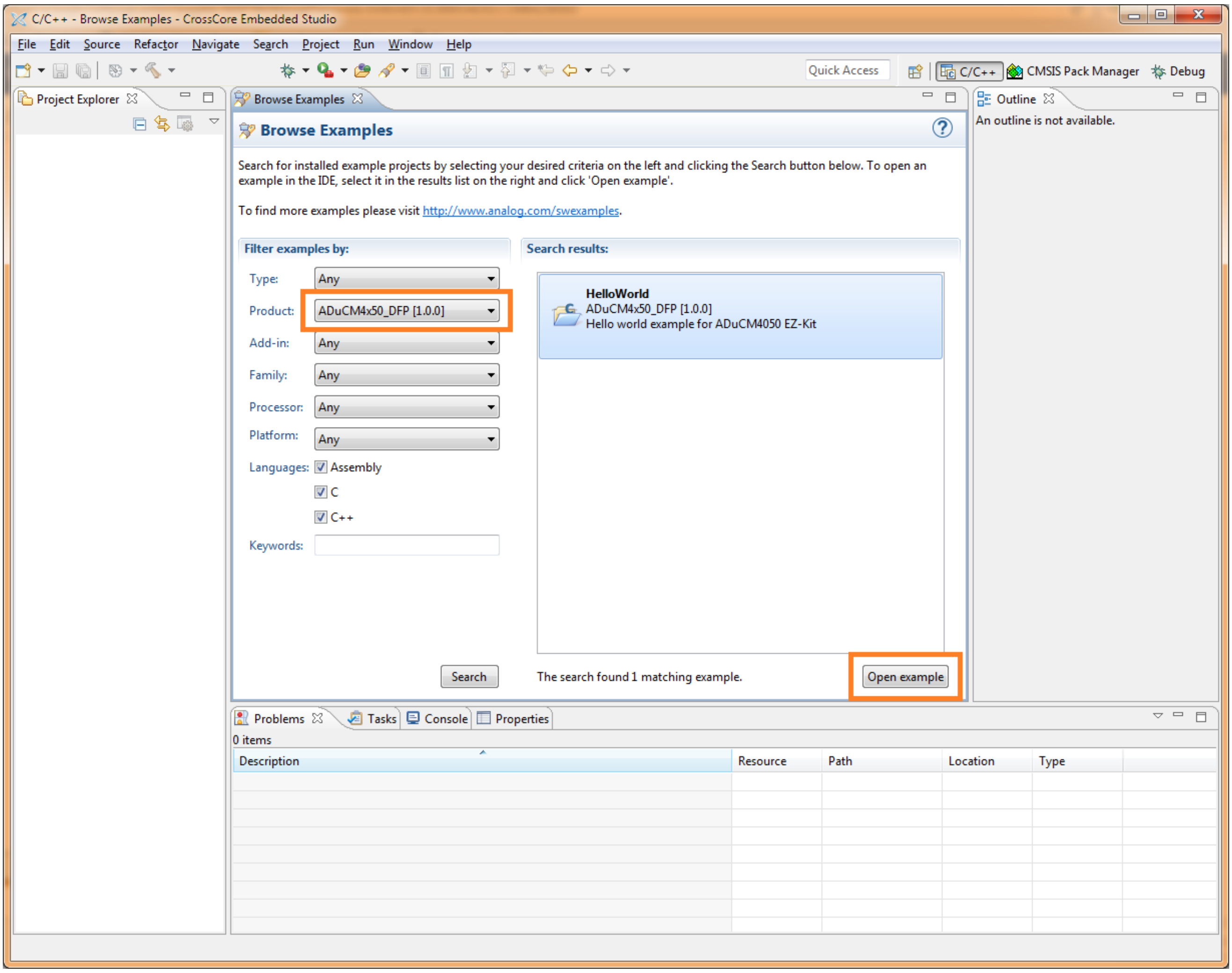
6.1 Examples Browser

The CCES Examples Browser can be used to open examples via Help > Browse Examples. See *Figure 24. Open CCES Examples Browser*. *(The version of the Device Family Pack used in the following description is 1.0.0, the first version released. Simply replace this version number with the one you want to use if you installed a more recent Device Family Pack.)*

Figure 24. Open CCES Examples Browser



Select `ADuCM4x50_DFP [1.0.0]` in the Product drop-down list, select the example you wish to open and click `Open example`. The example will then be copied into your workspace.

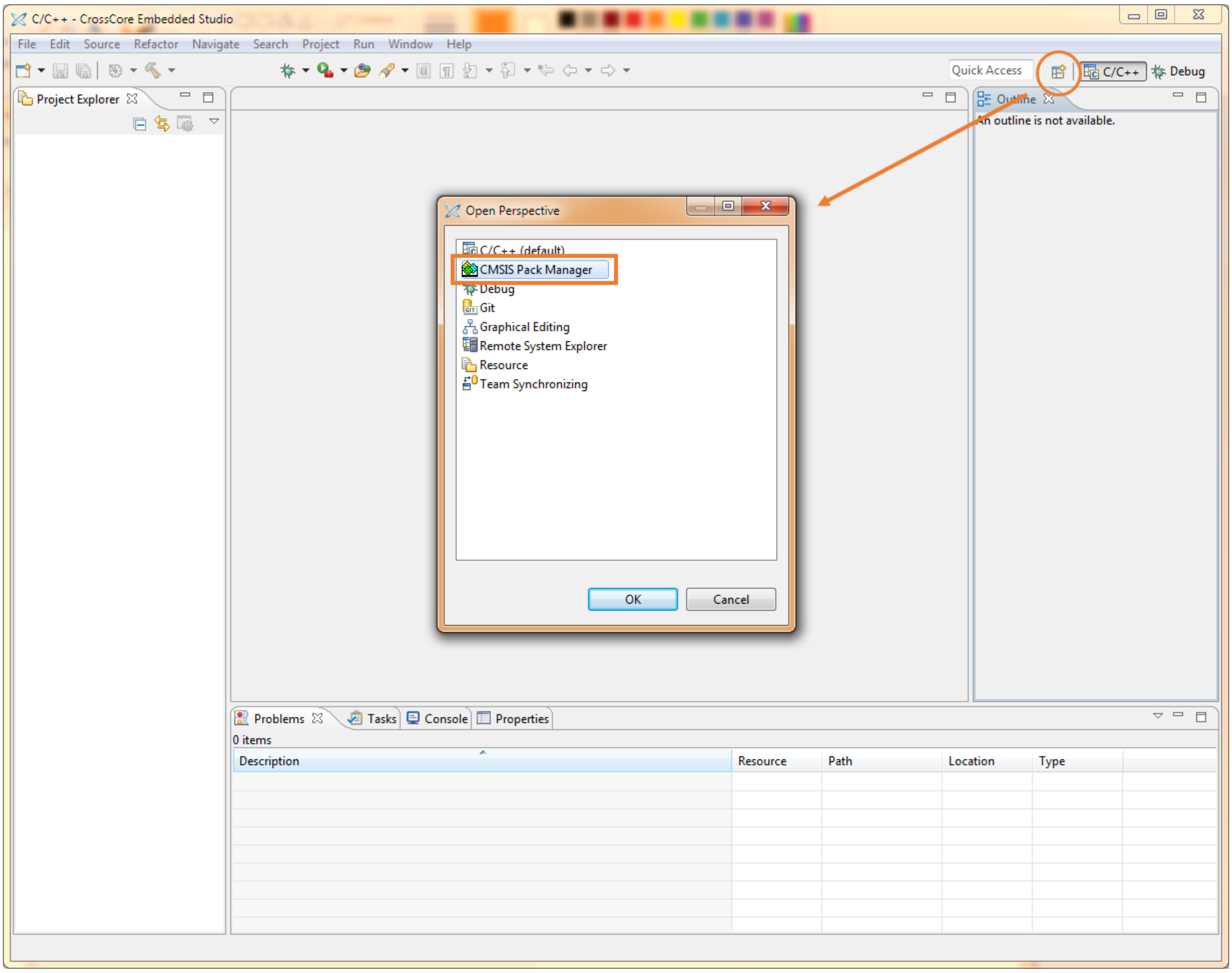


6.2 CMSIS Examples View

You can also open example projects from the CMSIS Pack Manager's Examples view.

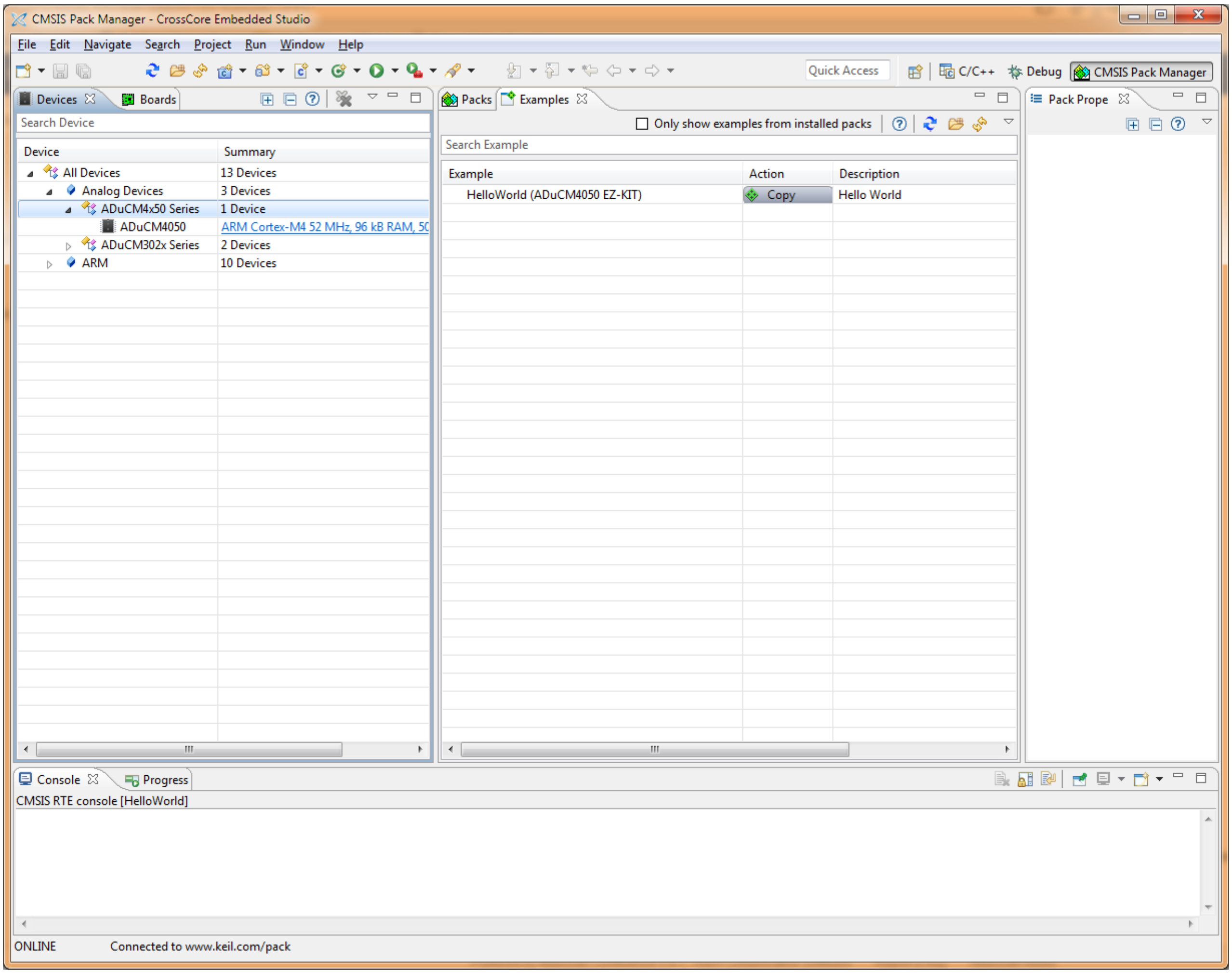
Open the CMSIS Pack Manager perspective by click the "Open Perspective" icon on the toolbar and selecting "CMSIS Pack Manager" in the Open Perspective window as shown in *Figure 25. Open CMSIS Pack Manager Perspective*.

Figure 25. Open CMSIS Pack Manager Perspective



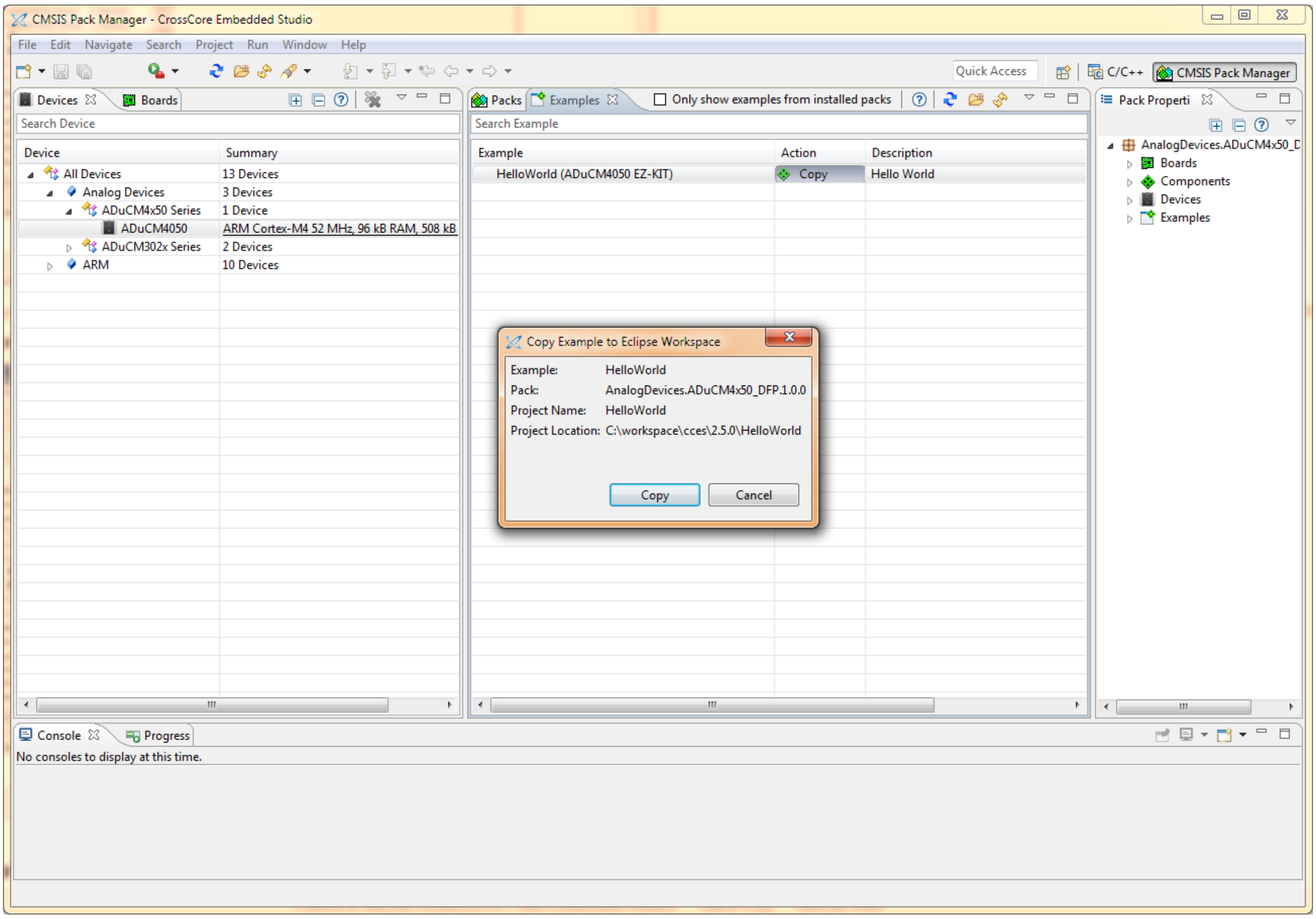
In the Examples tab, the examples in all installed DFPs (including ADuCM4x50 DFP) will be shown. To view examples for a particular device or device family, select the device or family in the Devices tab as shown in *Figure 26. Examples view*.

Figure 26. Examples view



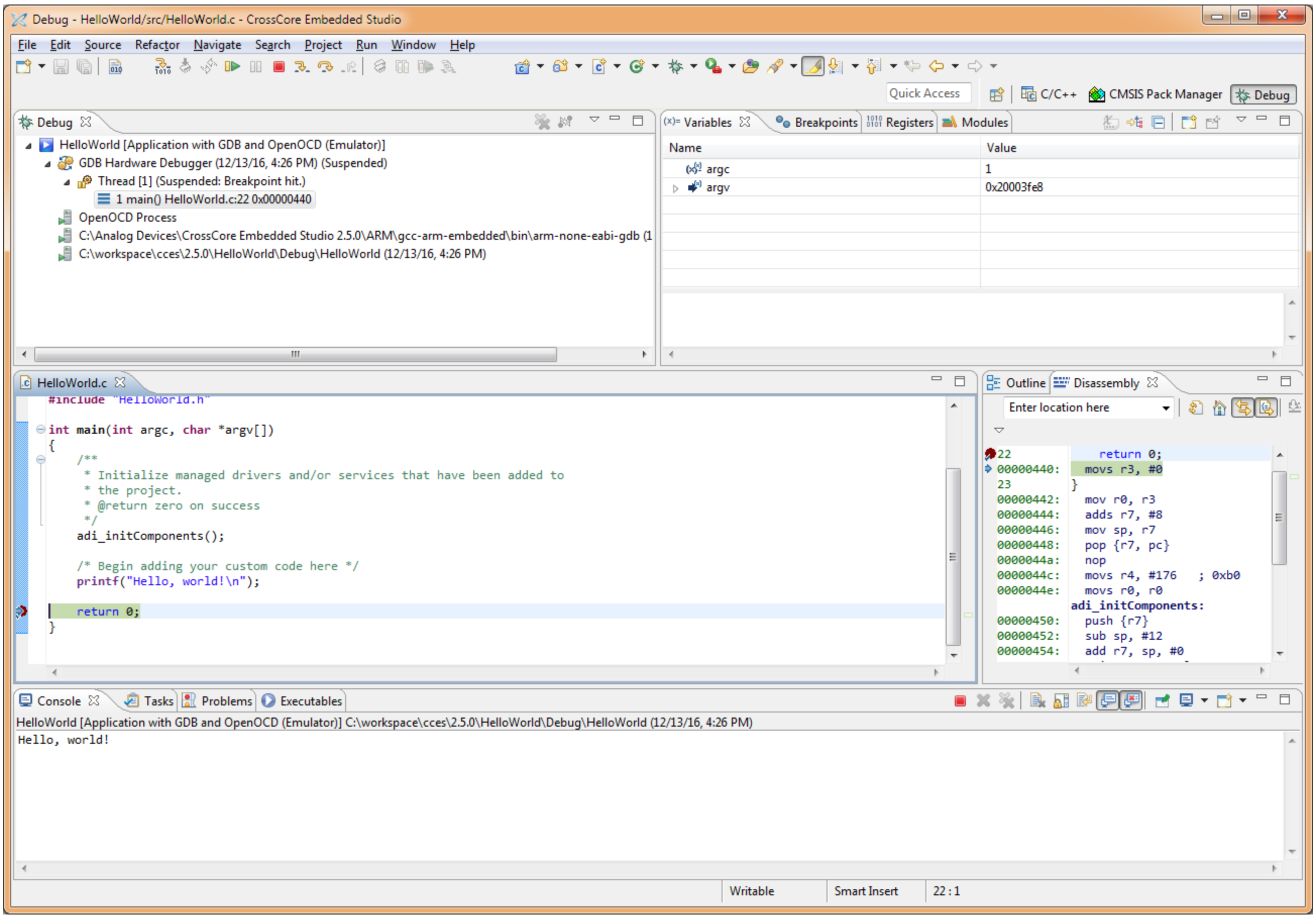
Click `Copy` in the `Action` column to select an example and click `Copy` in the confirmation dialog to import the example into your workspace as shown in *Figure 27. Copy Example to CCES Workspace*.

Figure 27. Copy Example to CCES Workspace



6.3 Build and Run

Build the project and launch the debugger (see the *Debug Configurations* section in this document for more information) to download and run the executable to the target. Follow the steps in the Readme.txt included in the example project to setup the hardware.


Debug - HelloWorld/src/HelloWorld.c - CrossCore Embedded Studio
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
C/C++
CMSIS Pack Manager
Debug
HelloWorld [Application with GDB and OpenOCD (Emulator)]
GDB Hardware Debugger (12/13/16, 4:26 PM) (Suspended)
Thread [1] (Suspended: Breakpoint hit.)
1 main() HelloWorld.c:22 0x00000440
OpenOCD Process
C:\Analog Devices\CrossCore Embedded Studio 2.5.0\ARM\gcc-arm-embedded\bin\arm-none-eabi-gdb
C:\workspace\cces\2.5.0\HelloWorld\Debug\HelloWorld (12/13/16, 4:26 PM)
Variables
Breakpoints
Registers
Modules
Name
Value
argc
1
argv
0x20003fe8
HelloWorld.c
#include "HelloWorld.h"
int main(int argc, char *argv[])
{
/**
* Initialize managed drivers and/or services that have been added to
* the project.
* @return zero on success
*/
adi_initComponents();
/* Begin adding your custom code here */
printf("Hello, world!\n");
return 0;
}
Outline
Disassembly
Enter location here
22 return 0;
00000440: movs r3, #0
23 }
00000442: mov r0, r3
00000444: adds r7, #8
00000446: mov sp, r7
00000448: pop {r7, pc}
0000044a: nop
0000044c: movs r4, #176 ; 0xb0
0000044e: movs r0, r0
adi_initComponents:
00000450: push {r7}
00000452: sub sp, #12
00000454: add r7, sp, #0
Console
Tasks
Problems
Executables
HelloWorld [Application with GDB and OpenOCD (Emulator)] C:\workspace\cces\2.5.0\HelloWorld\Debug\HelloWorld (12/13/16, 4:26 PM)
Hello, world!
Writable
Smart Insert
22 : 1

7 Device Driver API Documentation

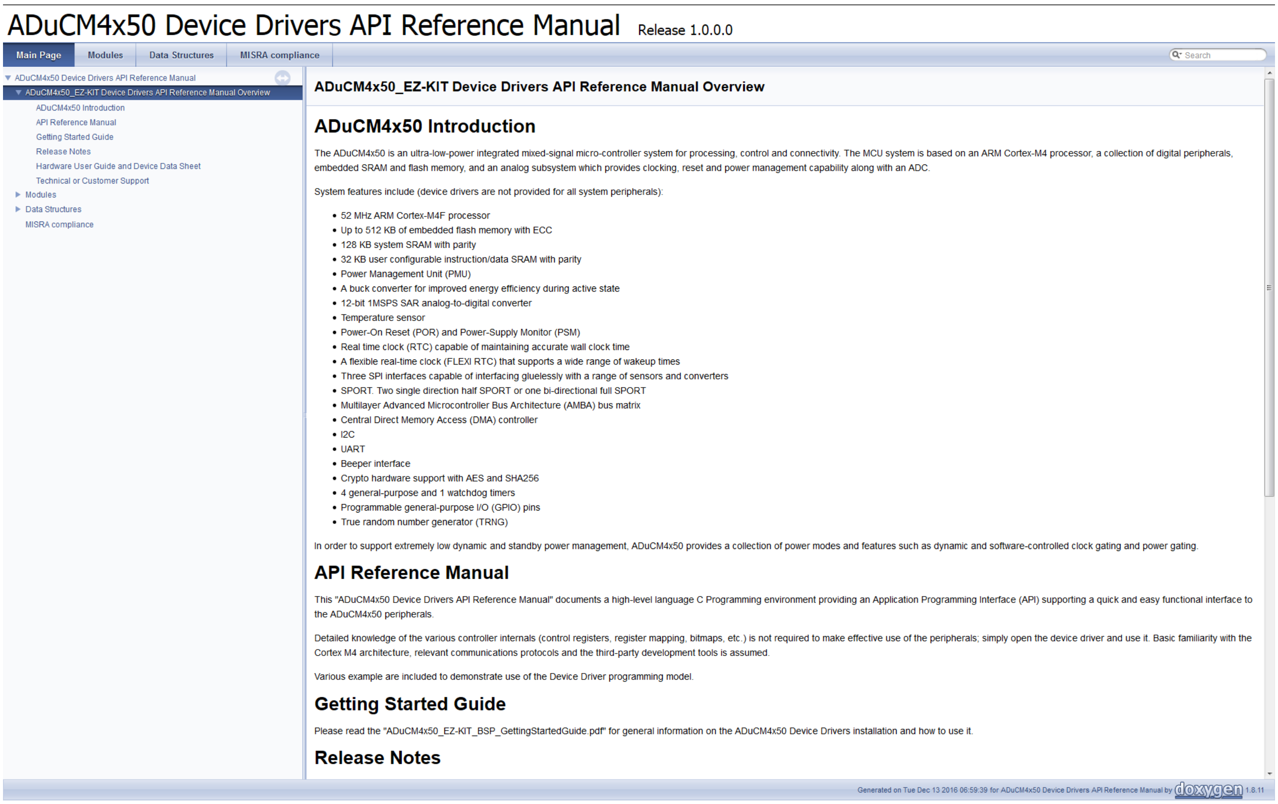
Device drivers should be added to or removed from a project in `system.rteconfig` instead of `system.svc`. See `Installation Components > CCES Project Options > RTE Configuration` section in this documentation for further information.

Complete documentation for the DFP is listed in `Introduction > References` section, at the top of this document. Most of the documentation is provided in PDF format.

The API documentation for the device drivers is available in HTML format as shown in *Figure 28. Device Driver Documentation*. The HTML documentation is located in the **`<ADuCM4x50_root>/Documents/html`** folder.

To open the HTML documentation, double-click on the index.html file.

Figure 28. Device Driver Documentation



7.1 Appendix

7.1.1 CMSIS

The ADuCM4x50 Device Family Pack is compliant with the Cortex Microcontroller Software Interface Standard (CMSIS). CMSIS prescribes a number of software organization aspects. One of the more convenient aspects of the CMSIS compliance is the availability of various CMSIS run-time library functions provided by the compiler vendor that implement many Cortex core access functions. These CMSIS access functions are used throughout the ADuCM4x50 DFP device driver implementation.

By wrapping up these Cortex core access functions into a compiler vendor library, the device drivers and application programmer are able to access the Cortex core implementation in a safe and reliable way. Examples of the CMSIS library access functions include functions to manage the NVIC (Nested Vectored Interrupt Controller) interrupt priority, priority grouping, interrupt enables, pending interrupts, active interrupts, etc.

Other CMSIS access functions include defining system startup, system clock and system timer functions, functions to access processor core registers, "intrinsic" functions to generate Cortex code that cannot be generated by ISO/IEC C, exclusive memory access functions, debug output functions for ITM messaging, etc. CMSIS also defines a number of naming conventions and various typedefs that are used throughout the ADuCM4x50 DFP.

Please consult *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors* mentioned in the `Introduction > References` section in this documentation or the www.arm.com website for complete CMSIS details.

7.1.2 Interrupt Vector Table

The IVT is a 32-bit wide table containing mostly interrupt vectors. It consists of two regions:

- The first sixteen (16) locations contain exception handler addresses. The highest location of these addresses have fixed (pre-determined) priorities.
- The balance of the IVT contains peripheral interrupt handler addresses which are not considered exceptions. Each of the peripheral interrupts has an individually programmable interrupt priority and they are therefore sometimes referred to as "programmable" interrupts, in contrast to the non-programmable (fixed-priority) exception handlers.

The IVT is declared and initialized in the `startup_<Device>.c` file. The organization of the first 16 locations (0:15) of the IVT is prescribed for ARM Cortex M-class processors as follows:

- IVT[0] = Initial Main Stack Pointer Value (MSP register)

The very first 32-bit value contained in the IVT is not an interrupt handler address at all. It is used to convey an initial value for the processor's main stack pointer (MSP) to the system start code. It must point to a valid RAM area in which the various reset function calls may have a valid stacking area (C-Runtime Stack).

- IVT[1] = Hardware Reset Interrupt Vector

The second 32-bit value of the IVT is defined to hold the system reset vector. This is also defined in startup_<Device>.c. The location is initialized with the reset interrupt handler function. When the system starts up, it calls the function pointed to by this location (once the boot kernel is complete).

- IVT[2:15] = Non-Programmable System Exception Handlers

These locations contain various exception handlers, e.g., NMI, Hard Fault, Memory Manager Fault, Bus Fault, etc. All of these handlers are given weak default bindings within the startup.c file, insuring all exceptions have a safe "trapping" implementation.

- Balance of IVT Contains Interrupt Vectors for Programmable Interrupts

The remaining IVT entries are mapped by the manufacture to the peripherals. In the case of the ADuCM4x50 processor, there are 72 (0-71) such peripheral interrupts. Each peripheral interrupt has a dedicated interrupt priority register that may be programmed at run-time to manage interrupt dispatching.

7.1.3 Startup_<Device>.c Content

The `<ADuCM4x50_root>/Source/GCC/startup_<Device>.c` file is required for every ADuCM4x50 application. This file is largely defined by the CMSIS standard and contains:

- Stack and Heap set-up
- Interrupt Vector Table

7.1.4 System_<Device>.c Content

The file `<ADuCM4x50_root>/Source/system_<Device>.c` is another CMSIS prescribed file implementing a number of required CMSIS APIs (SystemInit())

The system_<Device>.c file is a required and integral component for every ADuCM4x50 application.

- SystemInit()

This is a prescribed CMSIS startup function which is called by Reset Handler.

The first and most critical task performed during SystemInit() is the activation of the (potentially) relocated IVT. Any IVT relocation is done during the system reset handler under control of the **RELOCATE_IVT** macro. If the IVT has been moved, it must then be activated during SystemInit() by setting the Cortex core "Interrupt Vector Table Offset Register" in the Cortex Core System Control Block (SCB->VTOR) to the address of the new IVT.

Until the VTOR is reset, the default FLASH-based IVT remains active. The relocated IVT activation must be done *before* the application starts activating peripherals, but *after* the relocated IVT data has been copied.

Other important tasks performed during SystemInit() include bringing the clocks into a known state, configuring the PLL input source, and making the initial call to SystemCoreClockUpdate() (below), which must always be done (even by the application) after making any clock changes.

- SystemCoreClockUpdate()

This is another prescribed CMSIS API. The task performed here is to update the internal clock state variables within `system_<Device>.c` after making any clock changes. This insures that subsequent application calls to SystemGetClockFrequency() can return the correct frequency to device drivers attempting to configure themselves for serial BAUD rate, etc., or otherwise query the current system clock rate. SystemCoreClockUpdate() should always be called after any system clock changes.

8 ADuCM4x50 Device Family Pack for CCES Users Guide 3.0.0