ADuCM302x Device Family Pack User's Guide for Keil

Contents

1 Introduction

1.1 Purpose

This document describes the ADuCM302x Device Family Pack (DFP) for Keil uVision and its use. 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 Device Family Pack. This document explains what is included with the package 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
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
RTOS	Real-Time Operating System
TRACE	Debugging with TRACE access port

1.4 Conventions

Throughout this document, we refer to two important installation locations: the ADuCM302x Device Family Pack and the Keil toolchain installation root. Each of these packages can be installed in various places, which are referred to as follows:

- `<Keil_root>`
  - The default KEIL Pack installer places the product at location `C:\Keil_v5\ARM\Pack\AnalogDevices`. There will be the following folder for ADuCM302x within that location called **`ADuCM302x_DFP`**.
- **`<ADuCM302x_root>`**
  - The directory **C:\Keil_v5\ARM\Pack\AnalogDevices\`ADuCM302x_DFP\3.1.0`** which contains the content of the ADuCM302x KEIL Pack file.

1.5 References

1. Analog Devices : `<ADuCM302x_root>/Documents`
   a. ADuCM302x_DFP_3.1.0_Release_Notes.pdf
   b. ADuCM302x_DFP_Device_Drivers_UsersGuide.pdf
   c. ADuCM302x_DFP_Getting_Started_Guide_Keil.pdf (brief introduction)
   d. ADuCM302x_DFP_Users_Guide_Keil.pdf (this document)
   e. ADuCM302x Device Drivers API Reference Manual (`Docs/html` and hyperlinked)
2. For Keil `<Keil_root>/ARM/Hlp` [http://www.keil.com]
   a. Keil MDK for Cortex-M microcontroller.
   b. 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. SEGGER J-Link Emulator [http://www.segger.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
- ADuCM302x System Overview
- Build Configurations
- Examples
- Device Driver API Documentation

2 Product Overview

2.1 Software System Overview

The ADuCM302x EZ Kit BSP 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 BSP is designed to work with KEIL uVision in CMSIS pack format for ARM[II.a]
.

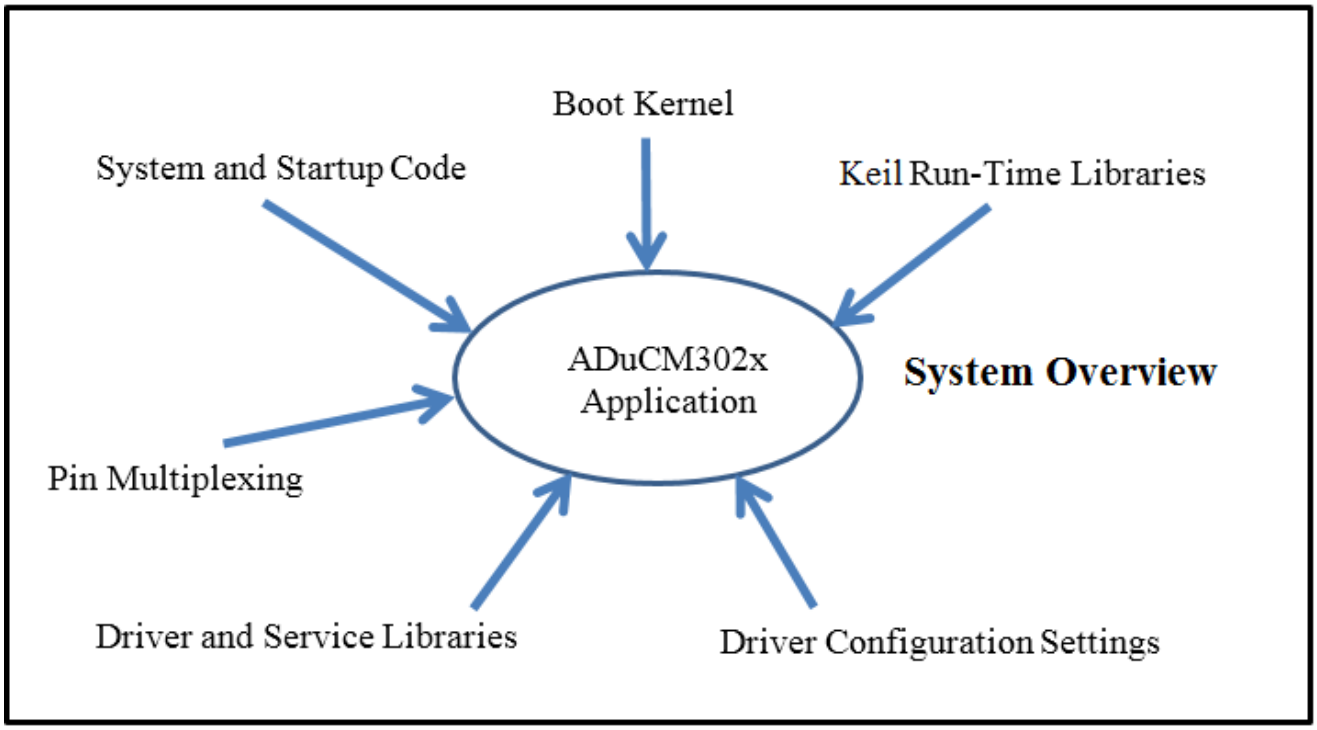

Figure 1. Software Overview

2.2 Hardware System Overview

The examples provided with the ADuCM302x BSP run on the Analog Devices' ADuCM302x-EZ-Board evaluation board. The evaluation board is connected to the host computer using a Segger J-Link lite emulator over the evaluation board's JTAG or TRACE debug port interface connectors. External I/O signals and system hardware are connected to the evaluation board connectors as shown in Figure 3, ADuCM302x EZ-Board.

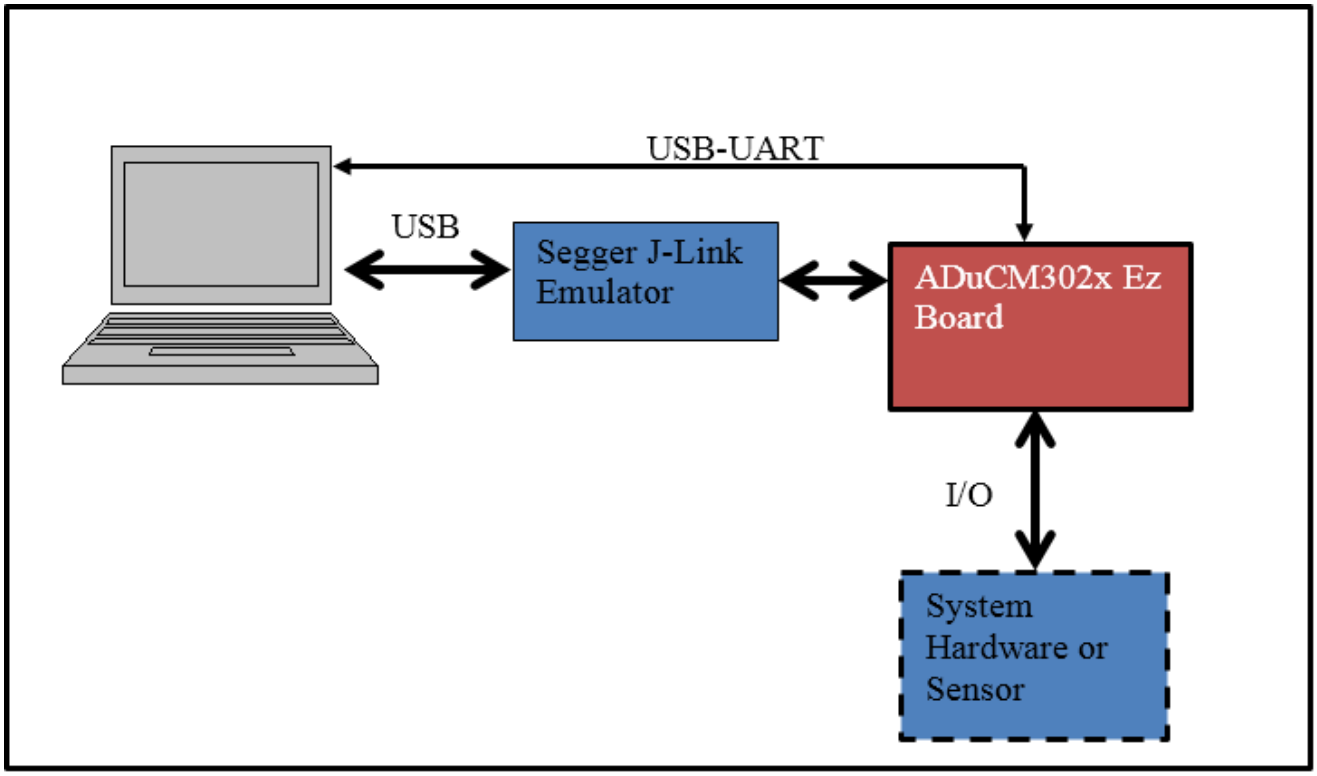

Figure 2. Hardware Overview

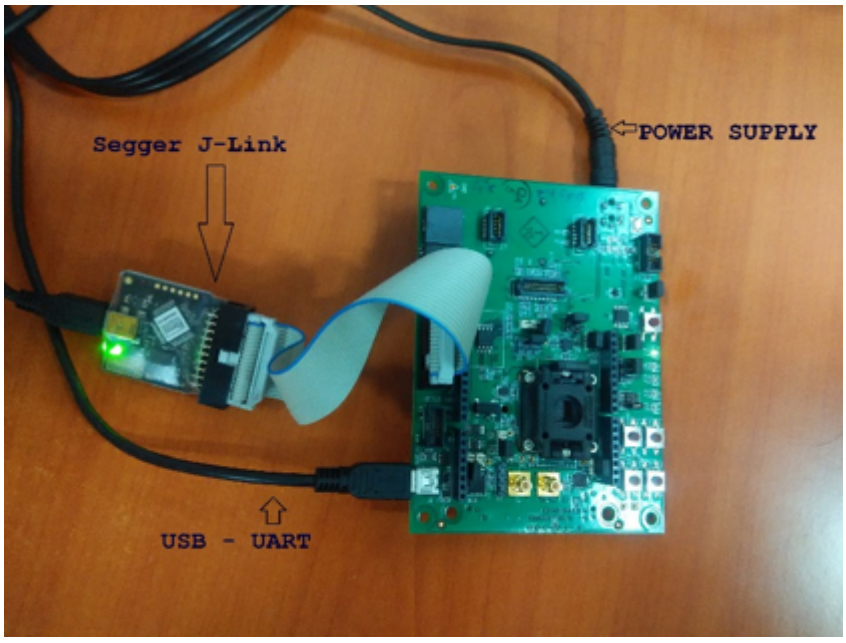

Figure 3. ADuCM302x EZ-Board

3 Installation Components

The KEIL MDK http://www2.keil.com/mdk5 or later must be purchased and installed prior to installing the ADuCM302x Device Family Pack. Follow the instructions in the KEIL MDK for ARM product installation procedure (Keil uVision Full license version).

Keil toolchain support files are placed with the Keil installation folder (e.g. **`C:\Keil_v5\ARM\Pack\AnalogDevices\ADuCM302x_DFP`**). This also includes files for flash loading, debugging, etc.

The ADuCM302x EZ-KIT Lite (startup code, device drivers, libraries, examples, tools, documentation, etc.) are placed at **`Keil_v5\ARM\Pack\AnalogDevices\ADuCM302x_DFP\x.y.z`**.

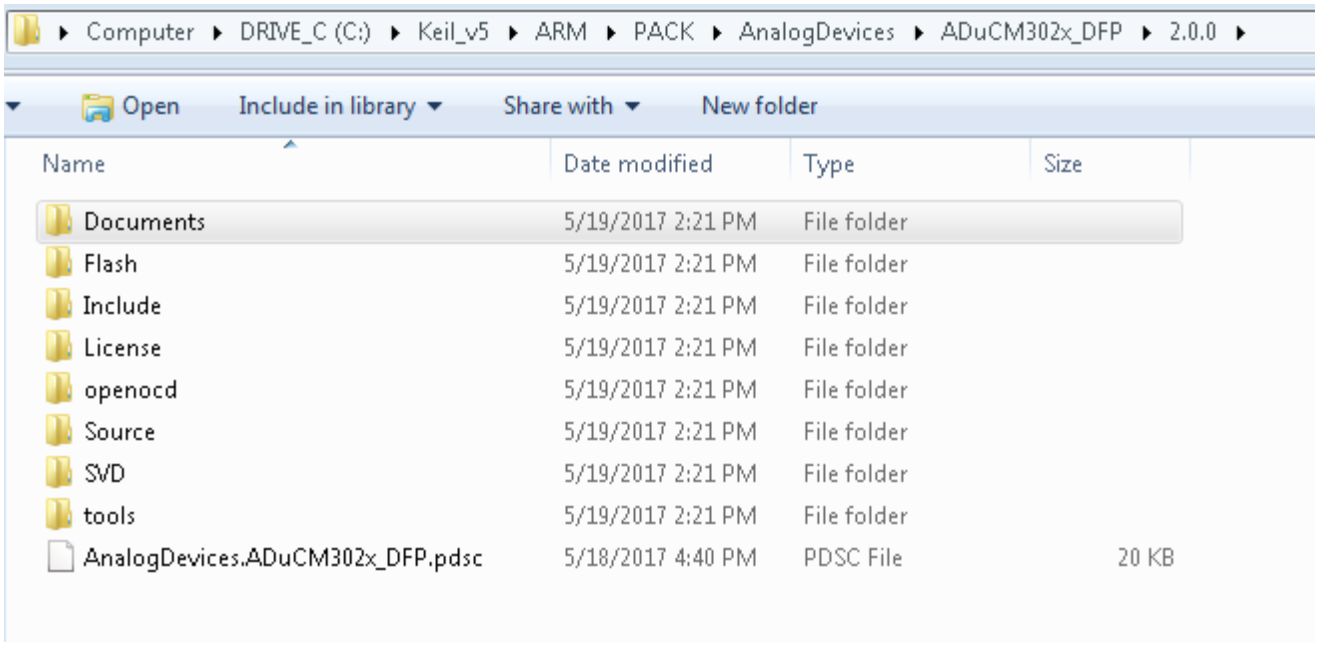

Figure 4. Installation Directory Structure

3.1 Keil Project Support Files

This section documents the KEIL-specific details of the ADuCM302x Device Family Pack. A working knowledge of the KEIL toolchain and environment is assumed. See the KEIL reference materials for details of installing, configuring and using the KEIL tools. The following are the list of important files contained in the ADuCM302x DFP, necessary to build applications in the Keil uVision environment.

- SCT File

- Jlink debugger setting file (JLinkSettings.ini)
- Flash Loader Algorithm (.FLM)

3.1.1 SCT File

A .SCT file contains the memory configuration of the ADuCM302x core, this file can be customized as per the application needs.

A SCT file can be stored in the same path as a Keil project file. The SCT file in the project can be used to define and allocate various memory regions:

- Internal SRAM size
- FLASH size
- Placement of all code and data blocks
- Reserves memory for post-link processing (CRC checksums, parity, etc)

The SCT file can be used to:

- Define Memory “Regions” (size, location, alignment, etc.).
- FLASH Area, Internal SRAM Code, SRAM Data, Special-Purpose, etc.
- Internal SRAM Bank Partition.
- Define “Blocks” for Specific Tasks
    - Runtime Stack, Heap Space, etc.
    - Size and Alignment.
- Specify Runtime “Initialization” Sections
    - Linker and C-Runtime Startup Collaboration.
    - Compress Code/Data for Expansion into Internal SRAM at Startup.
- Manage Code and Data “Placements” within Regions
    - Explicit Interrupt Vector Table (IVT) Placement.
    - Read-Only, Read-Write Attribute.
    - Special Section Handling.
    - Default Flash, Code and Data Placements.
    - Explicit Stack and Heap Block Placements.

3.1.2 Jlink Settings File

The J link setting file (JLinkSettings.ini) is present in each example project folder, this file helps the J-Link debugger to retain the device configuration every time a debugger session is initiated. If this file is not present the user has to manually select the device in the jlink settings.

3.1.3 Flash Loader Algorithm

The Flash Loader is used to burn application executables to the on-chip flash over the debug port (using the emulator). The application may then be executed directly from flash.

The flash algorithm is stored in the following path in the Pack Installation **`C:\Keil_v5\ARM\Pack\AnalogDevices\ADuCM302x_DFP\x.y.z\Flash`**.
Upon successfully connecting to the device, the flash algorithm should be selected to download the image into the flash device (Refer to Section J-Link/J-Trace - Setup and Connection).

3.2 KEIL Project Options

3.2.1 Options for Target

The Target tab in the projects Options for Target allows the user to set various project configuration described in the following section.

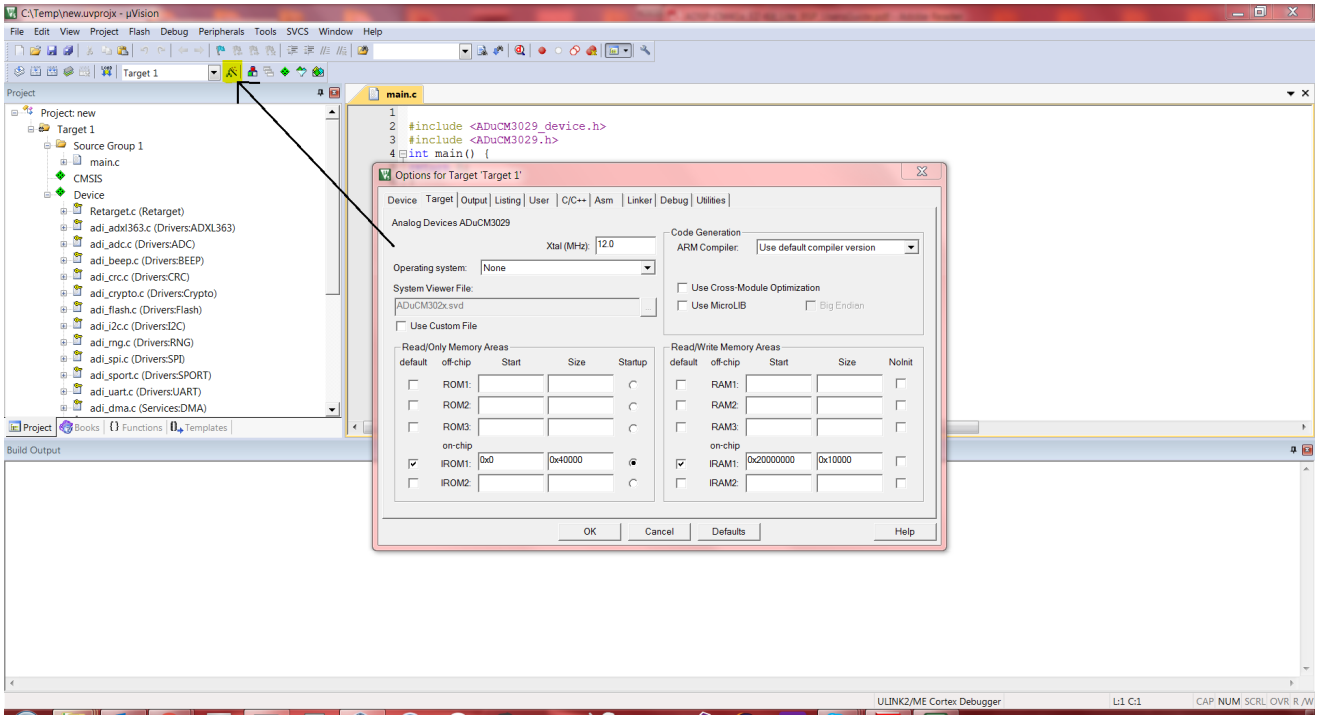

Figure 5. Project Options shows the project configuration options for the ADuCM3029 processor

3.2.2 Device Options

The Device tab in the Target Options allows the user to choose the target processor variant for which the project is being built. This selection is very important because it drives a number of other project settings, such as selecting the correct flash downloader from the pack file (see Figure 6. Device Selection).

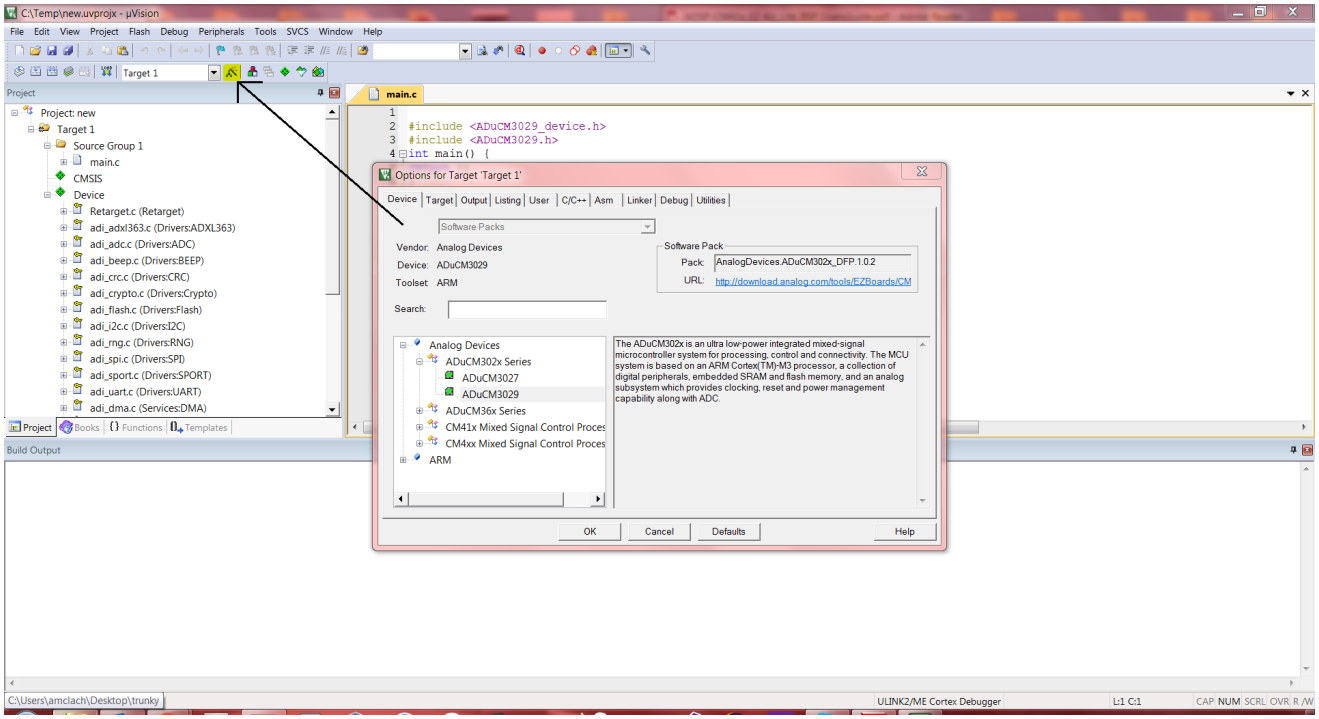

Figure 6. Device Selection

3.2.3 Output

The Output tab in the Options for Target allows the user to set the executable name or create a library file, and select the output file format that can be generated by the ARM CC compiler as shown in the Figure 7. Output Settings.

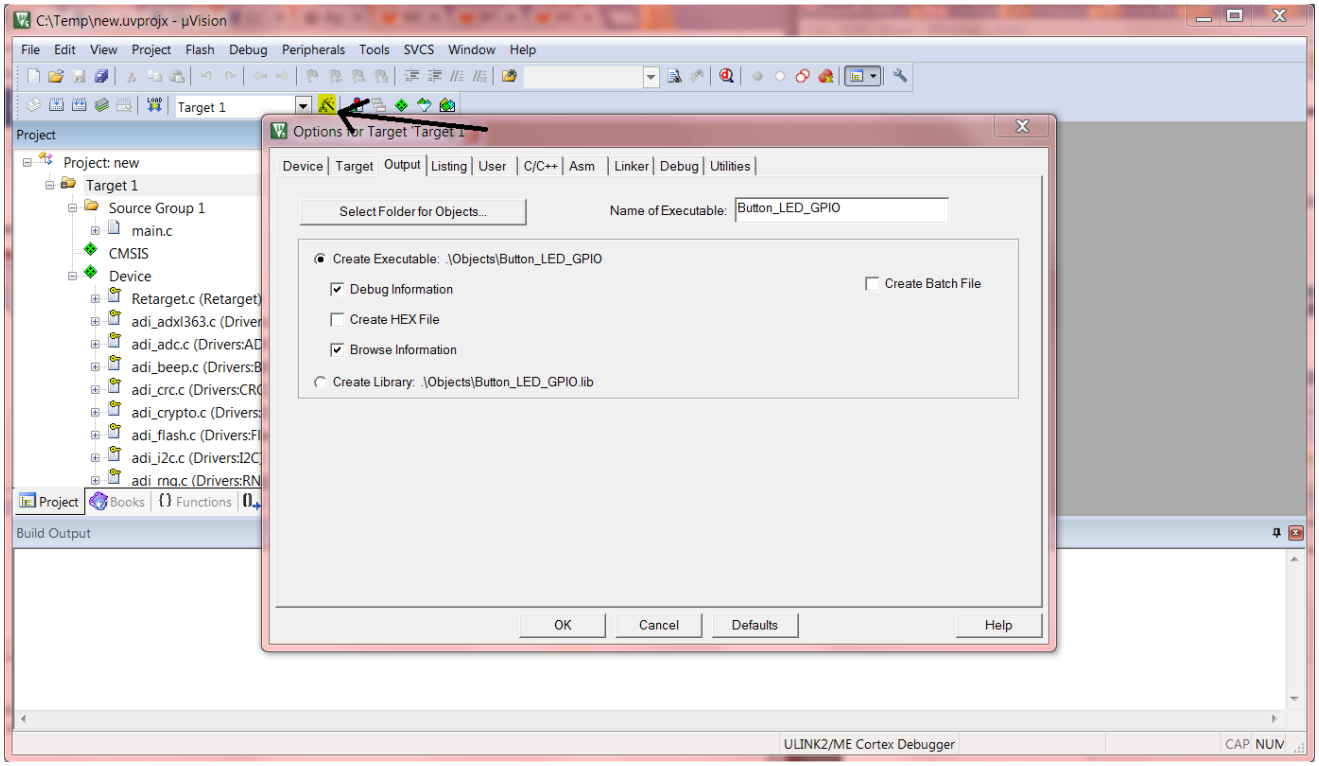

Figure 7. Output Settings

3.2.4 Linker Listing

The Listing tab in the Options for Target allows the user to select the Assembler listing and the linker map file for the KEIL ARM CC compiler as shown in the Figure 8. Linker Listing.

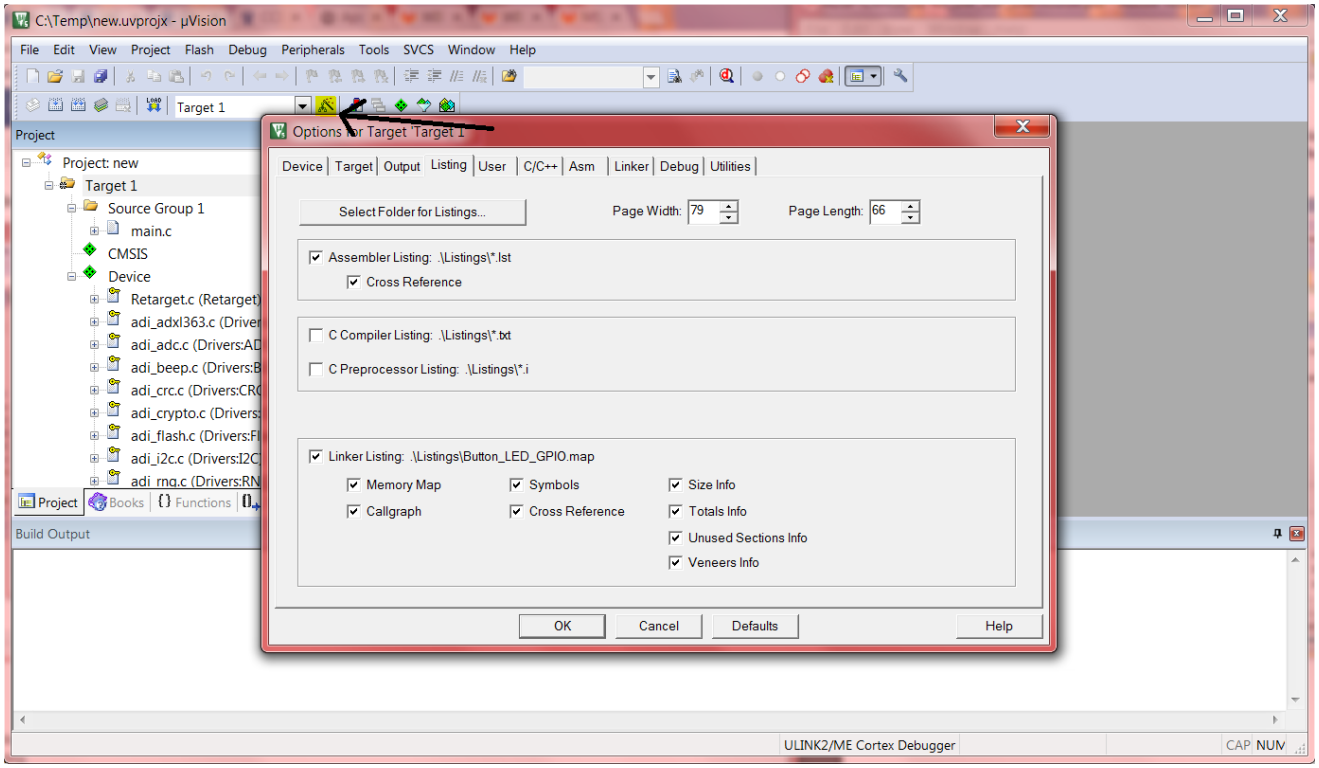

Figure 8. Linker Listing

3.2.5 User Setting

The User tab in the Options for Target allows the user to set any user command for a pre and post build for the KEIL ARM CC compiler as shown in the Figure 9. User Setting below.

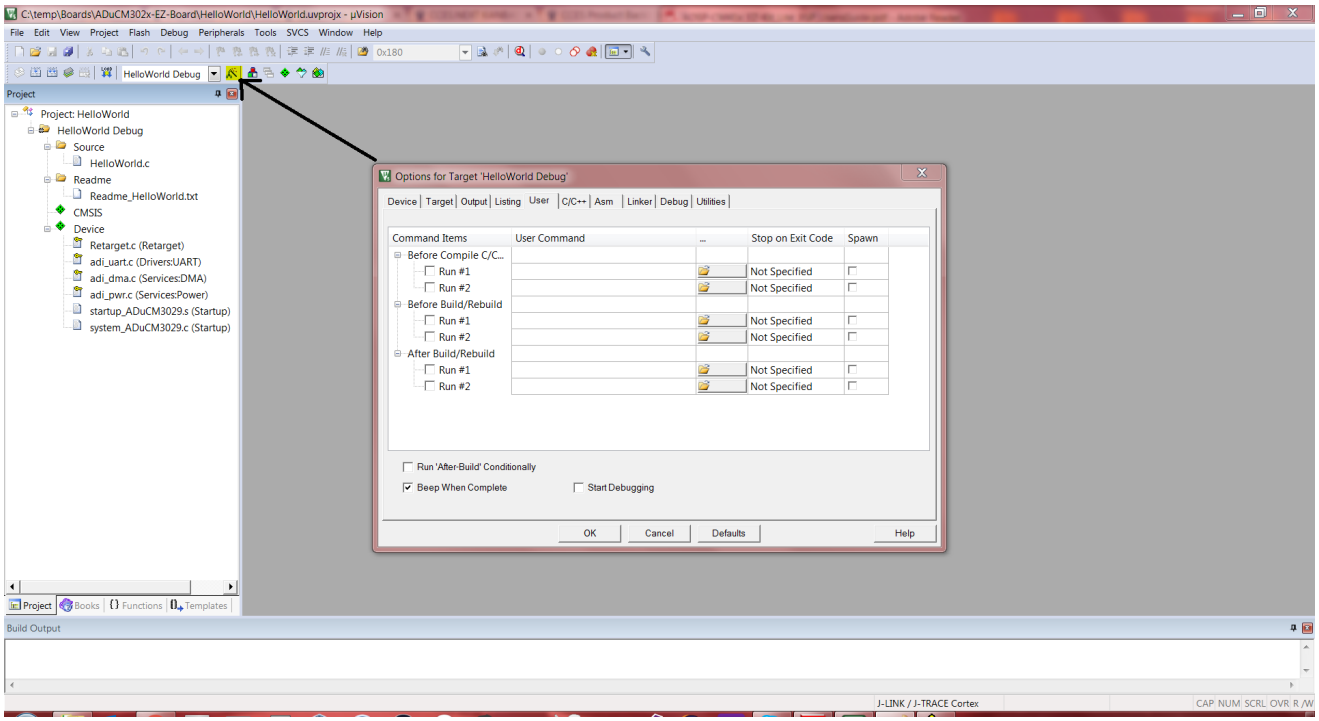

Figure 9. User Setting

3.2.6 C/C++ Setting

The C/C++ tab in the Options for Target allows the user to set any compiler flags, defines, include search paths, optimization levels into the build for the KEIL ARM CC compiler as shown in the Figure 10. C/C++ Setting below.

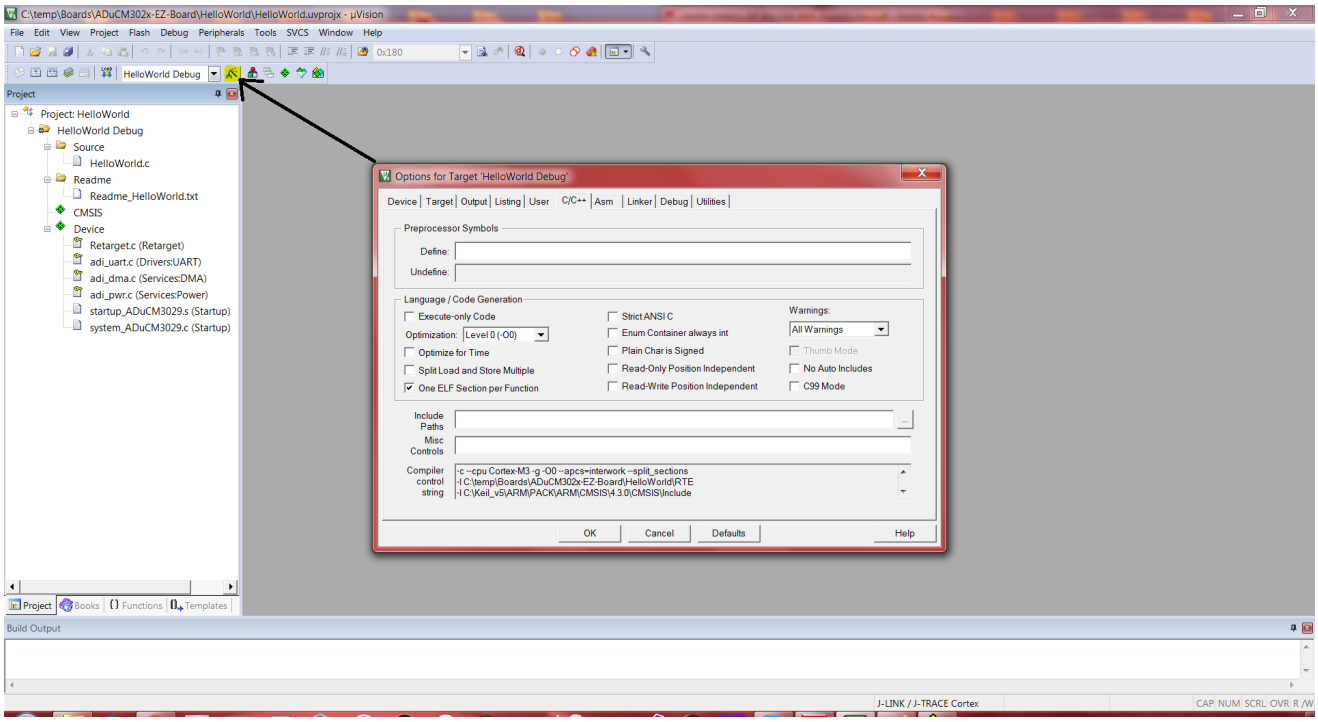

Figure 10. C/C++ Setting

3.2.7 ASM Setting

The ASM tab in the Options for Target allows the user to set any Assembler flags, defines, include search paths, PIP modes into the build for the KEIL ARM CC compiler as shown in the Figure 11. ASM Setting below.

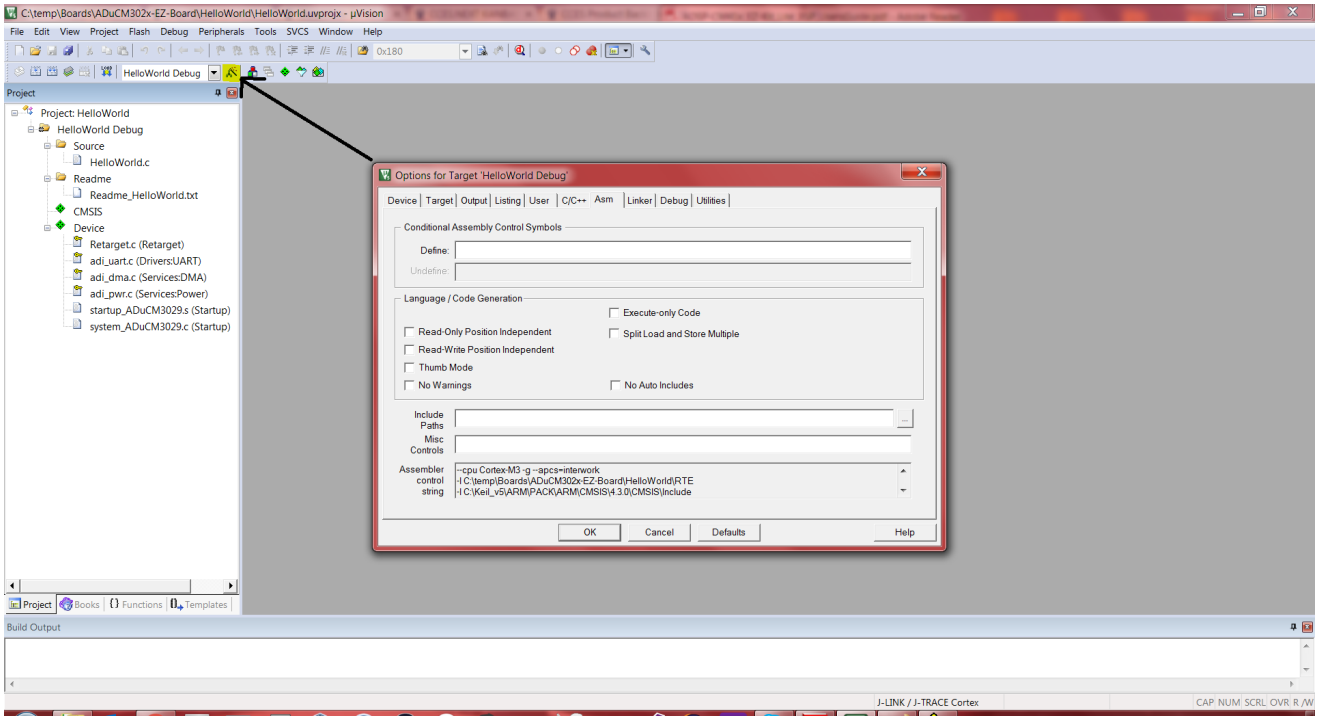

Figure 11. ASM Setting

3.2.8 Linker Setting

The Linker tab in the Options for Target allows the user to set the path for the memory configuration file (SCT file) or to set custom memory configuration in the tab itself for the KEIL ARM CC compiler as shown in the Figure 12. Linker Setting below.

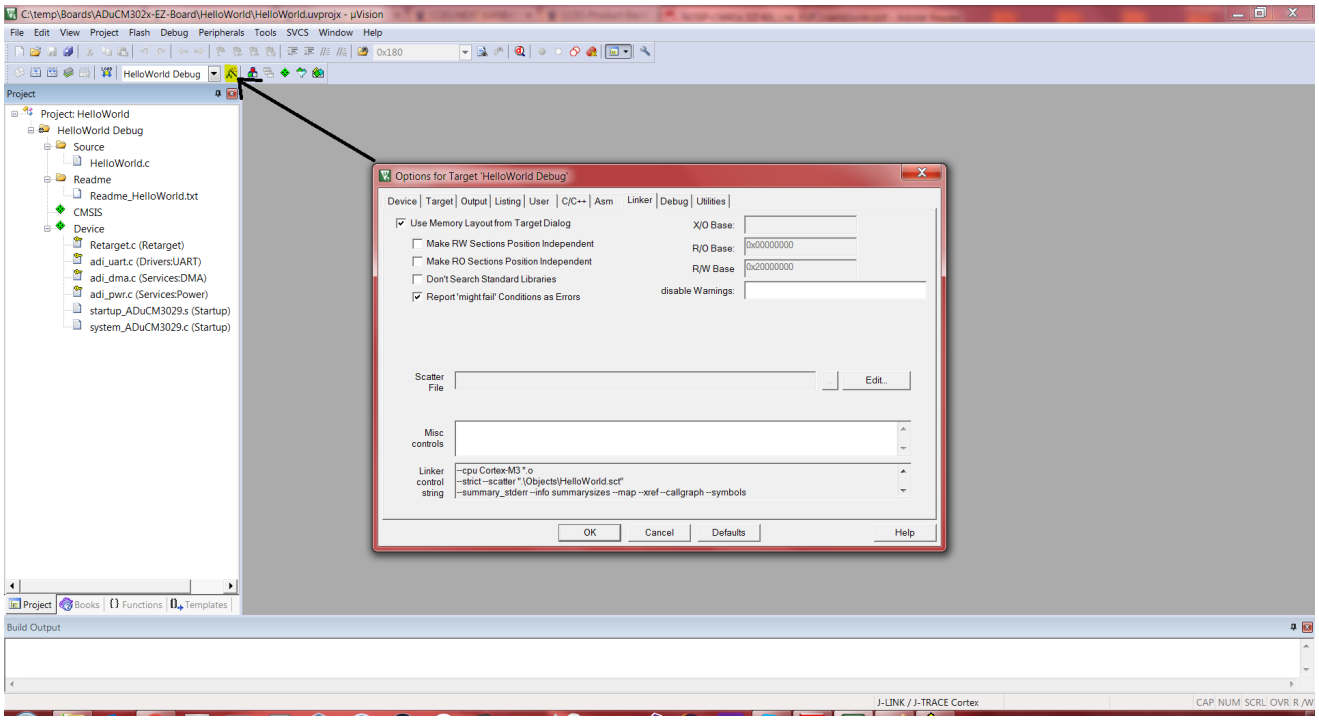

Figure 12. Linker Setting

3.2.9 Debugger Setting

The Debug tab in the Options for Target allows the user to configure the Segger J-Link and its parameters. See Figure 13. Debugger Setup below.

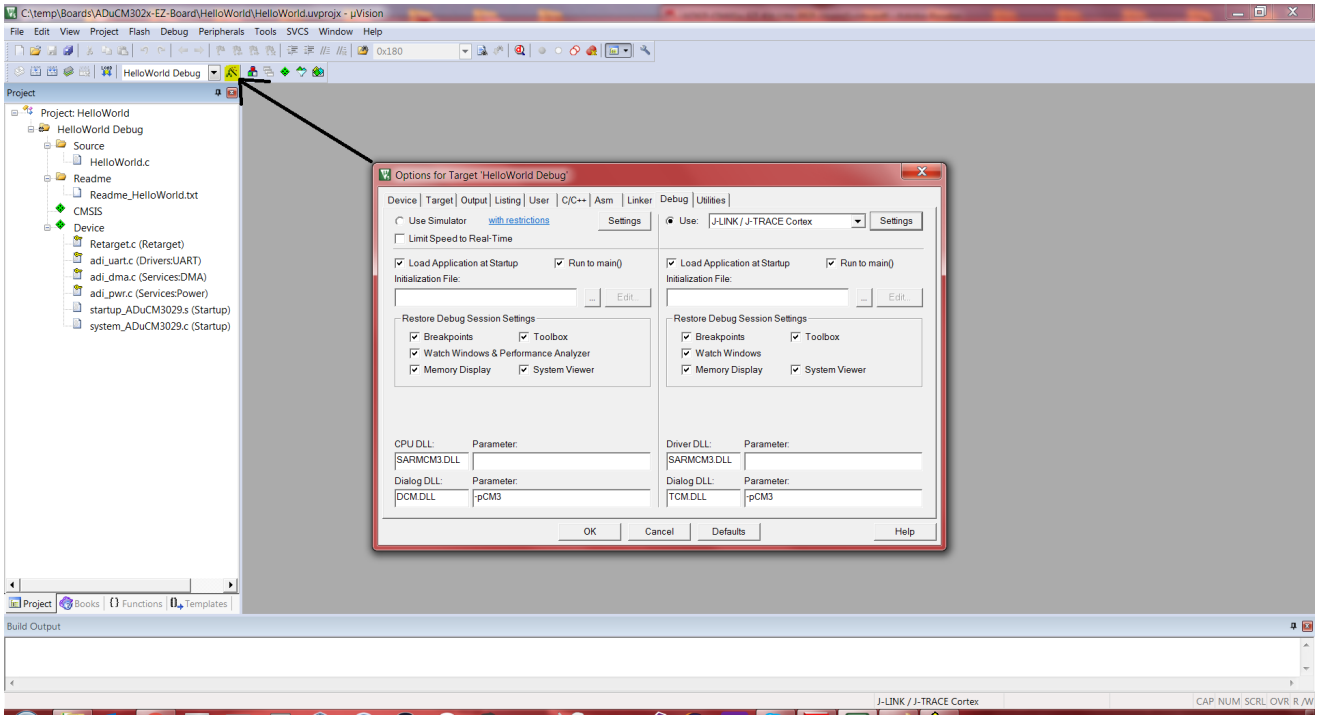

Figure 13. Debugger Setting

3.2.10 Debug Settings (J -Link/JTrace Setup and Connection)

The Settings under the main Debug tab in the Options for Target contains three sub settings (Debug, Trace, Flash Download) these allow the user to configure the Segger J-Link debugger and its modes (SWD/JTAG). Figure 14. Debugger Setting below shows the settings for a sample project.

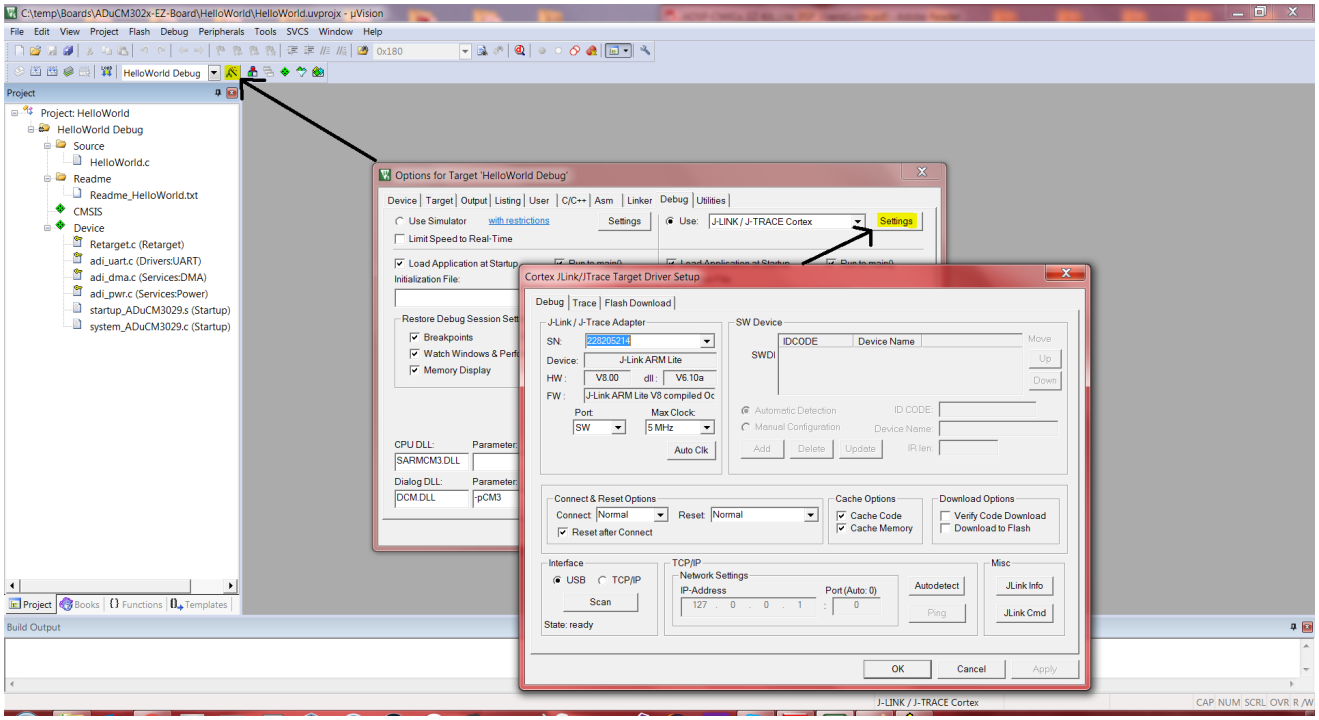

Figure 14. J-Link/JTrace Setting

Trace options can be set-up on a per project basis too. See Figure 15. Trace Setting below.

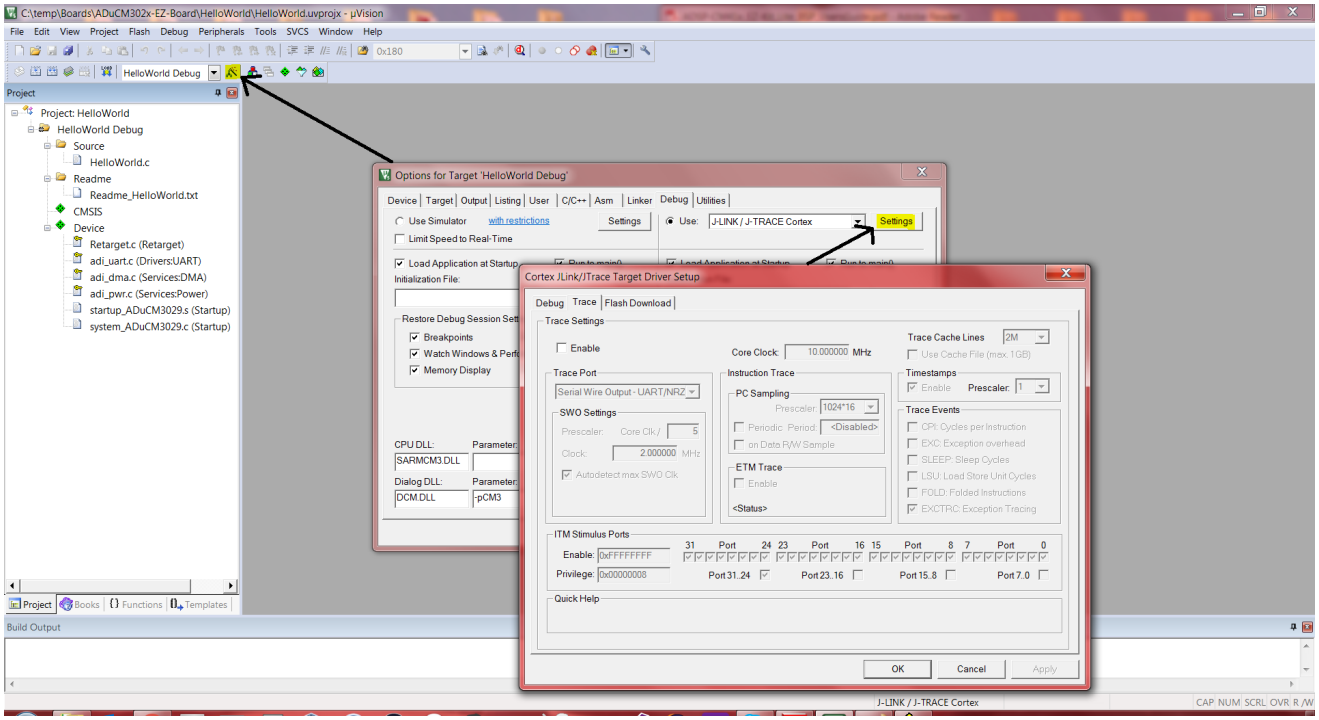

Figure 15. Trace Setting

The Flash Algorithm File has to be added as shown in the figure below on a per project basis. After adding the algorithm the config window will look as shown in the Figure 16. Flash Download Setting.

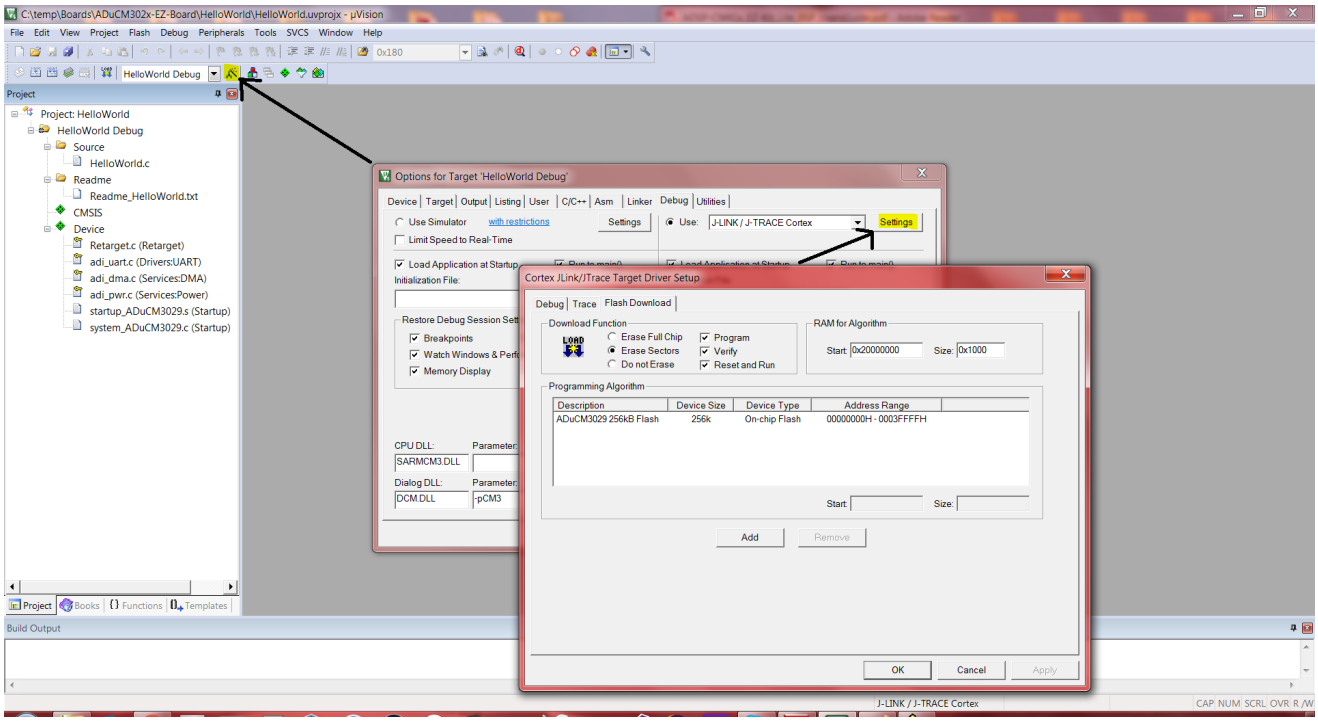

Figure 16. Flash Download Setting

3.2.11 Utilities

The Utilities tab in the Options for Target allows the user to set debugger parameters and choose custom flash utilities (if any) as shown in the Figure 17. Utilities below.

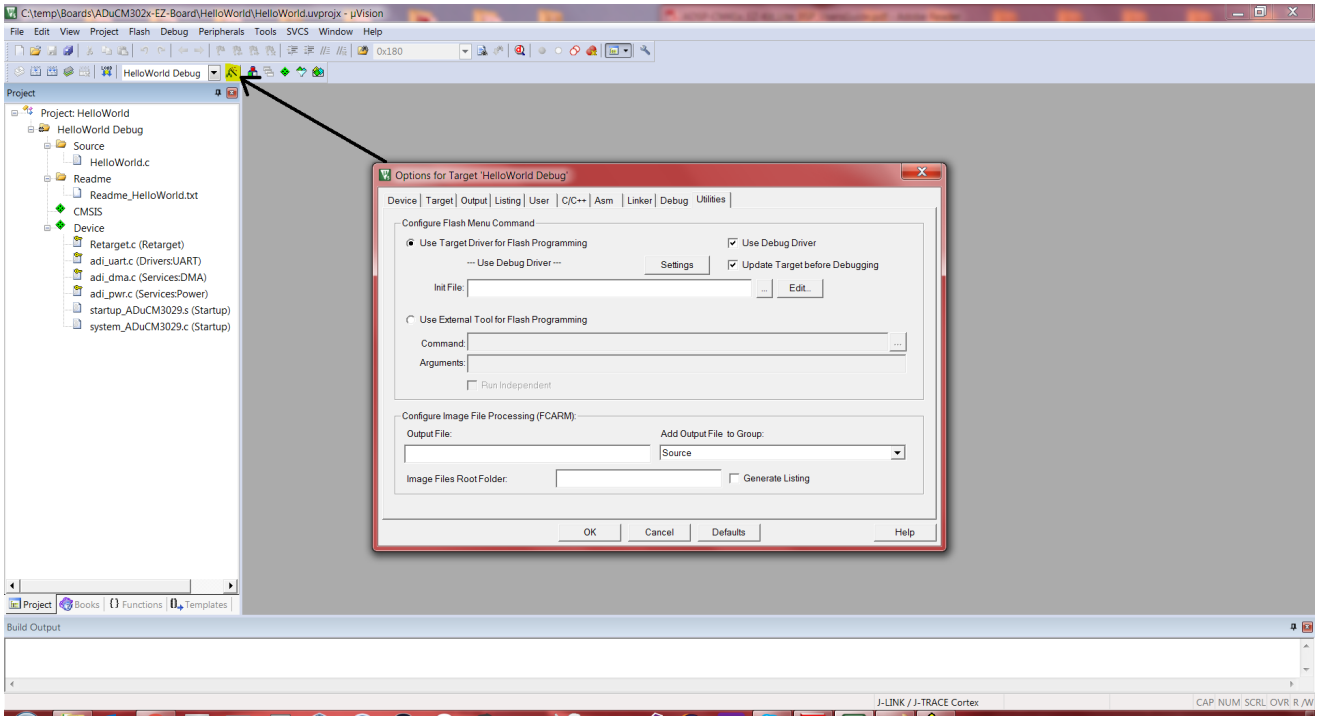

Figure 17. Utilities

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.

In general, the driver sources are located in the **<ADuCM302x_root>**/Source directory. The include file path **<ADuCM302x_root>**/Include must also be specified in the project's compiler/preprocessor options (see section 3.2.3 C/C++ Compiler – Preprocessor).

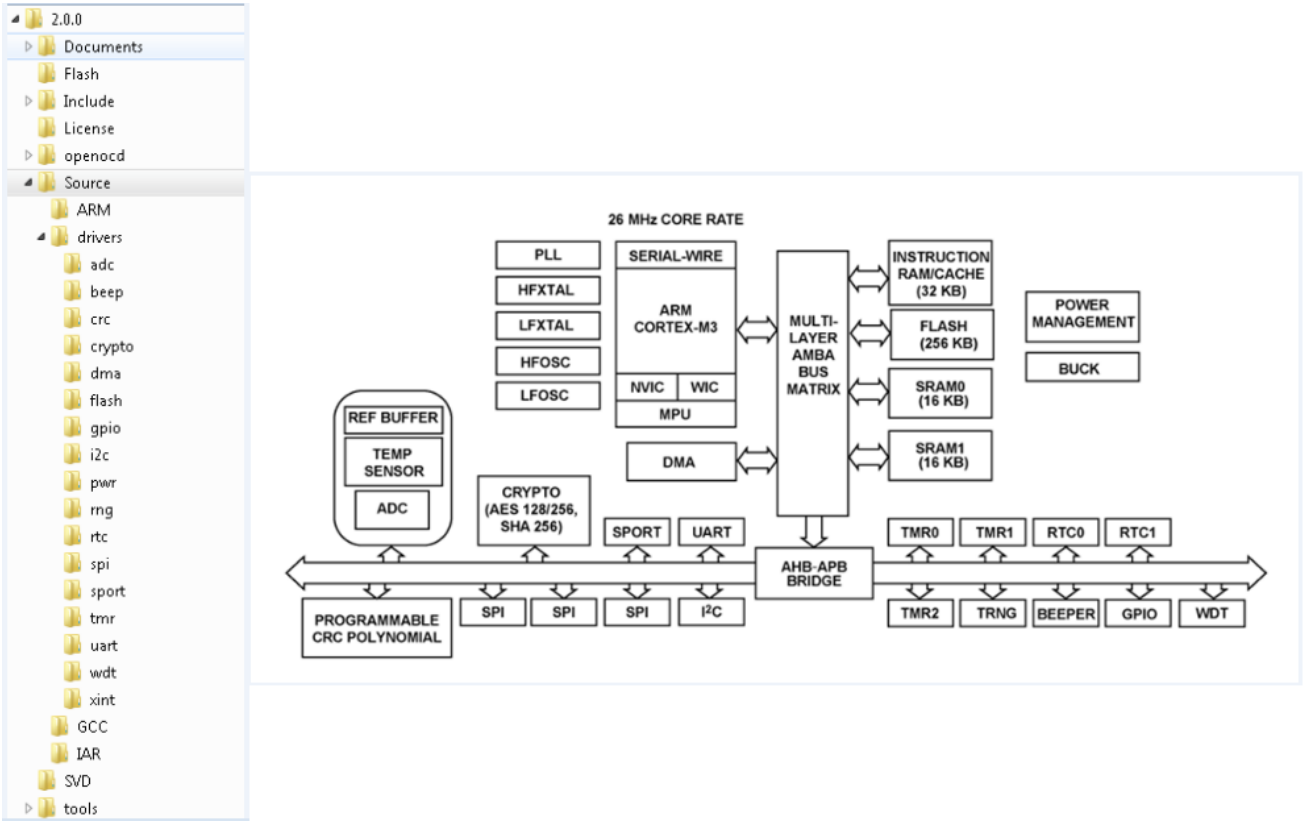

Figure 18. Peripherals and Driver Source

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 KEIL 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.

4.3 System Reset Strategy

All projects require the reset strategy to be set to "normal" in order to enable the emulator to download and debug programs on target hardware properly. The reset strategy is managed in the project options dialog. Selecting the correct reset strategy is both toolchain and emulator specific (see Figure 19. System Reset Setting). To set/verify the reset strategy on the Keil toolchain, click on the "Options for Target", then browse to the sub-dialog for "Debug->Settings". Under the "Reset" drop-down, select the "Normal" option.

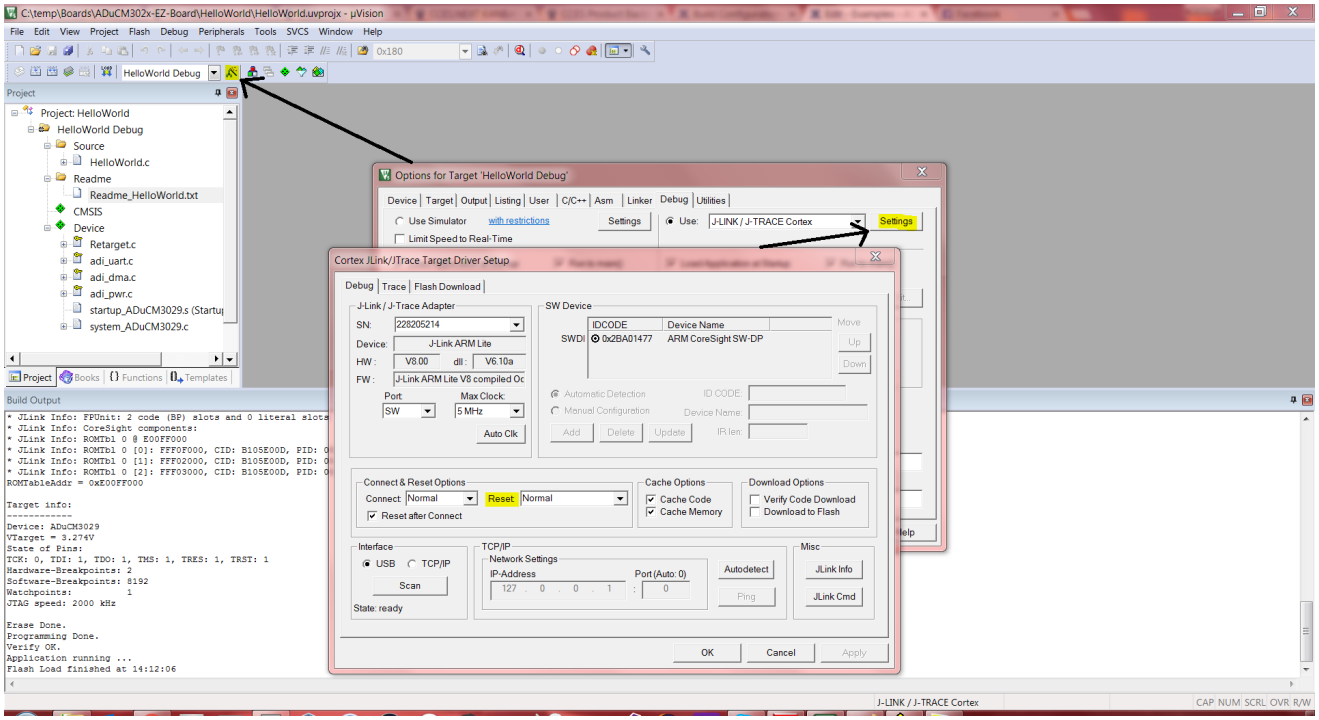

Figure 19. System Reset Setting

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.

Each of the modes is explained in more detail in the sections below. There are some initialization features that are common to all modes of operation.

5.1 Application Initialization

The functions SystemInit() and adi_initpinmux() are used to initialize an application. SystemInit() is required to initialize the ARM Cortex CMSIS infrastructure and adi_initpinmux() initializes the peripheral pin multiplexing, if static pin multiplexing is used (see section 5.1.2 Static Pin Multiplexing). Figure 20. Application Initialization below shows these functions being called from the user application main().

```
int main()
{
   /* Initialize system (required) */
```

```
    SystemInit();  /* system_<Device>.c */

    /* Initialize Pin Multiplexing (optional) */

    adi_initpinmux(); /* Auto-generated source file using PinMux
Utility */

    ...

    return 0;
}
```

Figure 20. Application Initialization

5.2 Static Pin Multiplexing

Included with the ADuCM302x EZ Kit Lite® Board Support Package is a Pin Multiplexing application which is capable of generating code to set all the port MUX and FER registers statically for all peripherals in a single call. The Pin Multiplexing application is a Java application which can be run from a command prompt:

Location: `<ADuCM302x_root>/tools/PinMuxUI`.

There are two versions of the Java application included with the Board Support Package (a 32-bit and 64-bit version). You will need to use the correct java.exe executable to run the application.

- 32-bit version: java -jar PinMuxUI_1.0.0.x_x86.jar
    - To run the 32-bit PinMux Stand-alone Utility, you should use the java.exe that is installed in C:**Program Files (x86)**\Java\jre8\bin (assuming that you have installed Java version 8).
- 64-bit version: java -jar PinMuxUI_1.0.0.x_x86_64.jar
    - To run the 64-bit PinMux Stand-alone Utility, you should use the java.exe that is installed in C:**Program Files**\Java\jre8\bin (assuming that you have installed Java version 8).

After starting the application, you must first select the correct processor type from the top-right drop-down list-box. You are then able to select the desired peripherals to be enabled. The application will not allow conflicting peripherals to be selected. The Generate Code button will create a C source file that sets the GPIO port configuration registers based on the peripherals and functions selected. The C source file should be manually added to your project. The C source file has a function adi_initpinmux() which can be called from the application source (see Figure 21. Pin Multiplexing Application) to set up the port MUX and FER registers.

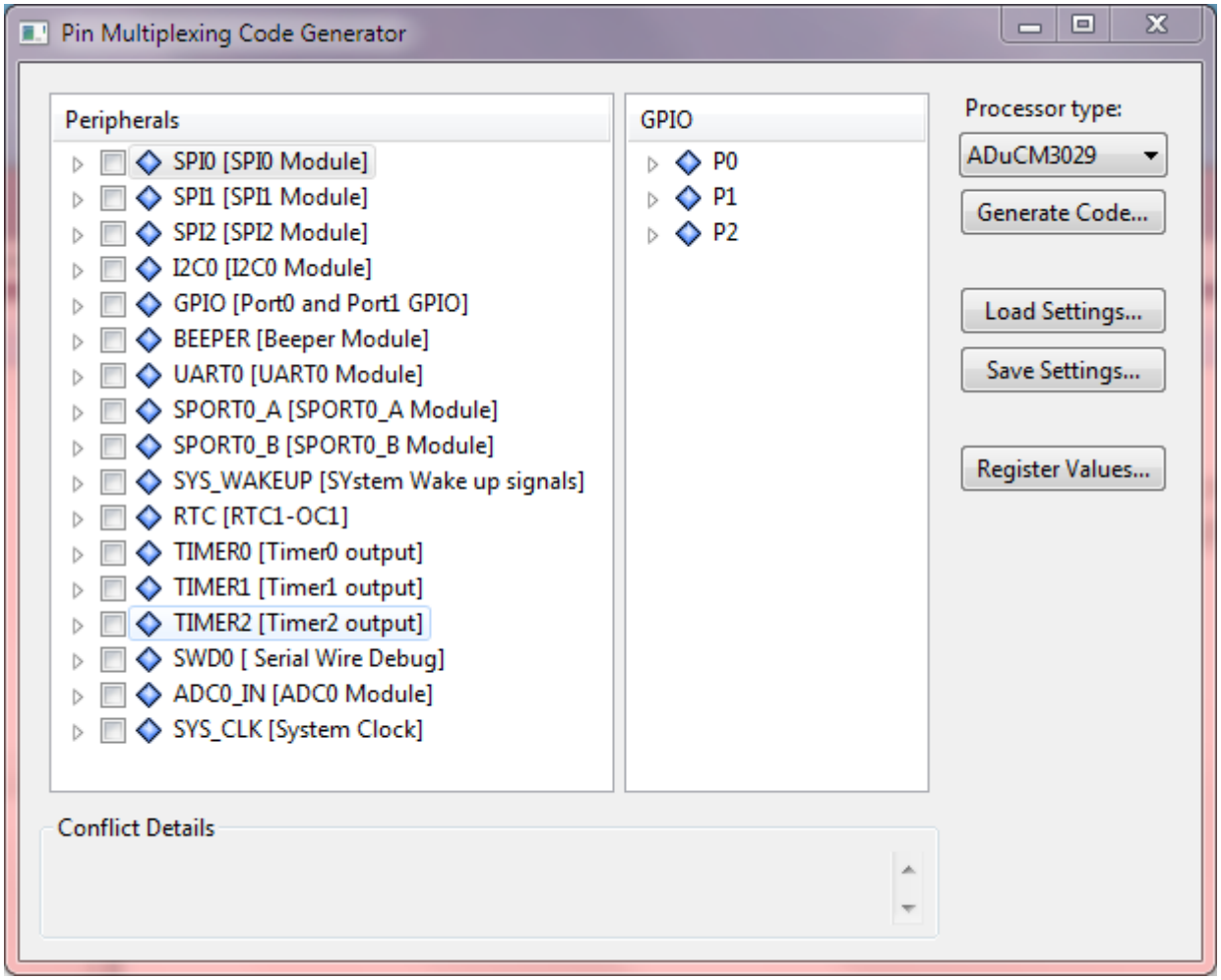

Figure 21. Pin Multiplexing Application

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 EZ Kit Lite® Board Support Package is a `UartDivCalculator` utility which is capable of providing the baudrate configuration values for a specified clock. This utility is available to help the user statically configure their Baudrate. The utility can also provide the baudrate configuration values for a set of baudrates.

5.4 Driver Include Files

The C/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

<ADuCM302x_root>/Include/config

to be added from the BSP installation. Applications may need to augment the pre-processor 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 <ADuCM302x_root>/Include/config/adi_global_config.h, which needs to be added to a new project. 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 to 0 as shown in figure 22 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 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 first copying them to the local project directory and then modifying the files as needed.

5.5.3 Configuration Overrides

The default factory configuration files can be edited directly for global changes for all applications. Individual overrides can be made by copying any configuration file(s) to the local project directory and making the changes there. It is recommended to make a backup of the default file set before making global changes.

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

1. Globally by modifying the default factory default files for global changes.
2. Locally by copying the driver's default configuration file into the application's source folder. Local edits can then be applied. You must ensure that the application's source folder appears before the default `Include\config` folder in the project's pre-processor include path option settings. *Note: Local overrides (if any) are the recommended override method.*
3. 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

Complete documentation for the DFP is listed in the 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 also available in HTML format as shown in Figure 23. 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.

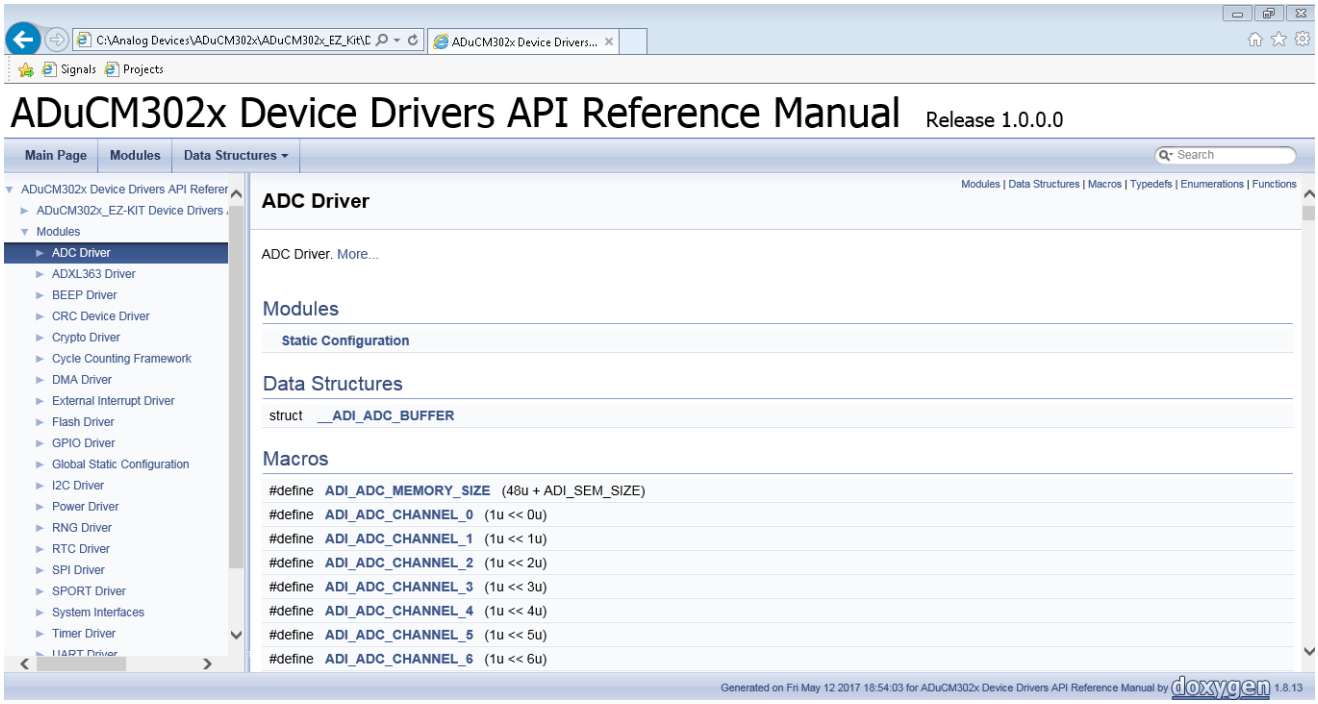

Figure 23. Device Driver Documentation

6.2 Appendix

6.2.1 CMSIS

The ADuCM302x Device Family Pack is compliant with the Cortex Microcontroller Software Interface Standard. 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* [III] reference or the www.arm.com website for complete CMSIS details.

6.2.2 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.

6.2.3 Startup_<Device>.c Content

The `<ADuCM302x_root>/Source/ARM/startup_<Device>.s` 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

6.2.4 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 that is required to be called at the very beginning of user main(), immediately after the C Run-time Library has setup the system and called user main(). Nothing else should be done in user main() until *after* the SystemInit() call is complete.

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.