ADuCM302x Device Family Pack User's Guide for CCES

ADuCM302x Device Family Pack User's Guide for CCES
Version 3.1.0, August 2018

http://www.analog.com

Contents

1 Introduction 4
1.1 Purpose 4
1.2 Scope of this Manual 4
1.3 Acronyms and Terms 5
1.4 Conventions 6
1.5 References 6
1.6 Additional Information 7
1.6.1 Manual Contents 7
2 Product Overview 8
2.1 Software System Overview 8
2.2 Hardware System Overview 8
3 Installation Components 12
3.1 CCES Project Support Files 12
3.1.1 OpenOCD Scripts 12
3.1.2 Linker Scripts 13
3.2 CCES Project Options 13
3.2.1 Tool Settings 14
3.2.2 Processor Settings 18
3.2.3 RTE Configuration 19
3.3 Debug Configurations 20
3.3.1 Target 20
3.3.2 Debugger 21
3.3.3 Limited Hardware Breakpoints 22
4 ADuCM302x System Overview 24
4.1 Block Diagram and Driver Layout 24
4.2 Boot-Time CRC Validation 25
5 Application Configuration 26
5.1 Application Initialization 26
5.2 Static Pin Multiplexing 27
5.3 UART Baud Rate Configuration Utility 28
5.4 Driver Include Files 29
5.5 Driver Configuration 30
5.5.1 Global Configuration 30
5.5.2 Configuration Defaults 30
5.5.3 Configuration Overrides 31
5.5.4 IVT Table Location 31
5.5.5 Interrupt Callbacks 31
6 Device Driver API Documentation 34
6.1 Device Driver API Documentation 34
6.2 34

6.3 Appendix 34
6.3.1 CMSIS 34
6.3.2 Interrupt Vector Table 35
6.3.3 Startup_<Device>.c Content 36
6.3.4 System_<Device>.c Content 36

1 Introduction

1.1 Purpose

This document describes how to use the ADuCM302x Device Family Pack (DFP) with CrossCore Embedded Studio (CCES). The ADuCM302x processor integrates an ARM Cortex-M3 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 ADuCM302x 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 ADuCM302x processor. This document assumes background in ADI’s ADuCM302x processor.

1.3 Acronyms and Terms

ADI	Analog Devices, Inc.
API	Application Programming Interface
ARM	Advanced RISC Machine
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 three important installation locations: the CCES toolchain installation root, the ADuCM302x DFP root and the ARM CMSIS root. Each of these packages can be installed in various places, which are referred to as follows:

- `<cces_root>`
  - The default CCES installer places the product at location **`C:/Analog Devices /CrossCore Embedded Studio x.y.z`** under Windows and `/opt/analog /cces/x.y.z` under Linux, but the install location may vary depending on user preferences. Where x.y.z is the version of CrossCore Embedded Studio (e.g. 2.6.0)
  - The default packs are placed at location `<cces_root>/ARM/packs /AnalogDevices`. There will be the following folder for ADuCM302x within that location called **`ADuCM302x_DFP`**.
- **`<ADuCM302x_root>`**
  - The directory `<cces_root>/ARM/packs/AnalogDevices /ADuCM302x_DFP/x.y.z` which contains the content of the Analog Devices ADuCM302x DFP.
    Where x.y.z is the version of the Device Family Pack (e.g. 2.0.0)
- **`<ARM_CMSIS_root>`**
  - The directory `<cces_root>/ARM/packs/ARM/CMSIS/4.5.0` which contains the content of the ARM CMSIS pack.

1.5 References

1. Analog Devices : `<ADuCM302x_root>/Documents`
   a. ADuCM302x_DFP_X.Y.Z_Release_Notes.pdf
   b. ADuCM302x_DFP_Device_Drivers_UsersGuide.pdf
   c. ADuCM302x_DFP_UsersGuide_CCES.pdf (this document)
   d. ADuCM302x Device Drivers API Reference Manual (Documents`/html` and hyperlinked)

2. For CrossCore Embedded Studio (CCES) [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>/Documents`**: Release notes.
3. *The Definitive Guide to the ARM CORTEX-M3*, Joseph Yiu, 2nd edition.
    - Every Cortex programmer’s bible; a must-have reference.
4. Micrium [http://micrium.com]
    a. uC/OS-II RTOS for ARM Cortex-M3
    b. uC/OS-II User’s Manual
5. ICE-1000 or ICE-2000 Emulator [http://www.analog.com]
6. ARM CMSIS pack [www.keil.com/cmsis/pack]

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
- ADuCM302x System Overview
- Build Configurations
- Examples
- Device Driver API Documentation

2 Product Overview

2.1 Software System Overview

The ADuCM302x Device Family Pack (DFP) provides files which are needed to write application software for the ADuCM302x 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 ADuCM302x DFP is designed to work with CrossCore Embedded Studio in CMSIS pack format for ARM.

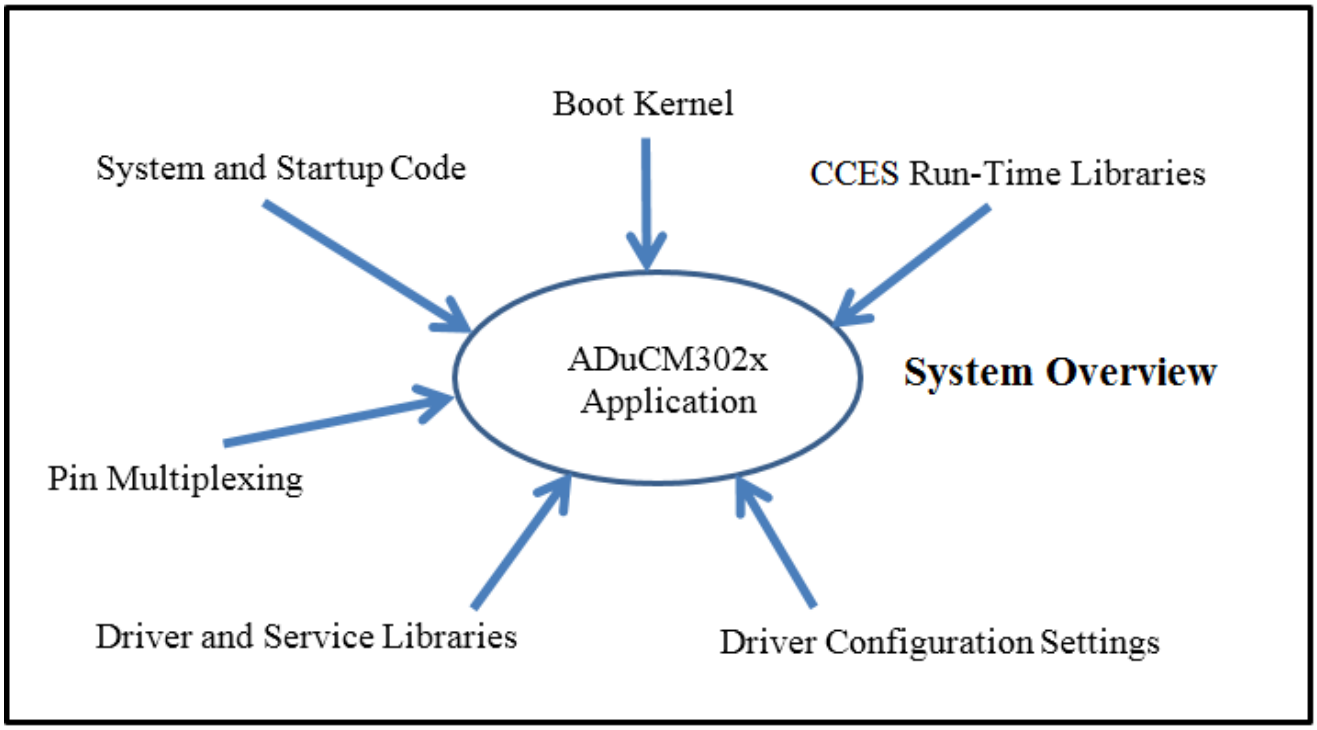

Figure 1. Software Overview

2.2 Hardware System Overview

The examples provided with the ADuCM302x DFP run on the Analog Devices' ADuCM302x-EZ-Board 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. ADuCM302x Evaluation Board with ICE-1000* and *Figure 5. ADuCM302x Evaluation Board with ICE-2000*).

If ICE-1000 is used, an adapter is required to connect ADuCM302x EZ-Board and ICE-1000 (see *Figure 4. Adapter for ICE-1000*).

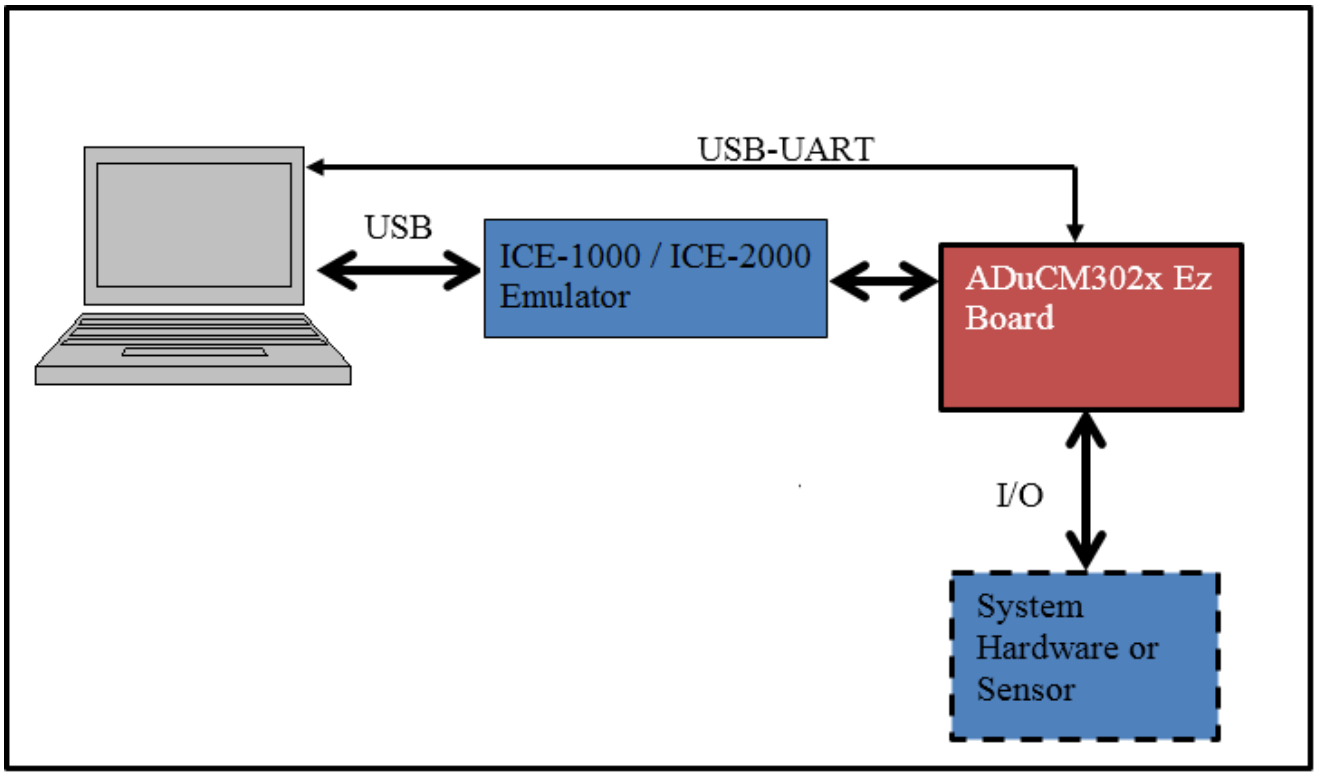

Figure 2. Hardware Overview

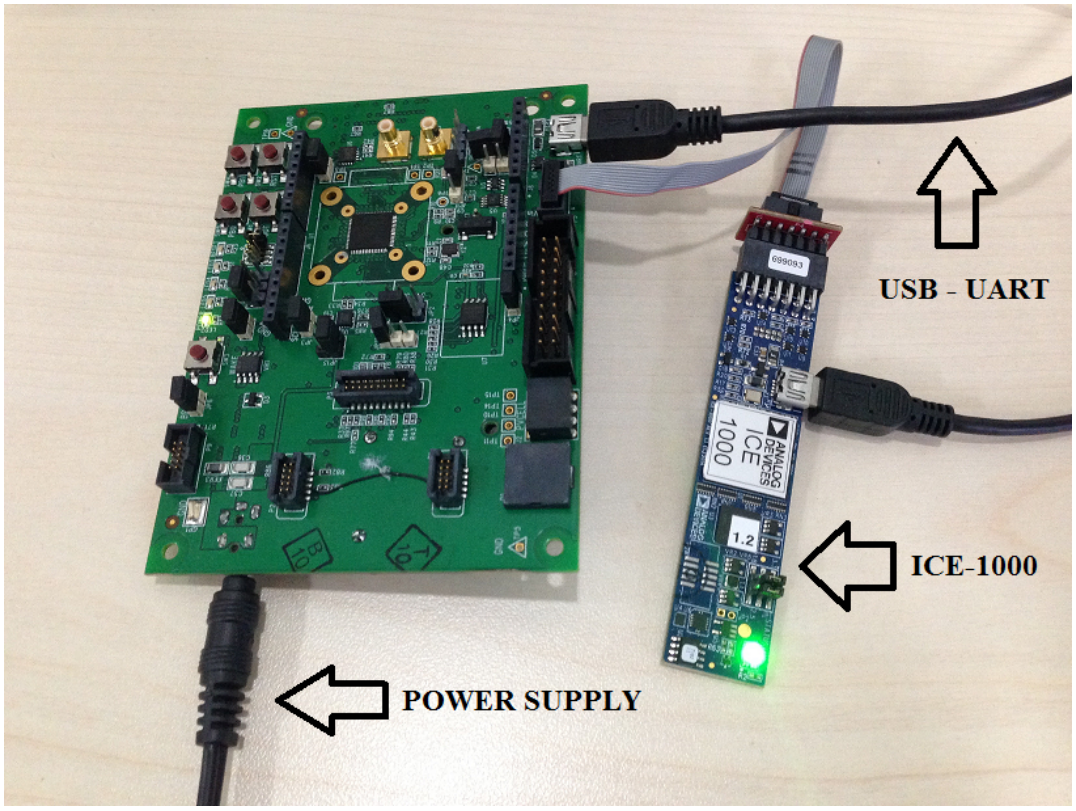

Figure 3. ADuCM302x Evaluation Board with ICE-1000

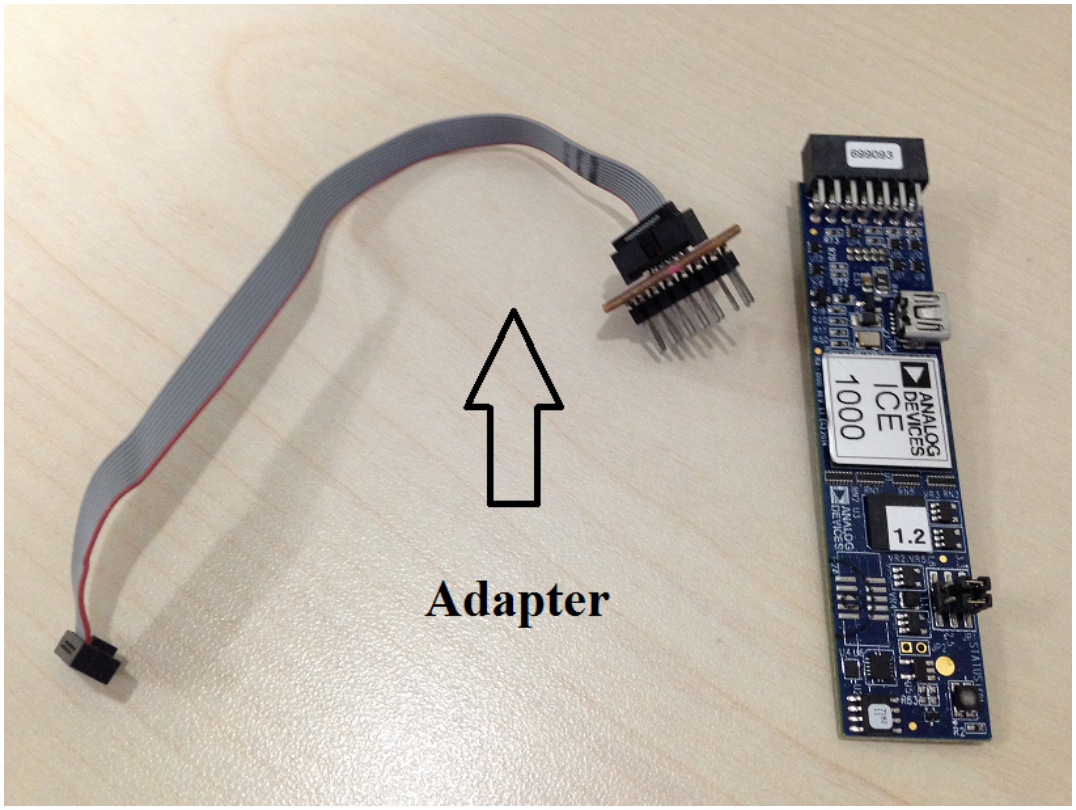

Figure 4. Adapter for ICE-1000

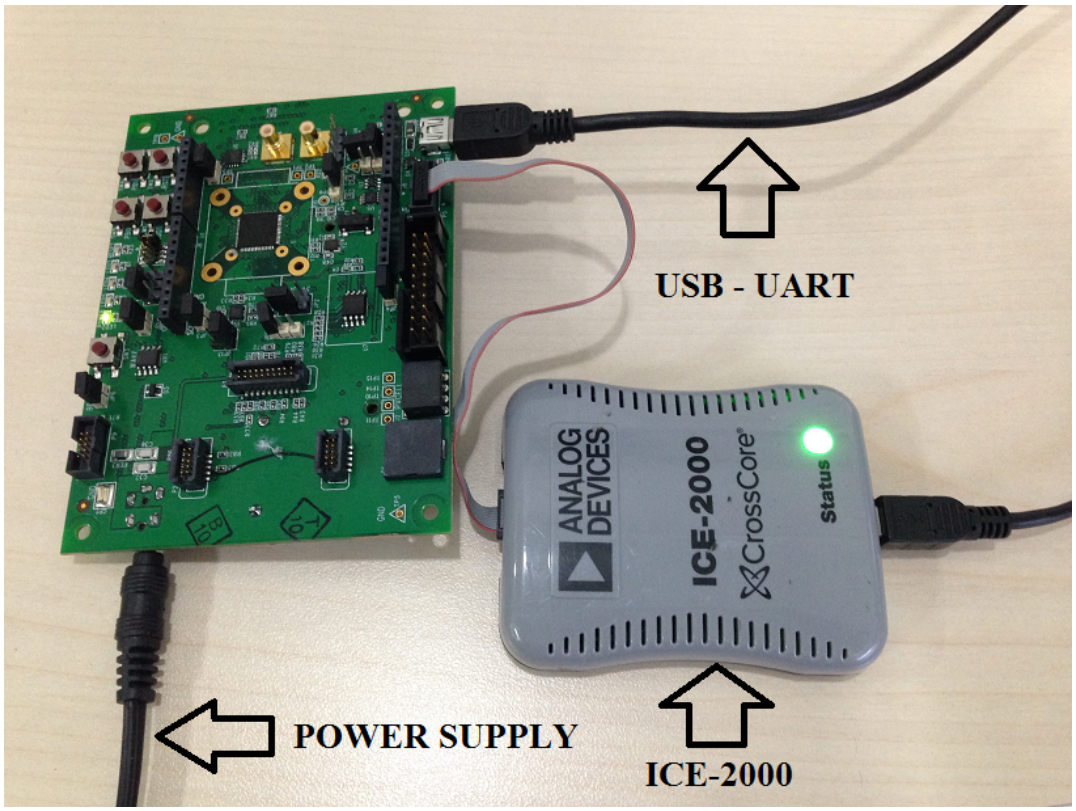

Figure 5. ADuCM302x Evaluation Board with ICE-2000

3 Installation Components

CrossCore Embedded Studio 2.6.0 http://www.analog.com/cces or later must be purchased and installed prior to installing the ADuCM302x Software Package. Follow the instructions in the CrossCore Embedded Studio (CCES) product installation procedure.

The ADuCM302x pack contents (startup code, device drivers, libraries, examples, tools, documentation, etc.) are placed at **`<cces_root>/ARM/packs/AnalogDevices/ADuCM302x_DFP/x.y.z`**.

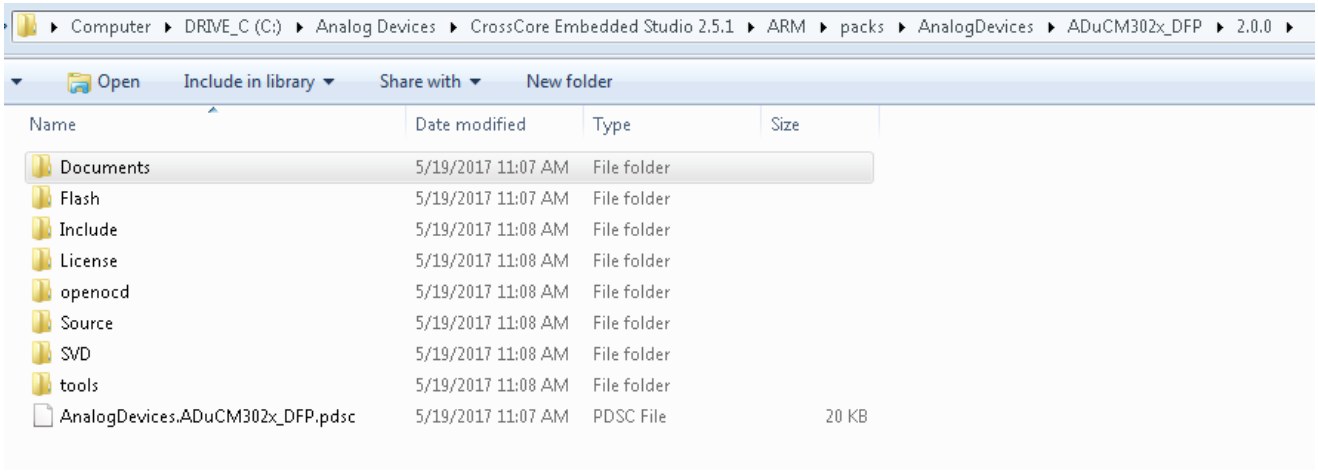

Figure 6. Installation Directory Structure

3.1 CCES Project Support Files

This section documents the CCES-specific details of the ADuCM302x 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 ADuCM302x Device Family Pack, necessary to build / run applications in the CCES environment.

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

3.1.1 OpenOCD Scripts

OpenOCD supports ADuCM3027/9 targets from Analog Devices. These parts are supported through the target config files `aducm3027.cfg` and `aducm3029.cfg`. These two parts only support SWD.

The target config file is to package everything about a given chip that board config files need to know (See ARM® Development Tools Documentation > OpenOCD User's Guide in CCES On-line Help), which includes:

- Set defaults
- Add TAPs to the scan chain
- Add CPU targets (includes GDB support)
- CPU/Chip/CPU-Core specific features
- On-Chip flash

The **<ADuCM302x_root>/openocd/scripts/target** folder contains *.tcl and *.cfg files.

- The `aducm302x.tcl` file defines common routines for Analog Devices ADuCM302x.
- The `aducm3027.cfg/aducm3029.cfg` files defines target-specific information and include `aducm302x.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 **<ADuCM302x_root>/Source/GCC** folder contains *.ld files.

The `ADuCM3027.ld/ADuCM3029.ld` files configure:

- Memory regions
- Library group
- Sections and symbol values

3.2 CCES Project Options

This section documents the CCES project options for ADuCM302x 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

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 7. CrossCore GCC ARM Embedded Assembler - Preprocessor* below.

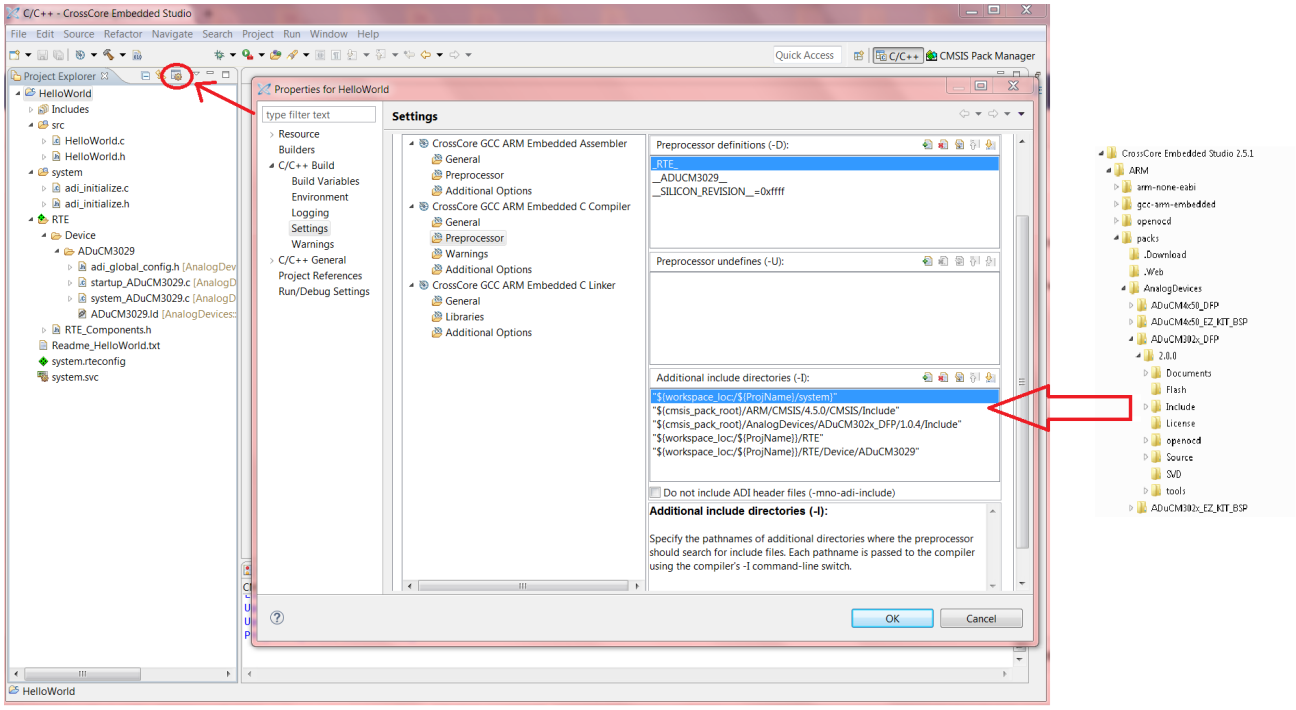

Figure 7. CrossCore GCC ARM Embedded Assembler - Preprocessor

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

CrossCore GCC ARM Embedded C Compiler

This section allows user to set any compiler flags, definitions, include search paths and optimization levels to be passed to the CrossCore GCC ARM Embedded C Compiler during build.

Optimization levels can be set from General sub-section as shown in *Figure 8. CrossCore GCC ARM Embedded C Compiler - General* below.

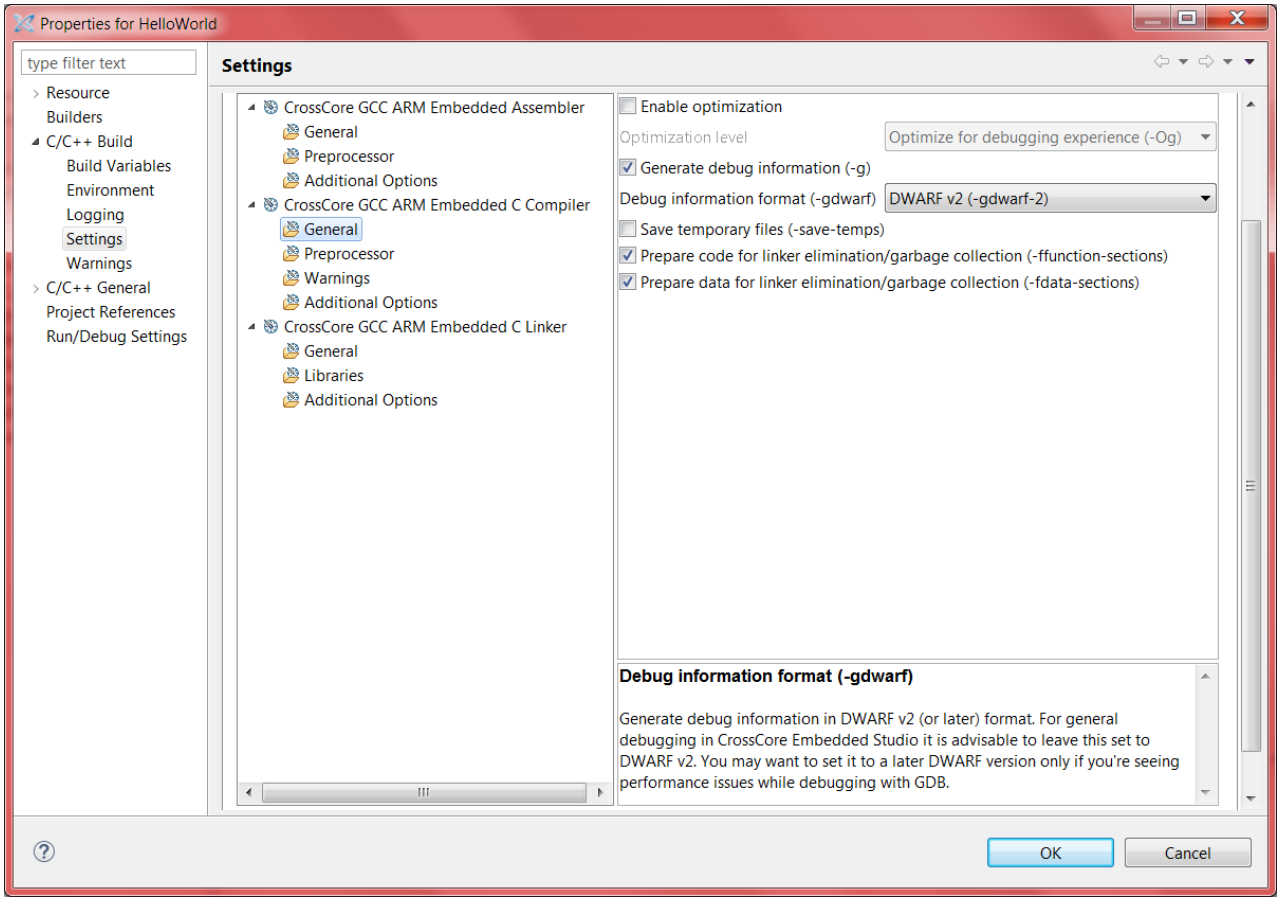

Figure 8. CrossCore GCC ARM Embedded C Compiler - General

Preprocessor definitions, undefines and Include search paths can be set from Preprocessor sub-section as shown in *Figure 9. CrossCore GCC ARM Embedded C Compiler - Preprocessor* below.

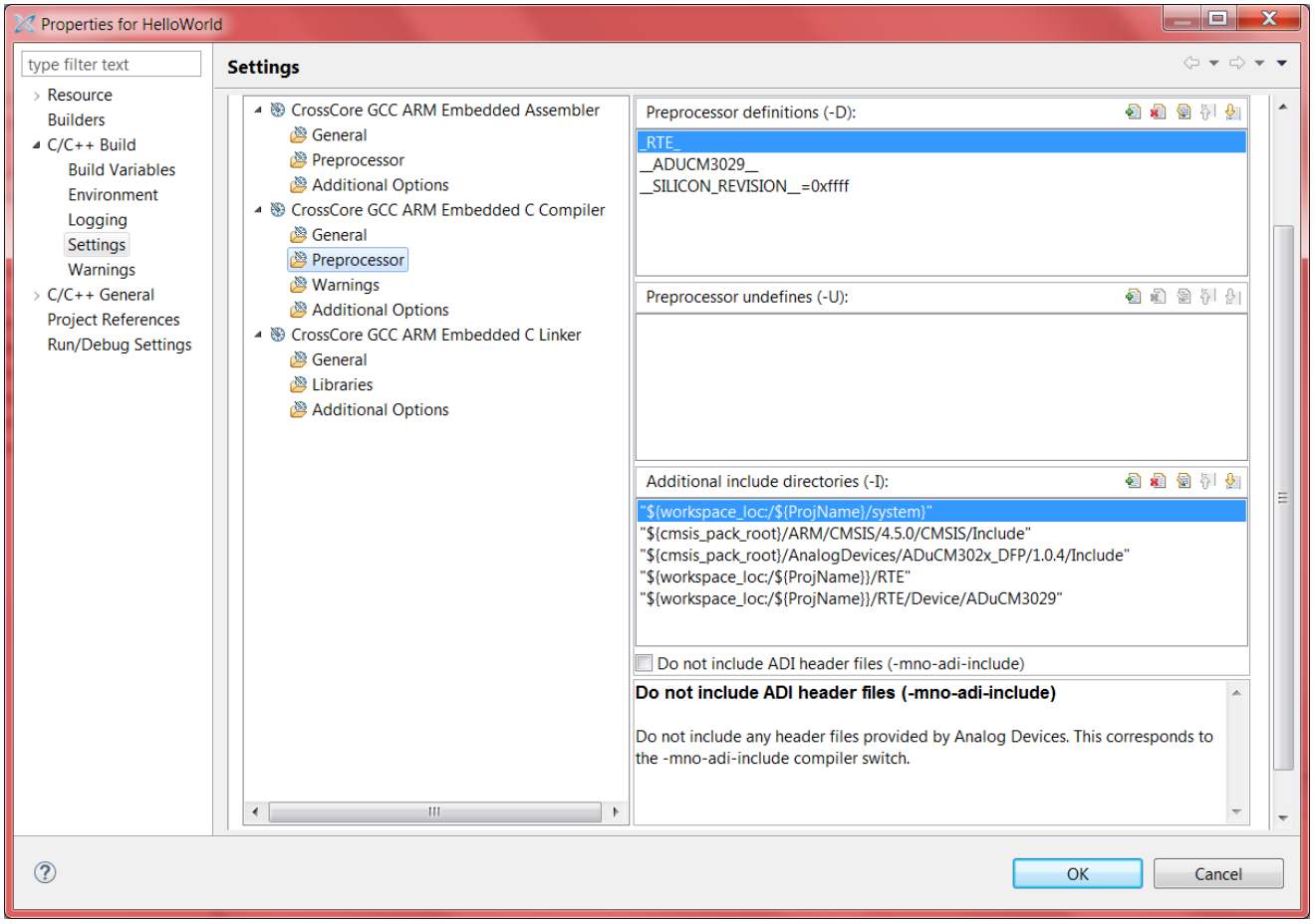

Figure 9. CrossCore GCC ARM Embedded C Compiler - Preprocessor

Additional compiler options can also be added from 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 set whether to enable linker elimination by the CrossCore GCC ARM Embedded C Linker as shown in *Figure 10. CrossCore GCC ARM Embedded C Linker - General* below.

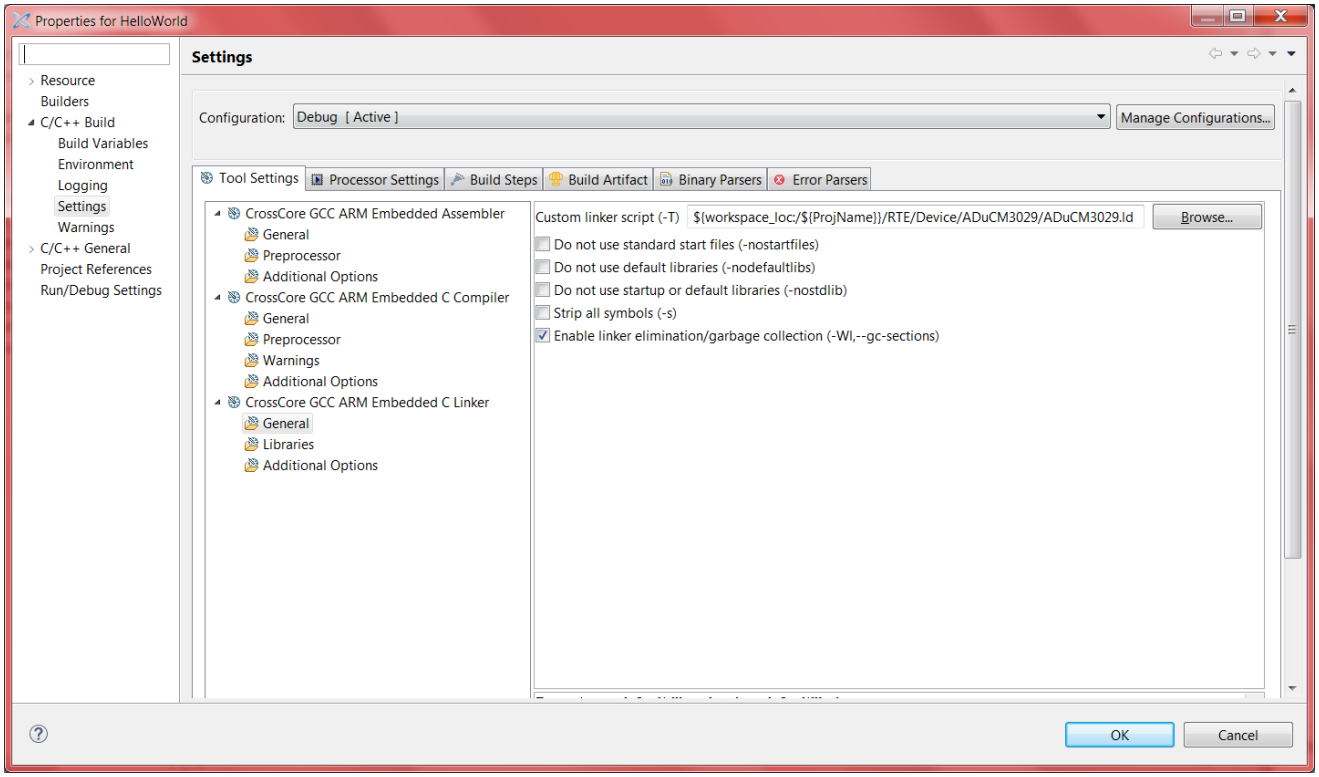

Figure 10. CrossCore GCC ARM Embedded C Linker - General

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 11. CrossCore GCC ARM Embedded C Linker - Libraries* below.

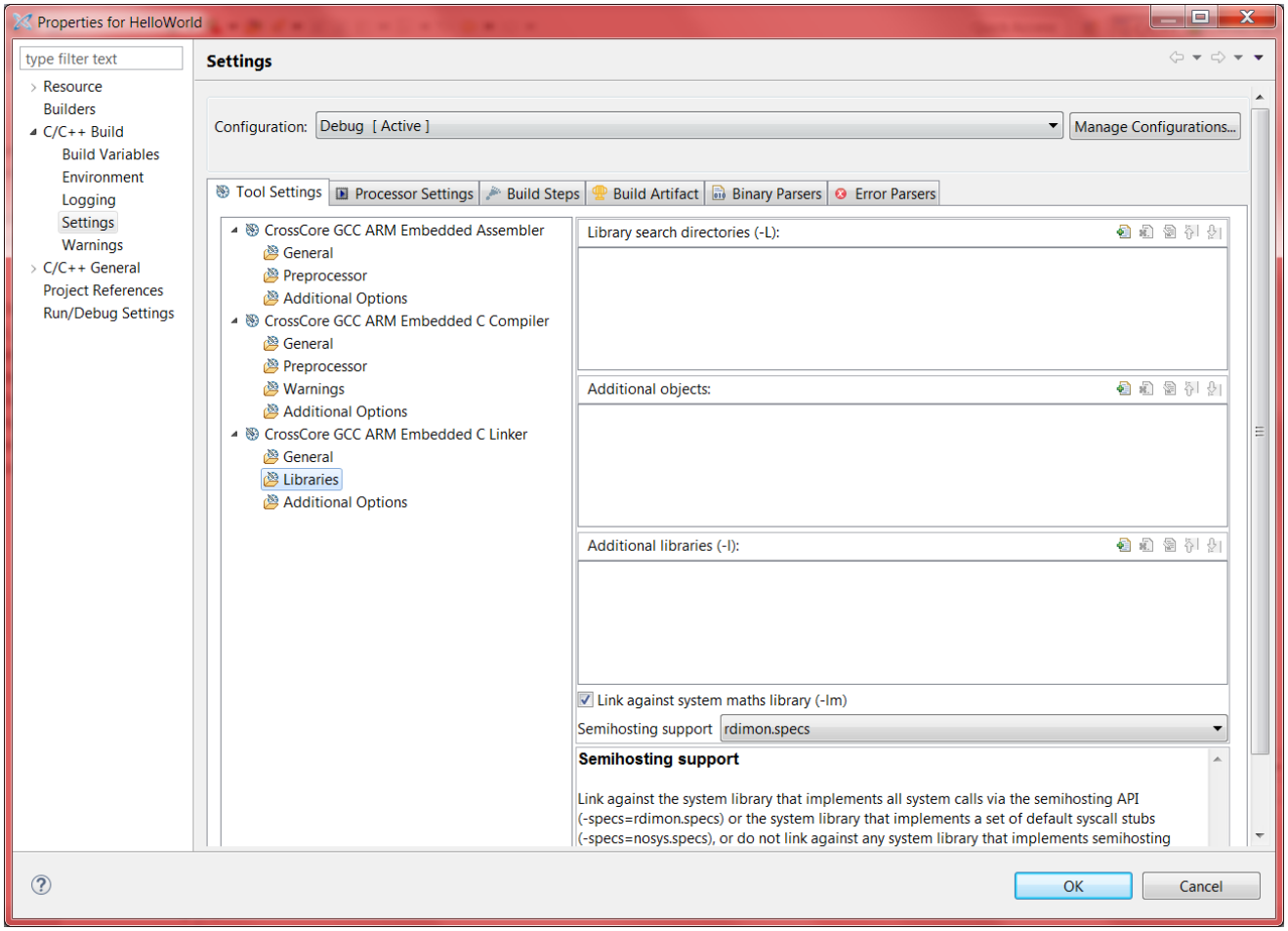

Figure 11. CrossCore GCC ARM Embedded C Linker - Libraries

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

3.2.2 Processor Settings

This tab can show the Processor Family and Processor Type. The silicon revision of a project can be set from Processor Settings tab as shown in *Figure 12. Processor Settings* below.

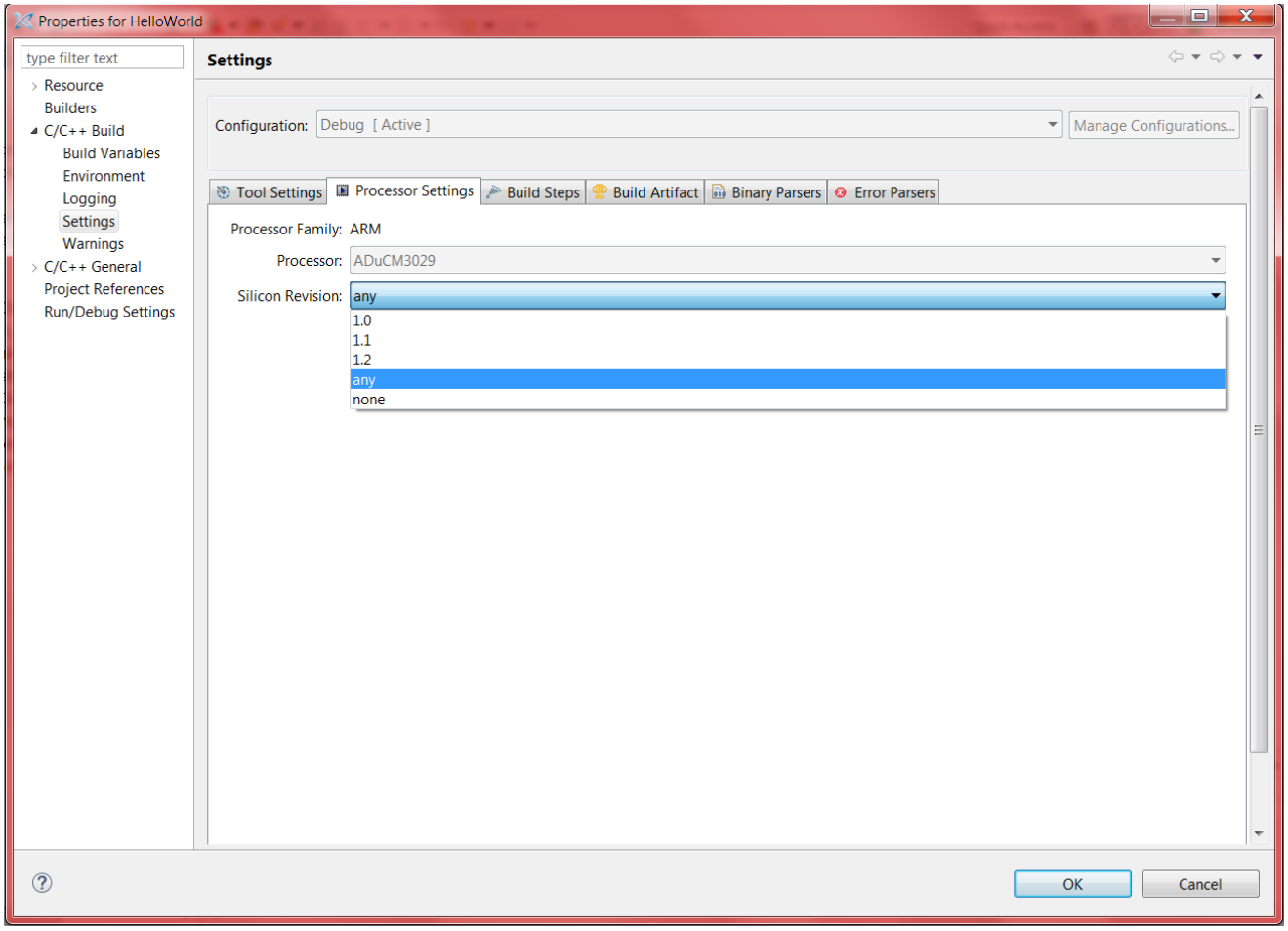

Figure 12. Processor Settings

Noted that after changing the silicon revision in this tab, the related source files in the project will be regenerated.

Processor type cannot be set from this tab, you can refer to section `RTE Configuration - Device` in this documentation to see how to change the processor type of a project.

3.2.3 RTE Configuration

Run-Time Environment (RTE) Configuration editor allows user to manage software components, devices, and packs in a project, which can be opened by double-clicking the `system.rteconfig` in the project. See the CMSIS C/C++ Development User's Guide in CCES On-line Help for further information.

CMSIS and Devices components including drivers and services should be 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 `system.rteconfig` and save the file.

`CMSIS-CORE` and `Startup` as well as `Global Configuration`, which is required by `Startup`, should be added manually when create a new project in order to build successfully.

Figure 13. 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.
- Using the `Change...` button to select a different device for the project.

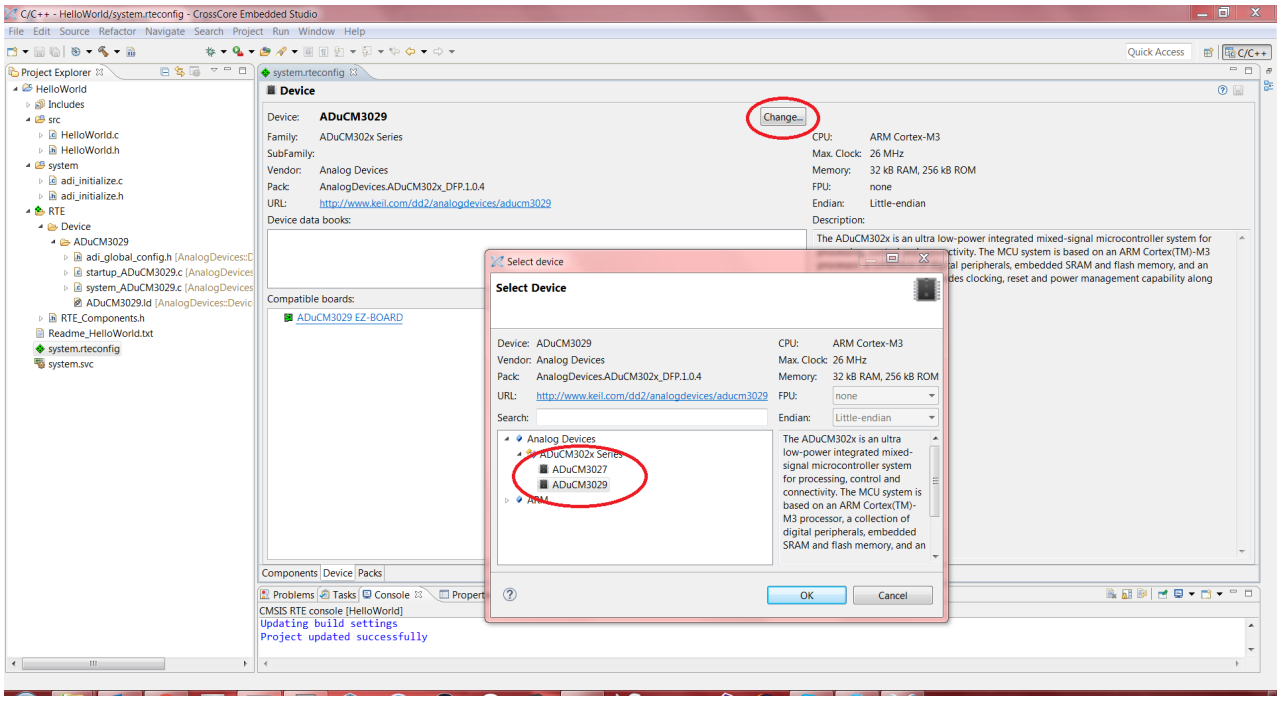

Figure 14. RTE Configuration - Device

Noted that after changing the processor type, the related source files in the project will be updated.

3.3 Debug Configurations

When creating a debug configuration for ADuCM302x 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 ADuCM3027/9" in the drop-down list as shown in *Figure 15. Debug Configurations - Target* below.

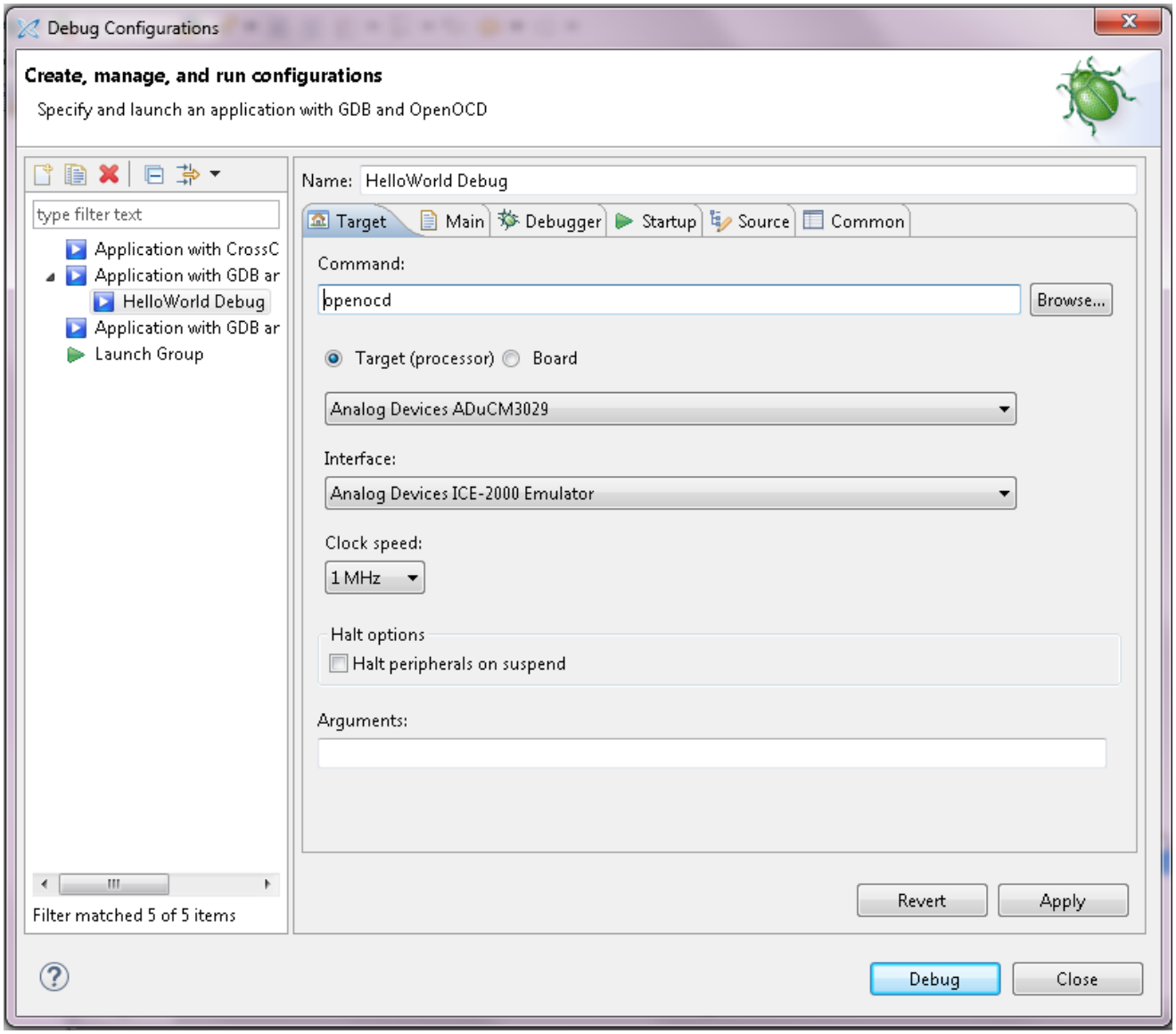

Figure 15. Debug Configurations - Target

3.3.2 Debugger

The Debugger tab allows you to configure GDB, as shown in *Figure 16. Debug Configurations - Debugger* below.

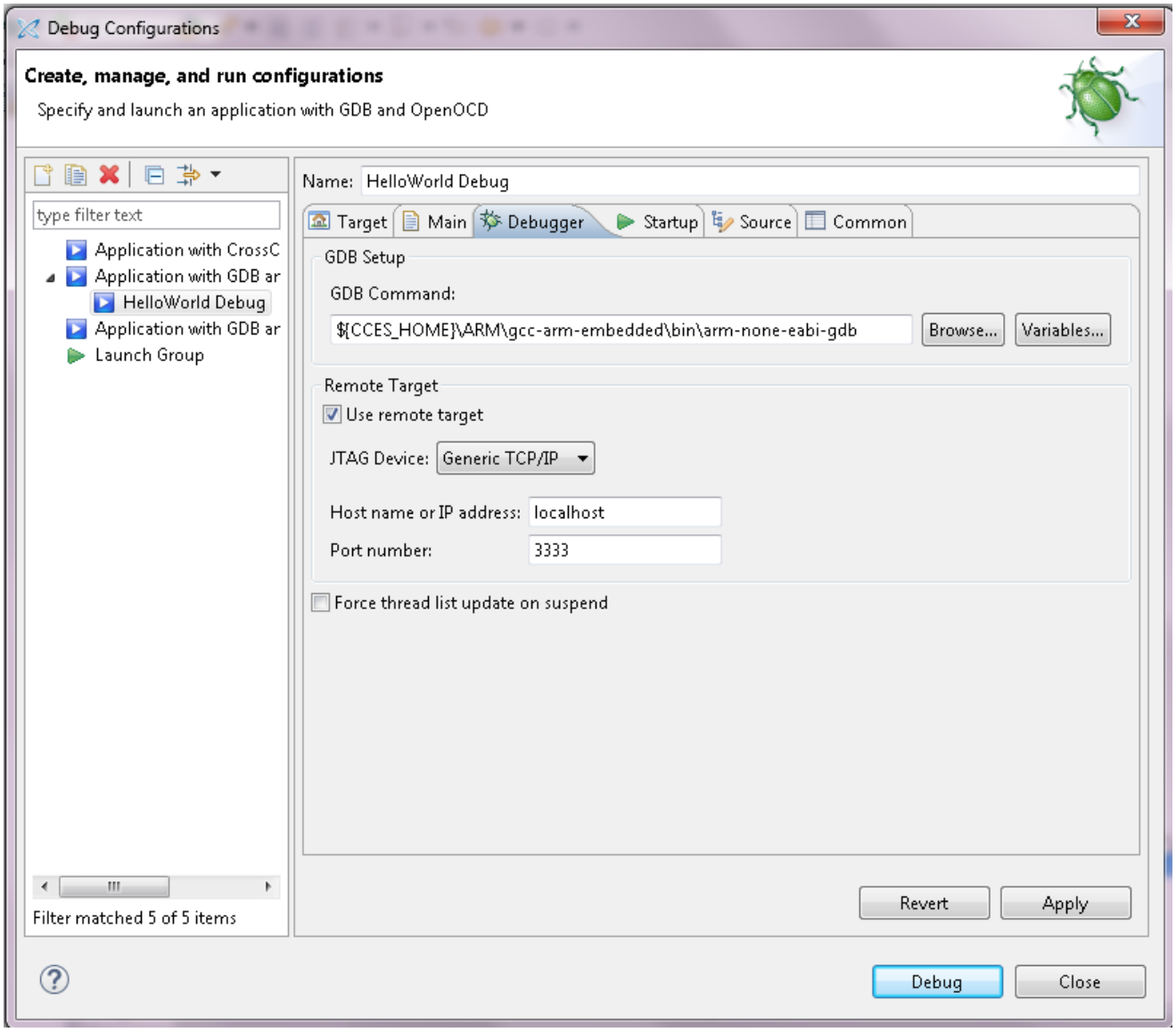

Figure 16. Debug Configurations - Debugger

3.3.3 Limited Hardware Breakpoints

After clicking Debug in the Debug Configurations window, a pop-up window will show, warning you of the limitation of hardware breakpoints in the selected target. It will ask if you want to remove the breakpoint setting command(s) from the Initialization Commands (See *Figure 17. Hardware Breakpoints Limited*). It's recommended to select Yes, otherwise you may not be able to insert hardware breakpoint(s) during debugging.

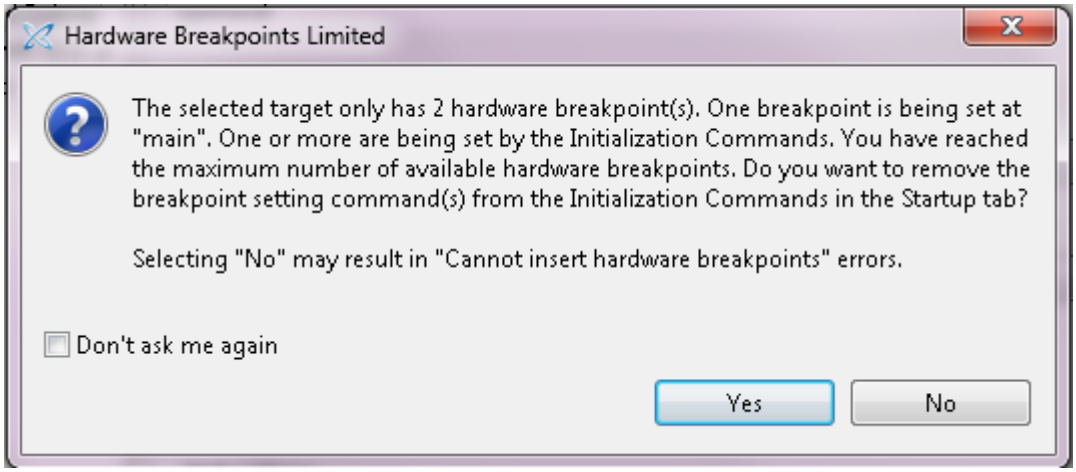

Figure 17. Hardware Breakpoints Limited

4 ADuCM302x System Overview

4.1 Block Diagram and Driver Layout

The Peripheral Device Drivers and System Services installed with this software package are used to configure and use various ADuCM302x on-chip peripherals. *Figure 18. Peripherals and Driver Source* illustrates the available peripherals and interconnects on the ADuCM302x processor and corresponding source files in the **<ADuCM302x_root>**/`Source` directory.

The driver sources are located in the **<ADuCM302x_root>**/`Source` directory. When you add a new CMSIS driver or service component to your project, the necessary include search path **<ADuCM302x_root>**/`Include` will also be added to the project's additional include directories. See the section `CrossCore GCC ARM Embedded C Compiler – Preprocessor` in `Installation Components` of this document for more information.

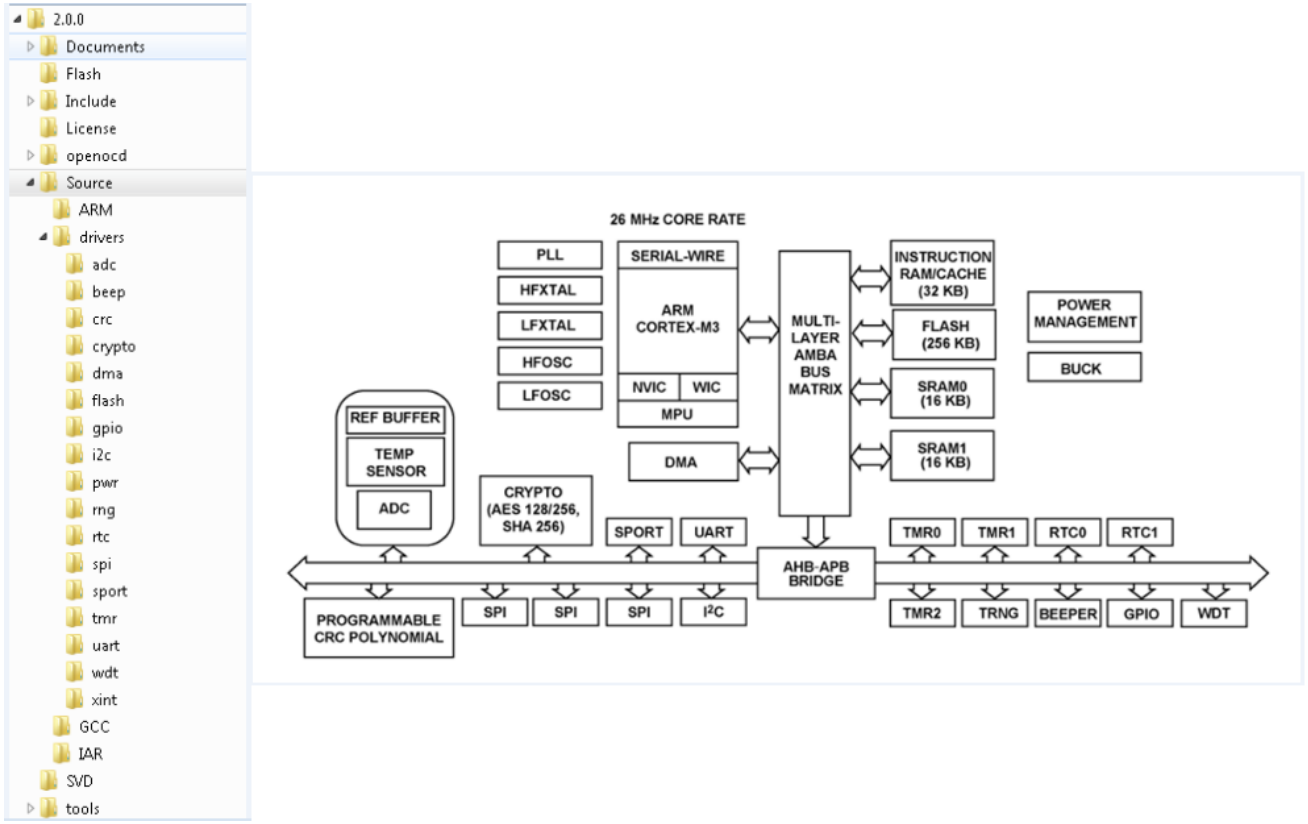

Figure 18. Peripherals and Driver Source

4.2 Boot-Time CRC Validation

The ADuCM302x 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.

ADuCM302x 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 and/or services 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().

```
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 19. - Application Initialization

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

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

Figure 20. - adi_initComponents()

5.2 Static Pin Multiplexing

The Pin Multiplexing Add-in is recommended by default when creating a CCES project for ADuCM302x; 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. After doing save, a C source file (`project_name /system/pinmux/GeneratedSources/pinmux_config.c`) will be generated automatically which sets the GPIO port configuration registers based on the peripherals and functions selected. 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.

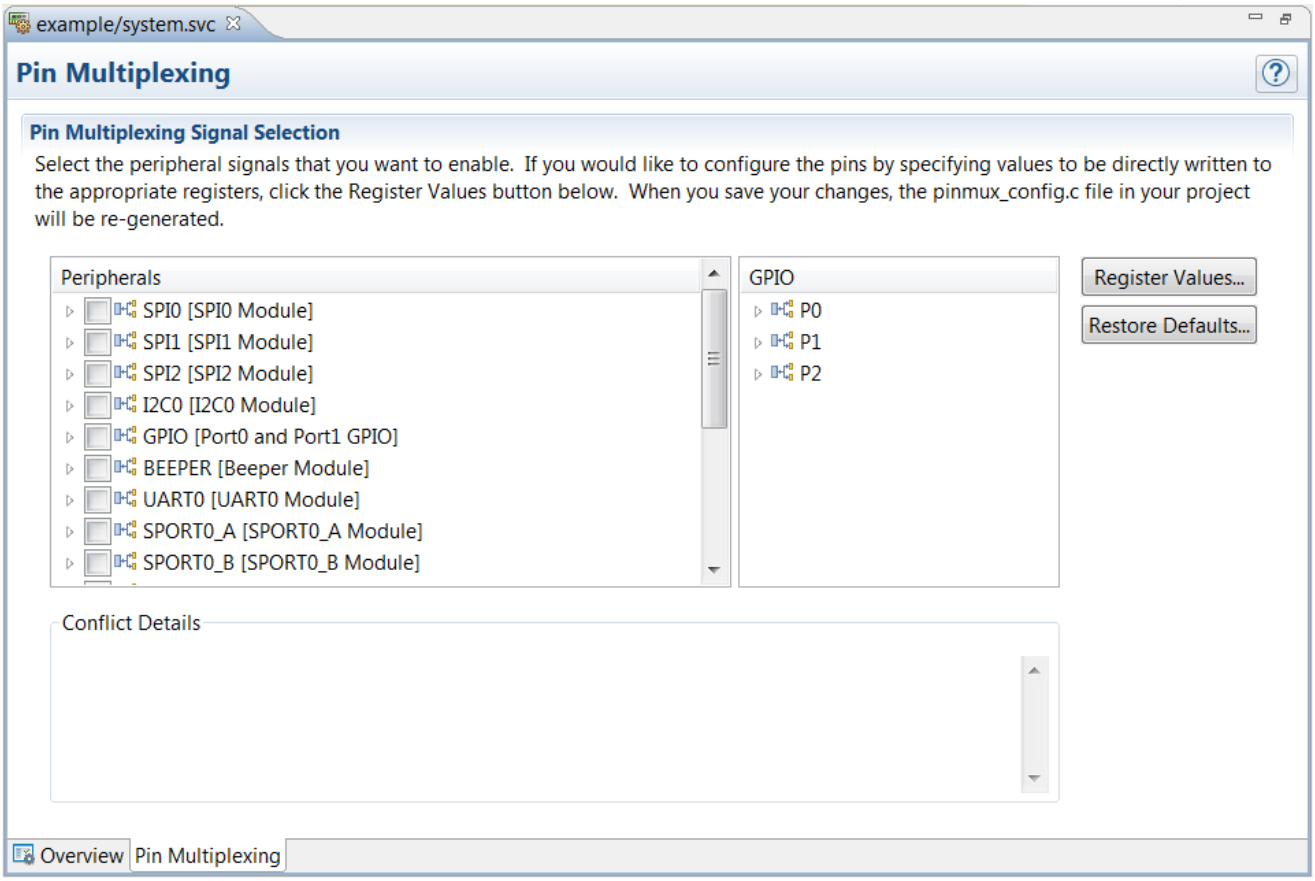

Figure 21. Pin Multiplexing Add-in

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.

5.3 UART Baud Rate Configuration Utility

Included with the ADuCM302x Device Family Pack is a `UartDivCalculator` command line 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

`<ADuCM302x_root>/Include`

`<ARM_CMSIS_root>``/CMSIS/Include`

which are added by CMSIS components automatically. 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) initialization:

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 `<ADuCM302x_root>/Include/config/adi_global_config.h`, which will be copied to the local project directory automatically after adding the `Global Configuration` CMSIS component. We recommend using the same approach for overriding the driver-specific configuration files as described below to override the global feature set-up (See more about configuration overrides in the sections below). For example, to overwrite the RTOS feature, set the corresponding macro in `adi_global_config.h` to 0 as shown in *Figure 22. Global Configuration File Contents* below:

```
/*! Set this macro to 1 to enable multi-threaded support */
#define ADI_CFG_ENABLE_RTOS_SUPPORT 0
```

Figure 22. Global Configuration File Contents

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 `<ADuCM302x_root>/Include/config` directory, which will be copied to your local project directory automatically after adding the driver component so you can edit for localized changes.

5.5.3 Configuration Overrides

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

1. Local edits can be applied.
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-M3 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 to 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-M3 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.

6 Device Driver API Documentation

6.1 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 27. Device Driver Documentation*. The HTML documentation is located in the **`<ADuCM302x_root>/Documents/html`** folder.

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

6.2

Figure 27. Device Driver Documentation

6.3 Appendix

6.3.1 CMSIS

The ADuCM302x 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 ADuCM302x 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 ADuCM302x DFP.

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

6.3.2 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 ADuCM302x processor, there are 64 (0-63) such peripheral interrupts. Each peripheral interrupt has a dedicated interrupt priority register that may be programmed at run-time to manage interrupt dispatching.

6.3.3 Startup_<Device>.c Content

The `<ADuCM302x_root>/Source/GCC/startup_<Device>.c` file is required for every ADuCM302x application. This file is largely defined by the CMSIS standard and contains:

- Stack and Heap set-up
- Interrupt Vector Table

6.3.4 System_<Device>.c Content

The file `<ADuCM302x_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 ADuCM302x 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.