ADUCM350BBCZ SOFTWARE USER'S GUIDE

ANALOG DEVICES, INC.

www.analog.com

Table of Contents

1 Introduction 6
1.1 Purpose 6
1.2 Scope of this Manual 6
1.3 Acronyms and Terms 6
1.4 Conventions 7
1.5 References 8
1.6 Additional Information 9

2 Product Overview 9
2.1 Software System Overview 9
2.2 Hardware System Overview 10

3 Installation Components 11
3.1 IAR Product Activation Files 12
3.1.1 Device Database Files 13
3.1.2 Linker Control Files 14
3.1.3 Flash Loader Configuration Files 15
3.1.4 Debugger Configuration Files 15
3.1.5 GUI-Based Register Editing 15
3.1.6 Flash Loader Sources 16
3.2 IAR Project Options 17
3.2.1 General Options – Target 17
3.2.2 General Options – Library Configuration 18
3.2.3 C/C++ Compiler – Preprocessor 19
3.2.4 Debugger - Setup and Download 20
3.2.5 J-Link/J-Trace - Setup and Connection 21

4 ADuCM350BBCZ System Overview 22
4.1 Block Diagram and Driver Layout 22
4.2 Boot-Time CRC Validation 23
4.3 Run-Time Parity Validation 25
4.4 Generating CRC and Parity Info 25
4.4.1 Specify Linker KEEP Directives 25
4.4.2 Specify Linker Command Line Settings 26
4.4.3 Specify iElfTool Post-Build Settings 26
4.4.4 Generic iElfTool Command Line Description 27
4.5 System Reset Strategy 28

5 Build Configurations 29
5.1 Application Configuration 29

5.1.1 Application Initialization....30
5.1.2 Static Pin Multiplexing....30
5.1.3 Dynamic Pin Multiplexing....32
5.1.4 Driver Include Files....32
5.1.5 uC/OS-II Include Files....33
5.2 Driver Configuration....34
5.2.1 Global Configuration....34
5.2.2 Configuration Defaults....35
5.2.3 Configuration Overrides....35
5.2.4 IVT Table Location....36
5.2.5 Interrupt Callbacks....36
5.3 RTOS Configuration....37
5.3.1 RTOS Aware Drivers....39
5.3.2 RTOS Aware Drivers and the need for OSAL support....39
5.3.3 RTOS Aware Driver and IVT Table Location....39

6 Examples....39
6.1 Build and Run....40
6.2 Self-Test Debug Code....40
6.3 Capturing STDIO from Examples....40
6.4 RTOS Builds....41

7 USB Software....42
7.1 Software Requirements for USB....42
7.2 USB Examples....42

8 Device Driver API Documentation....46

9 Appendix....47
9.1 CMSIS....47
9.2 Interrupt Vector Table....47
9.3 Startup.c Content....49
9.4 System.c Content....50

List of Figures

Figure 1 Software Overview 9
Figure 2 Hardware Overview 10
Figure 3 Eval-ADuCM350EBZ Evaluation Board 10
Figure 4 Installation 12
Figure 5 Processor Device Selection 17
Figure 6 Library Configuration 18
Figure 7 IAR Additional Include Paths 19
Figure 8 Debugger Setup and Downloader 20
Figure 9 J-Link Setup and Connection 21
Figure 10 Peripherals and Driver Source 22
Figure 11 Configuring Pre-Boot Checksum 24
Figure 12 - Application Initialization 30
Figure 13 - Pin Multiplexing Application 32
Figure 14 – IAR: C/C++ Compiler Includes for uC/OS-II RTOS 33
Figure 15 Global Configuration File Contents 34
Figure 16 - RTOS Project Files 38
Figure 17 - Micrium Environment Variable 41
Figure 18 - USB PHDC Components 43
Figure 19 - USB MSC Components 44
Figure 20 – USB CDC Components 45
Figure 21 - Device Driver Documentation 46

Copyright, Disclaimer & Trademark Statements

1 Introduction

1.1 Purpose

This document describes the ADuCM350BBCZ Software Package and its use on the Eval-ADUCM350EBZ target board. The ADuCM350BBCZ 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 ADuCM350BBCZ Software Package. This document explains what's 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 ADuCM350BBCZ processor. This document assumes background in ADI's ADuCM350BBCZ processor.

1.3 Acronyms and Terms

ADI	Analog Devices, Inc.
API	Application Programming Interface
ARM	Advanced RISC Microcontroller
CMSIS	Cortex Microcontroller Software Interface Standard
Cortex	A series of ARM microcontroller core designs
CRC	Cyclic Redundancy Check
ELF	Executable and Linkable Format
HRM	Hardware Reference Manual
ISR	Interrupt Service Routine
IVT	Interrupt Vector Table
JTAG	Joint Test Action Group
NVIC	Nested Vectored Interrupt Controller
OSAL	Operating System Abstraction Layer
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 three important installation locations: the ADuCM350BBCZ Software Package, the IAR tool chain installation root, and the Micrium install root. Each of these packages can be installed in various places, so we refer to them here as follows:

- **<ADuCM350BBCZ_root>**

  The ADuCM350BBCZ Software Package installer places the product at default location `C:\Analog Devices\ADuCM350BBCZ\Eval-ADUCM350EBZ` (where, `Eval-ADUCM350EBZ` is the target board for the install), but the install location may vary depending on user preferences.

- **<EWARM_root>**

  The default IAR Embedded Workbench for ARM installer places the product at location `C:\Program Files (x86)\IAR Systems\Embedded Workbench 7.0`.

  The user may want to amend the default IAR install location to something more specific, e.g., `"…7.30.1"`. Do this by clicking on the "Choose Destination Location" "Change…" button and making the desired change.

- **<Micrium_root>**

  The default Micrium installer places the product at location `C:\Micrium\Software`, but the install location may vary depending on user preferences.

1.5 References

I. Analog Devices: `<ADuCM350BBCZ_root>/doc`

   a. ADuCM350BBCZ Devices Drivers Release Notes

   b. ADuCM350BBCZ Devices Drivers Getting Started Guide

   c. ADuCM350BBCZ Software User's Guide (this document)

   d. ADuCM350BBCZ Device Drivers API Reference Manual (hyperlinked)

II. Analog Devices: http://www.analog.com/aducm350_design_files

   a. UG-587 Product User Guide / Hardware Reference Manual

   b. ADuCM350 Data Sheet

   c. ADuCM350 Application Notes

      i. AN-1262 Serial Download Protocol

      ii. AN-1263 Security Integrity of the ADuCM350

      iii. AN1282 Profiling the ADuCM350 Supply Current in an Example Application

   d. UG-668 Eval-ADUCM350EBZ User Guide

III. IAR: `<EWARM_root>\arm\doc` [http://www.iar.com]

   a. IAR Embedded Workbench® for ARM

   b. IDE Project Management and Building Guide for ARM

   c. C/C++ Compiler Development Guide for ARM

   d. C-Spy Debugging Guide for ARM

IV. *The Definitive Guide to the ARM CORTEX-M3 and CORTEX-M4 Processors*, Joseph Yiu, 3rd ed.

   Every Cortex programmer's bible; a must-have reference.

V. Micrium [http://micrium.com]

   a. uC/OS-II RTOS for ARM Cortex-M3

   b. uC/OS-II User's Manual

   c. uC/USB-Device-V4 User's Manual

VI. SEGGER J-Link OB Emulator [https://www.segger.com/jlink-ob.html]

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

2 Product Overview

2.1 Software System Overview

The ADuCM350BBCZ Software Package provides files needed to write application software for the ADuCM350BBCZ processor. The product consists of a boot kernel, startup, system and driver source code, driver configuration settings, driver libraries, sample applications and associated document (see Figure 1 Software Overview).

The ADuCM350BBCZ Software Package is designed to work with IAR Embedded Workbench for ARM[III.a] and optionally, the Micrium uC/OS-II RTOS[V].

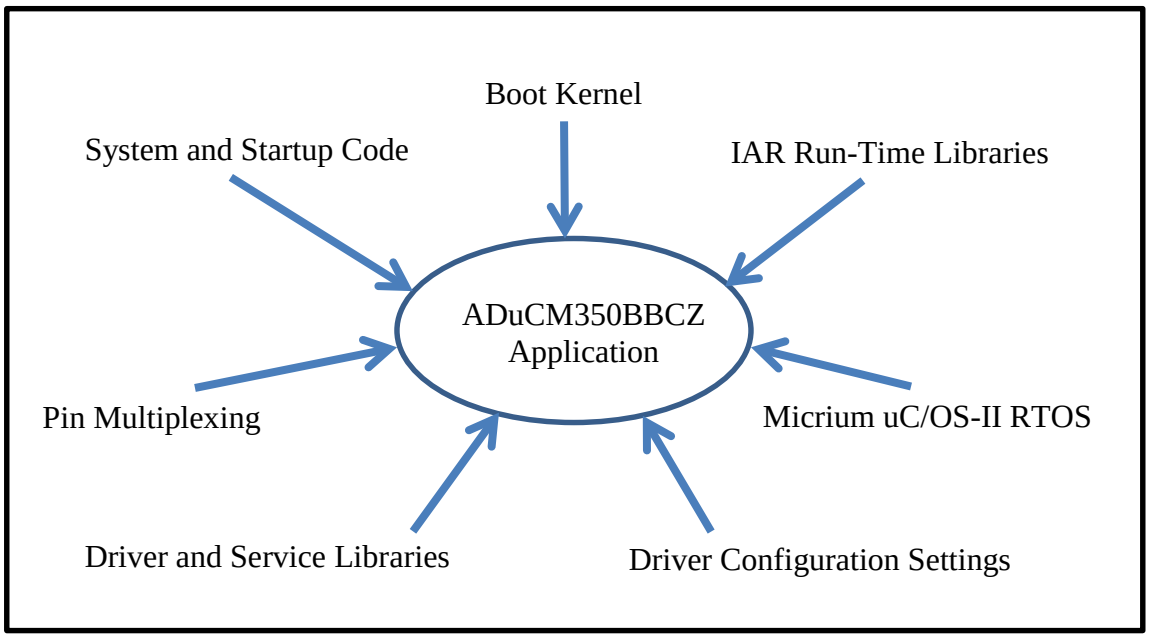


Figure 1 Software Overview

2.2 Hardware System Overview

The examples provided in the ADuCM350BBCZ Software Package run on the Analog Devices' Eval-ADUCM350EBZ evaluation board. The evaluation board is connected to the host PC using an Analog Devices' USB-SWD/UART-EMUZ emulator board. This board uses a Segger J-Link OB[VI] emulator and connects to the host PC with a mini-USB cable. External I/O signals and system hardware are connected to the evaluation board connectors as shown in Figure 3 Eval-ADuCM350EBZ Evaluation Board. See Eval-ADUCM350EBZ User Guide (UG-668)[II.d] for more information on the hardware configuration.

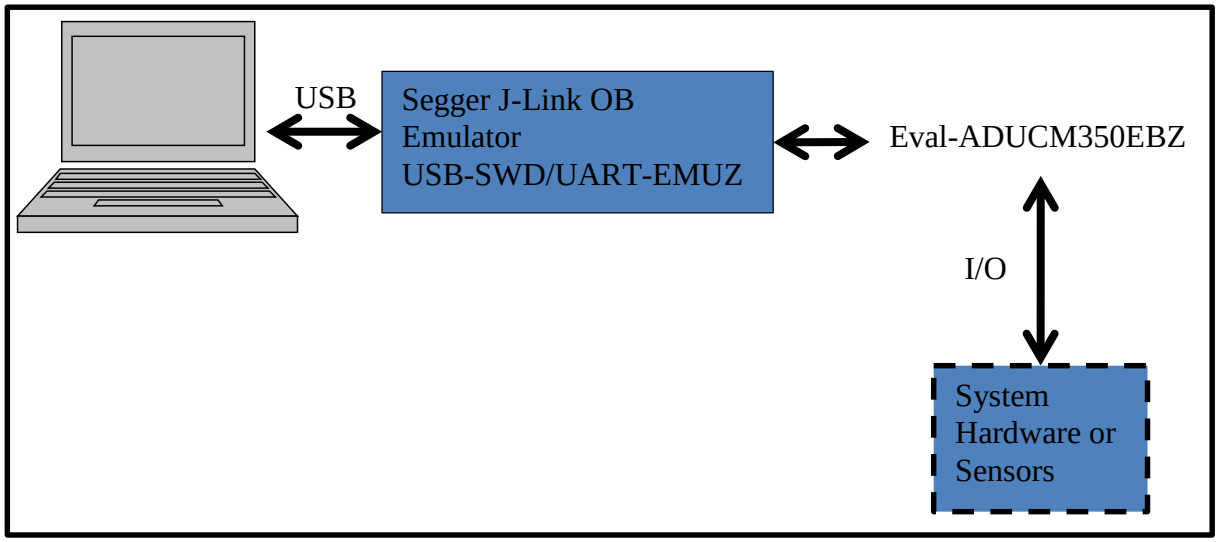


Figure 2 Hardware Overview

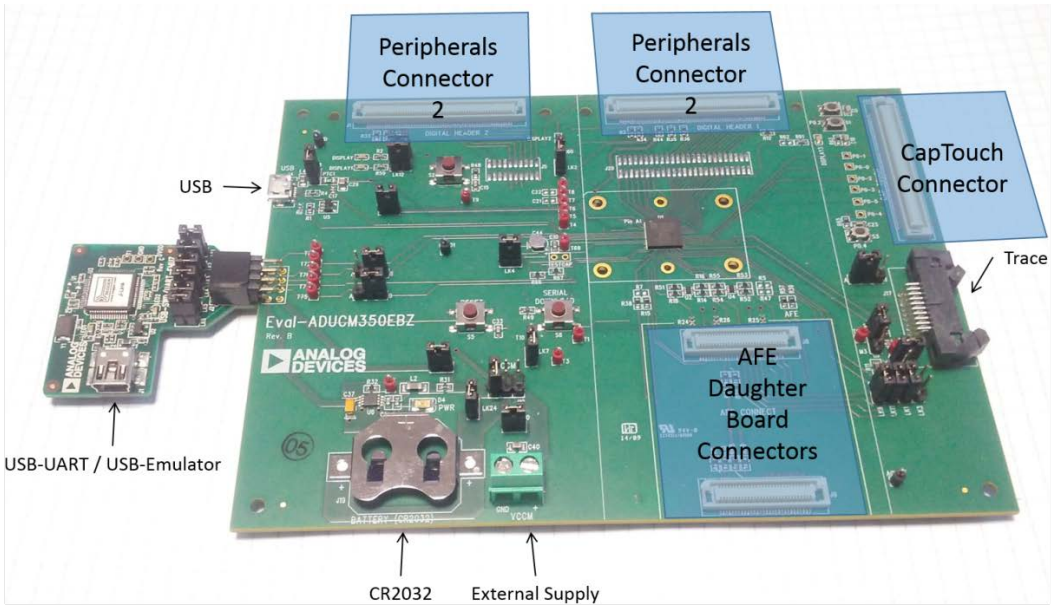


Figure 3 Eval-ADuCM350EBZ Evaluation Board

3 Installation Components

- IAR Embedded Workbench 7.30.1[III.a] must be purchased and installed prior to installing the ADuCM350BBCZ Software Package. Follow the instructions in the IAR Embedded Workbench for ARM product installation manual.
- The Micrium uC/OS-II RTOS for ARM Cortex-M3 [V] is not included in the ADuCM350BBCZ Software Package installation. If it is desired to use the Micrium RTOS it must be purchased and downloaded directly from Micrium.
- The Micrium uC/USB-Device-V4 [V.c] is not included in the ADuCM350BBCZ Software Package installation. If it is desired to use the Micrium USB Device software it must be purchased and downloaded directly from Micrium.
- The Micrium uC/USB-Device-CDC (Communications Device Class) is not included in the ADuCM350BBCZ Software Package installation. If it is desired to use the Micrium USB Device CDC class software it must be purchased and downloaded directly from Micrium.
- The Micrium uC/USB-Device-MSC (Mass Storage Class) is not included in the ADuCM350BBCZ Software Package installation. If it is desired to use the Micrium USB Device MSC class software it must be purchased and downloaded directly from Micrium.
- The Micrium uC/USB-Device-PHDC (Personal Healthcare Device Class) is not included in the ADuCM350BBCZ Software Package installation. If it is desired to use the Micrium USB Device PHDC class software it must be purchased and downloaded directly from Micrium.

The ADuCM350BBCZ Software Package installer places files in two areas (see Figure 4 Installation):

- IAR configuration files are placed in the IAR Embedded Workbench installation folder: `<EWARM_root>`. The IAR configuration files consist of files for linking, flash loading, debugging, etc.

- The ADuCM350BBCZ Software Package files (startup code, device drivers, libraries, examples, tools, documentation, etc.) are placed in the Analog Devices installation folder: `<ADuCM350BBCZ_root>`.

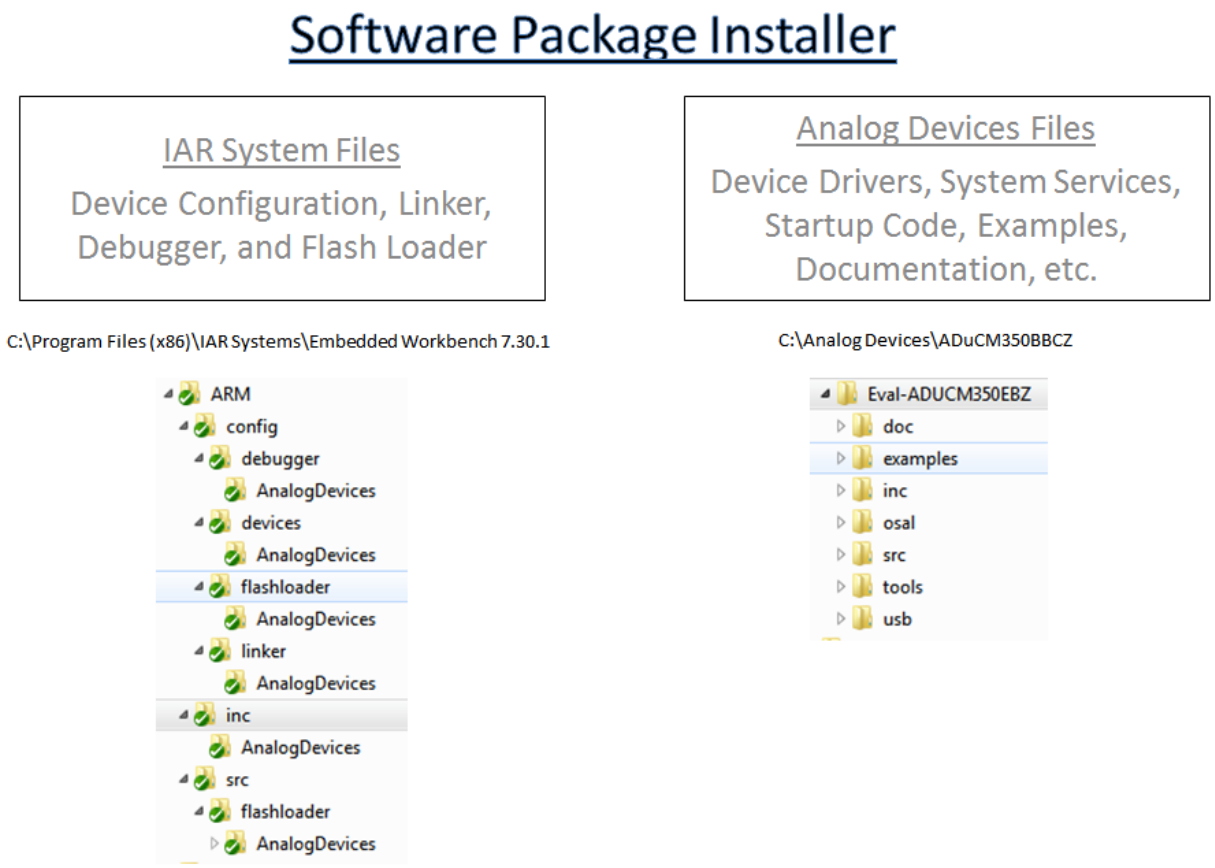


Figure 4 Installation

3.1 IAR Product Activation Files

This section documents the IAR-specific details of the ADuCM350BBCZ Software Package installer. A working knowledge of the IAR tool chain and environment is assumed. See the IAR reference materials for details of installing, configuring and using the IAR tools.

The following is a brief list of IAR "product activation" files added by the ADuCM350BBCZ Software Package installer, which are installed as required into the `<EWARM_root>` directory structure:

- Device Database Files
- Linker Control Files
- Flash Loader Files
- Debugger Register Display Files
- GUI-Based Register Editing files
- Flash Loader and Source files

3.1.1 Device Database Files

The `<EWARM_root>\ARM\config\devices\AnalogDevices` folder contains *.menu and *.i79 files.

Note: These two files map out the entire product activation file set which tell the IAR tool chain how to handle the ADuCM350BBCZ processor; selecting the target processor in the main project options window sets the default files for the entire project.

The `<EWARM_root>\ARM\config\devices\AnalogDevices\ADuCM350BBCZ.menu` files configure:

- Database Entry Presented for User Target Selection
- Points to the "*.i79" file

The ADuCM350BBCZ .i79 files configure:

- Chip Attributes (endian, instruction set, floating point, emulator scripts, etc.)
- Core Architecture Selection
- Default Debugger Display File
- Default Linker Control File
- Default Flash Loader File

3.1.2 Linker Control Files

To link and locate an application in memory according to your requirements refer to the 'linker configuration file' section in the IAR C/C++ Development Guide for ARM8[III.c].

The `<EWARM_root>\ARM\config\linker\AnalogDevices` folder contains *.icf files. These files are used to:

- Define Memory "Regions" (size, location, alignment, etc.)
    - Flash Area, Internal SRAM Code, SRAM Data, Special-Purpose, etc.
    - Internal SRAM Bank Partitioning (2 x 16k)
- Define "Blocks" for Specific Tasks
    - Runtime Stack, Heap Space, etc.
    - Size & 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

The `<EWARM_root>\ARM\config\linker\AnalogDevices\ADuCM350BBCZ.icf` file:

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

3.1.3 Flash Loader Configuration Files

The IAR flash loader folder `<EWARM_root>\ARM\config\flashloader\AnalogDevices` folder contains the flash loader *.board and *.flash files. 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.

Flash Loader Operation:

1. IDE Generates User's Target Code Executable Image
2. IDE → Target Code Image → **Flash Loader** → Flash

- FlashADuCM350BBCZ.mac (macro file to optionally process callbacks)
- FlashADuCM350BBCZ.out (Flash Loader executable)
- FlashADuCM350BBCZ.board (points to ".flash" file)
- FlashADuCM350BBCZ.flash (flash parameters and executable pointer)

3.1.4 Debugger Configuration Files

The `<EWARM_root>\ARM\config\debugger\AnalogDevices` folder contains *.ddf and script files. These files are used to configure debugger attributes:

- Register Display Information
  - IDE Register View Menu
  - ioADuCM350BBCZ.ddf
- Register characterization (Name, Address, Width, Format, Bit field extent)
- Peripheral Grouping of Registers (ADC, Counter, DMA, GPIO …)

3.1.5 GUI-Based Register Editing

The `<EWARM_root>\ARM\inc\AnalogDevices` folder contains *.h files. These files are used to configure:

- Register Access Information
  - Is used for Interactive IDE Register Edits During Debug
  - Is defined ioaducm50MMCZ.h
  - Defines various "IO" statements characterizing register attributes (name, address, width, and access mode). This information is used to by the IDE to modify register content in response to user edits in the debugger's register display windows.

3.1.6 Flash Loader Sources

The `<EWARM_root>\ARM\src\flashloader\AnalogDevices\FlashADUCM350BBCZ` folder contains the files used to build the flash loader utility. These files are not needed to use the default flash loader contained in the main IAR `config` directory; they are only provided as a reference or if the flash loader needs to be modified and/or rebuilt.

- Main Source Directory
    - Flash Sources
    - Flash Project Files
    - Flash Linker Control File

3.2 IAR Project Options

3.2.1 General Options – Target

The General Options Target tab is used to select the processor variant for the project. This selection is very important because it drives a number of other project settings, such as selecting the correct flash downloader, linker configuration file, etc. There is one choice for the Analog Devices ADuCM350BBCZ processor. Figure 5 Processor Device Selection, shows the device selection for the ADuCM350BBCZ processor.

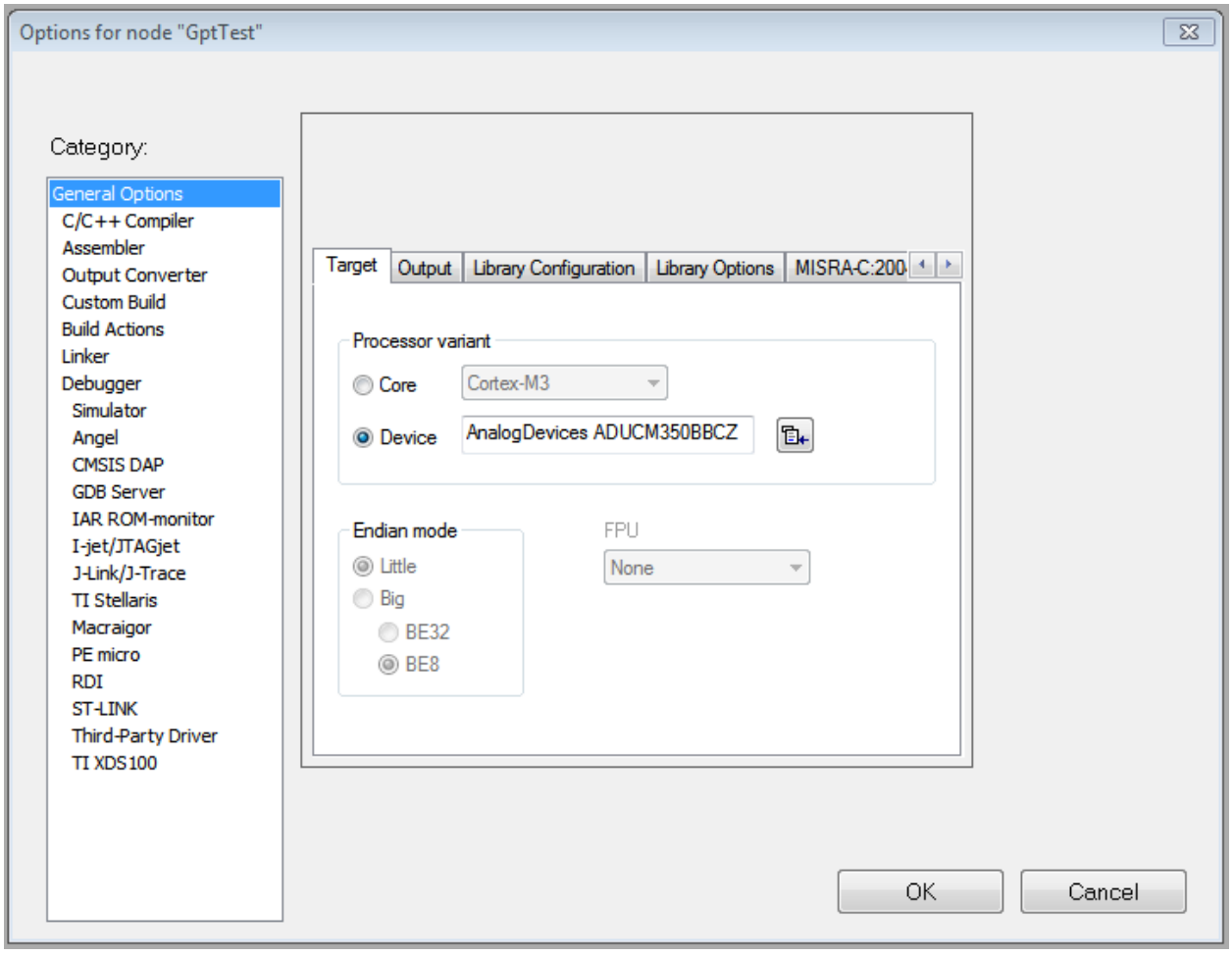


Figure 5 Processor Device Selection

3.2.2 General Options – Library Configuration

The General Options Library Configuration tab is used to select the low level library and CMSIS options (see Figure 6 Library Configuration). If it is desired to have target system `printf` output redirected to the debugger Terminal I/O window, the "Semihosted" and "Via semihosting" options must be selected. The "Use CMSIS" option must be selected to link in the standard Cortex M3 core access run-time libraries which are used by the software package.

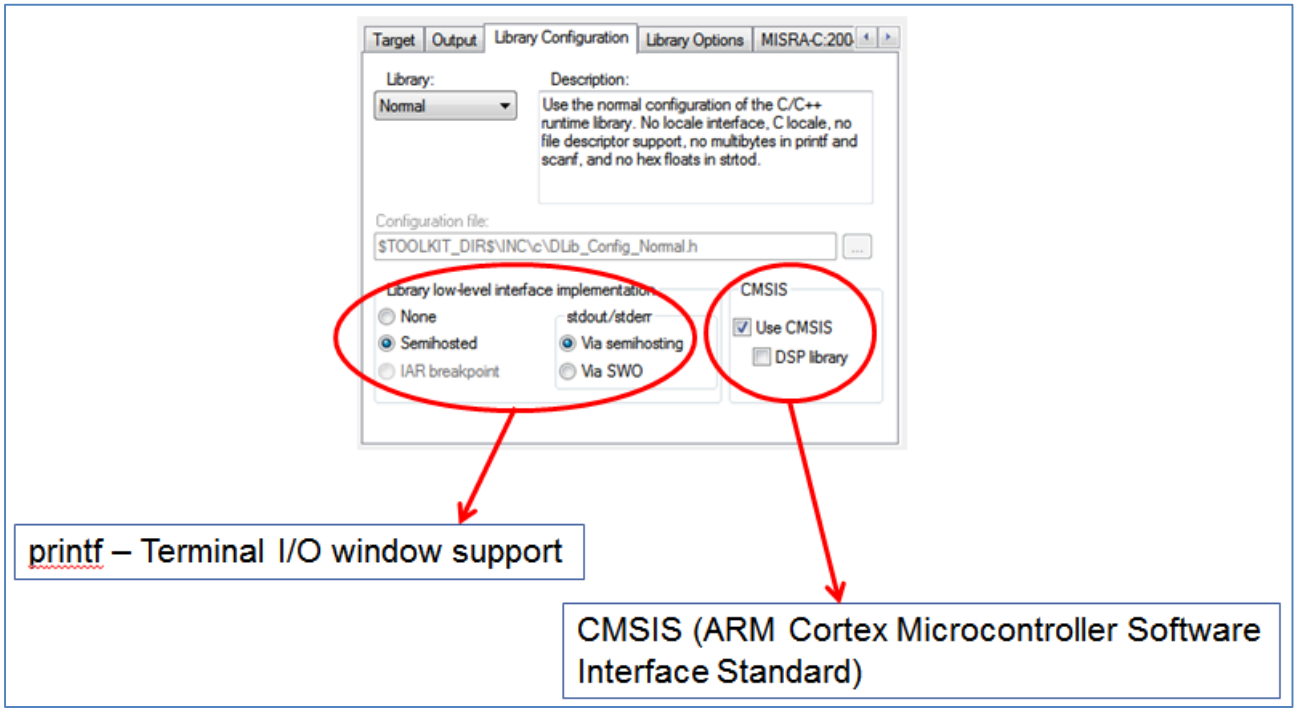


Figure 6 Library Configuration

3.2.3 C/C++ Compiler – Preprocessor

Depending on an application's layout, additional include paths may need to be specified.

Note that the IAR-provided `$PROJ_DIR$` macro is used in all the example project files (*.ewp) to make relative references within the project directory so that projects can be relocatable. This and various other predefined macros for referencing build-time paths, directories, temporary files, output files, etc., are provided by IAR (see *IAR Project Management and Building Guide for ARM*[III.b]).

The C/C++ Compiler Preprocessor tab is used to define these additional include paths. All ADuCM350BBCZ Software Package application projects must always add the following search paths, as a minimum:

```
<ADuCM350BBCZ_root>\inc
<ADuCM350BBCZ_root>\inc\config
```

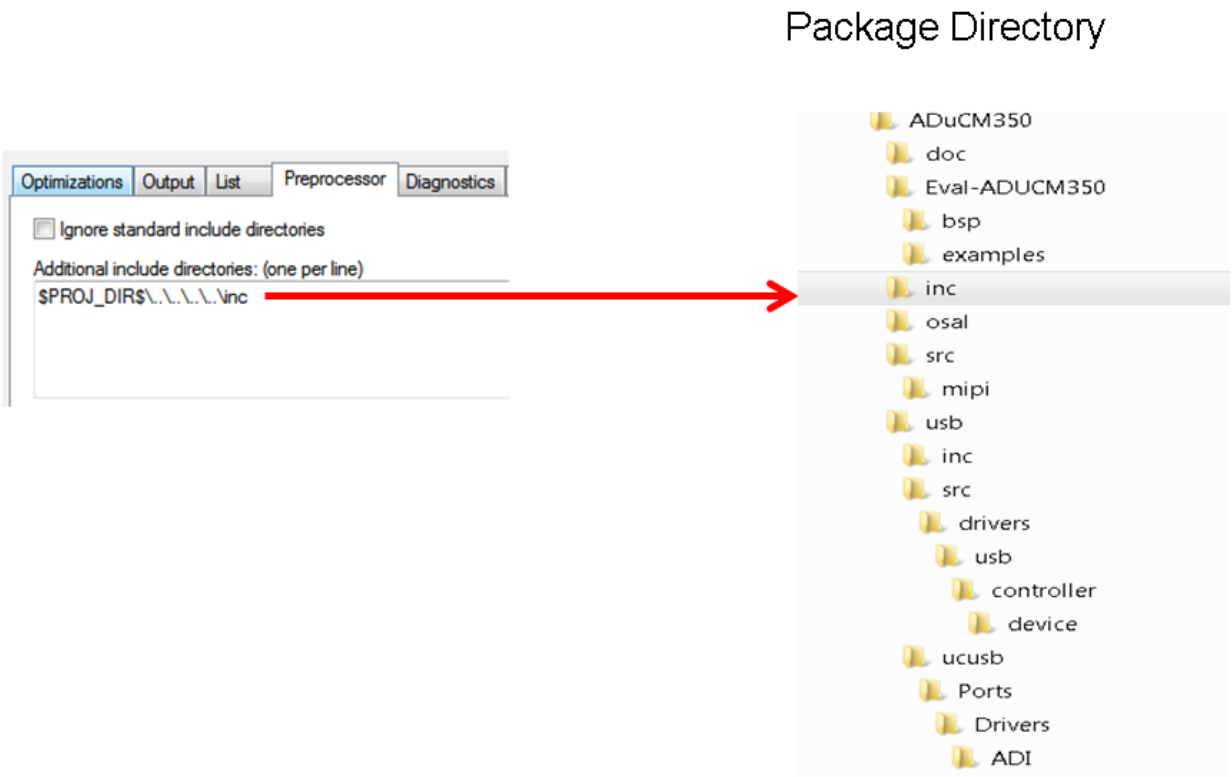


Figure 7 IAR Additional Include Paths

3.2.4 Debugger - Setup and Download

The debugger Setup tab is used to select the desired target and debugger. The examples included with this software package are pre-configured to use the Segger J-Link emulator. The Driver selection is set to J-Link/J-Trace. The debugger Download tab is used to enable the default flash loader (see Figure 8 Debugger Setup and Downloader) for burning executables into target flash for debug.

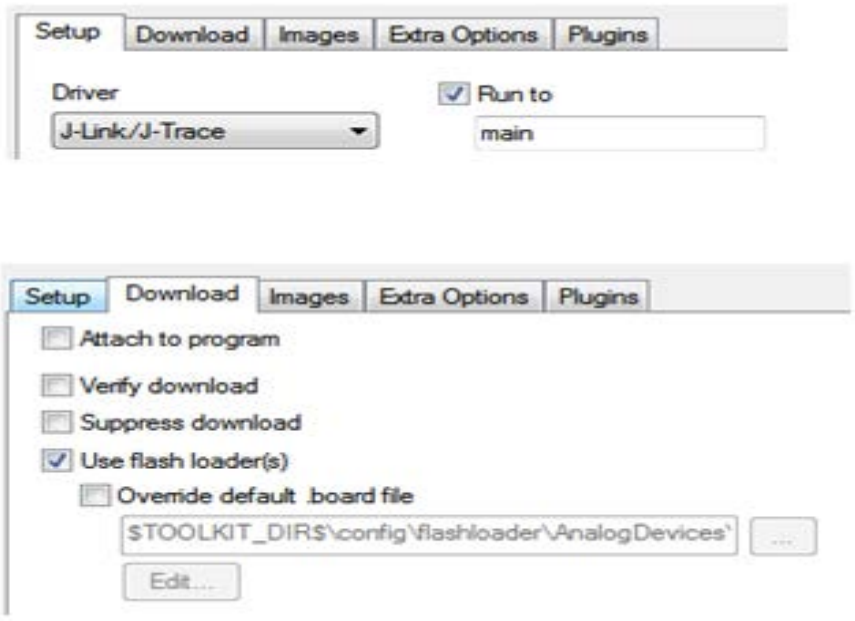


Figure 8 Debugger Setup and Downloader

3.2.5 J-Link/J-Trace - Setup and Connection

All projects (shipped with the release and newly-created, alike) must use the “Halt after bootloader” reset strategy in order to use the emulator to download and debug programs on target hardware. The reset strategy is managed in the project options dialog. Selecting the correct reset strategy is toolchain-specific.

To set/verify the reset strategy on the IAR toolchain, right-click on the top-most project element in the project’s “Workspace” tree-view window for each configuration (typically, “Debug” and “Release”) and select pop-up menu item “Options…” (alternatively, ALT+F7). Then browse to the sub-dialog for Debugger->J-Link/J-Trace” and select the “Setup” tab. Under the “Reset” drop-down listbox, select the “Halt after bootloader” option (see Figure 9 J-Link Setup and Connection).

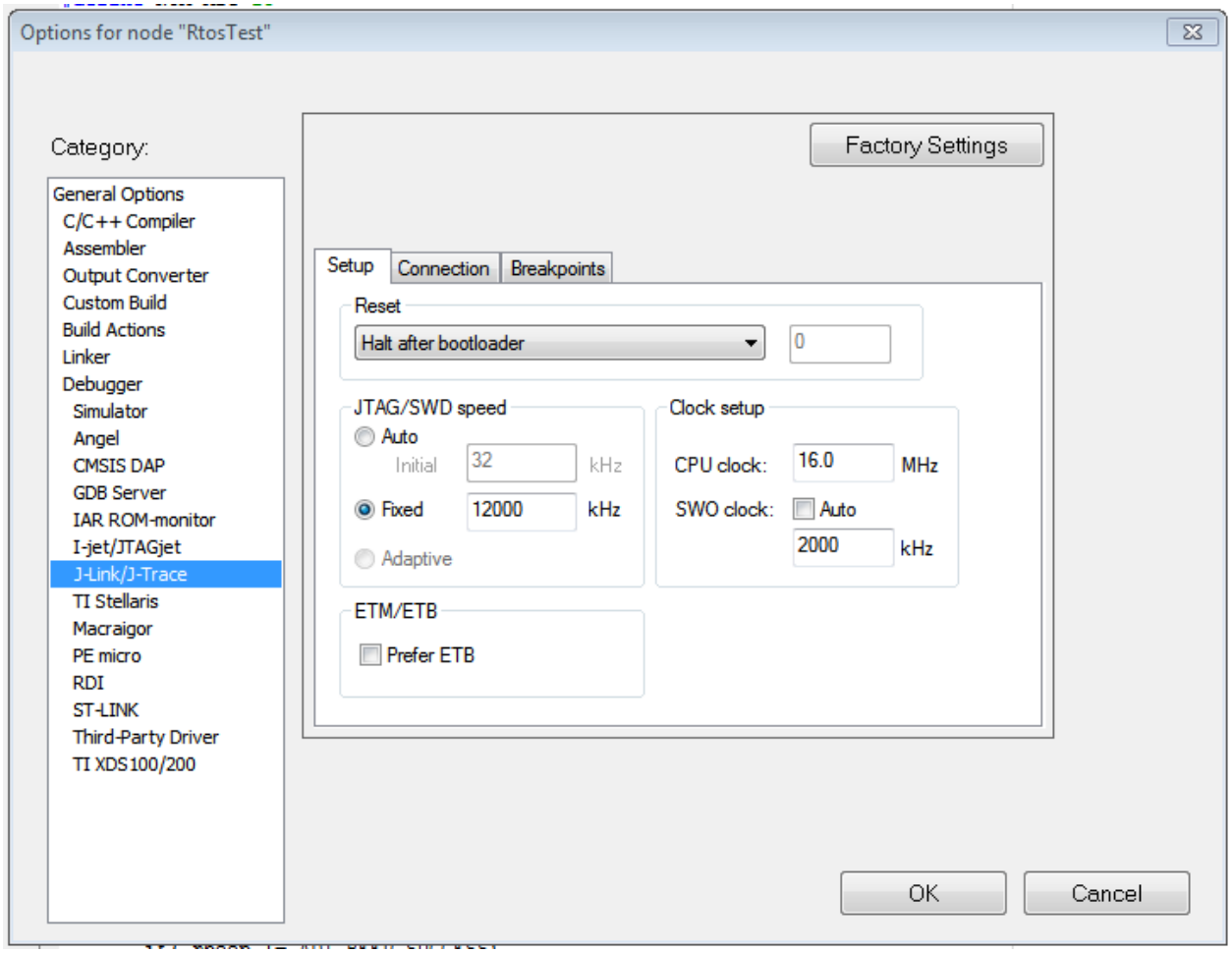


Figure 9 J-Link Setup and Connection

4 ADuCM350BBCZ 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 ADuCM350BBCZ on-chip peripherals. Figure 10 Peripherals and Driver Source illustrates the available peripherals and interconnects on the ADuCM350BBCZ processor and corresponding source files in the `<ADuCM350BBCZ_root>\src` directory.

In general, the driver sources are located in the `<ADuCM350BBCZ_root>\src` directory. The USB controller driver is located in the `<ADuCM350BBCZ_root>\usb` directory. When creating a project you must include the source files directly into the project. The include file path, `<ADuCM350BBCZ_root>\inc`, must also be specified in the project's compiler/preprocessor options (see section 3.2.3 C/C++ Compiler – Preprocessor).

In addition to the device drivers, services and USB driver, the software package also includes the OSAL (Operating System Abstraction Layer) package, which provides a wrapper around the optional RTOS (if used), isolating user code from the underlying RTOS vendor/implementation.

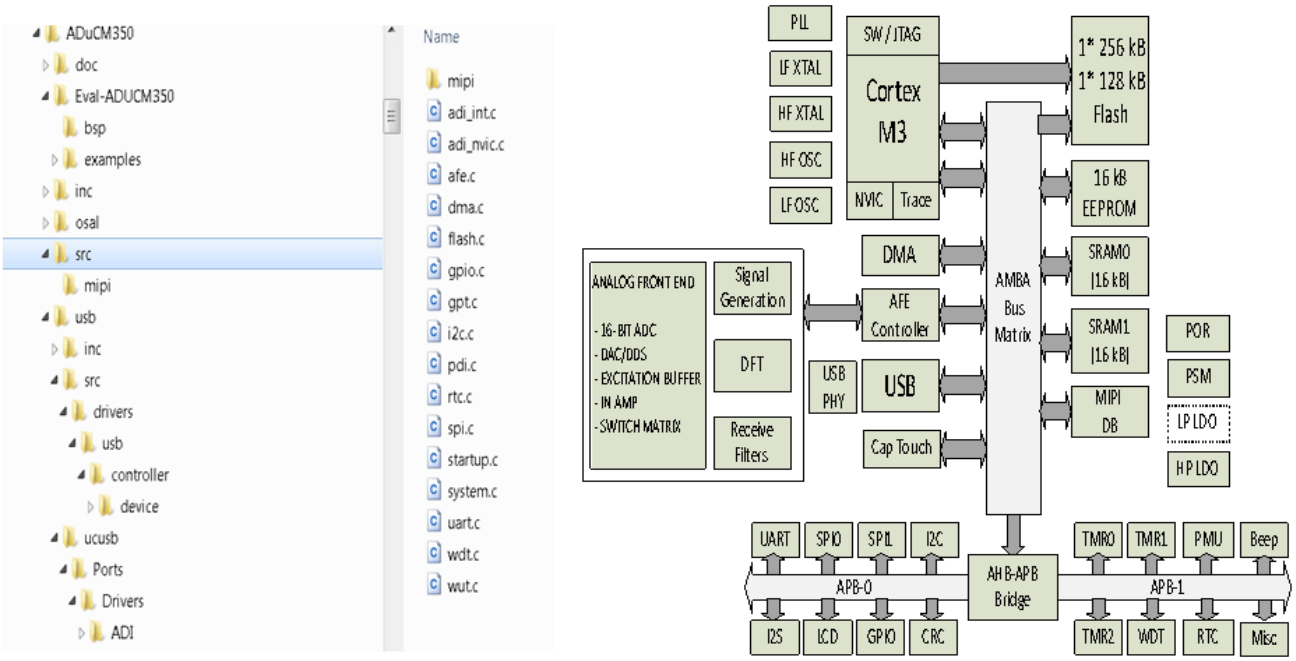


Figure 10 Peripherals and Driver Source

4.2 Boot-Time CRC Validation

The ADuCM350BBCZ 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, validating flash integrity, configuring/monitoring the UART controller for serial downloader commands, etc. Please refer to the **Application Notes** in the `<ADuCM350BBCZ_root>\doc\AppNotes` directory for details on these and related topics.

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 CRC test is bypassed if an all-1's (bulk erase default) CRC signature is detected, but any non-all-1's CRC signature value is interpreted by the kernel as a valid CRC Flash0/Page0 code to be validated.

The Page0 embedded CRC signature is stored at reserved location 0x000007FC (see default linker control file, `<EWARM_root>\arm\config\linker\AnalogDevices\ADuCM350BBCZ.icf`). The build-time CRC signature is computed and implanted into the executable during the target build process (as a post-link build command). At boot time, the kernel computes a run-time Flash0/Page0 CRC value using the on-chip CRC hardware and compares it against the build-time version implanted within the executable. If the two CRC values match, the normal boot sequence proceeds.

On a CRC failure, the chip enters a secure mode and will *only* respond to a serial downloader bulk-erase command as a means to protect against granting control to a suspect code image while also allowing recovery of failing chips by downloading a fresh (and presumably intact) code image. See `<ADuCM350BBCZ_root>\tools\SerialDownloader` directory, as well as related documentation in `<ADuCM350BBCZ_root>\doc\AppNotes`.

To generate the required forward CRC signature, the IAR linker "Checksum" dialogue box is used to configure the IDE to generate and implant the CRC into the executable using the IAR ELF file utility tool, "iElfTool", as a post-link build command (see Figure 11 Configuring Pre-Boot Checksum). A modified version of the iElfTool may also be used to generate reverse CRC signatures, CRC signatures for any flash page(s), and flash parity encodings as well.

Additionally, the extra option "—keep __checksum" needs to be passed to the linker. This is also shown in Figure 11.

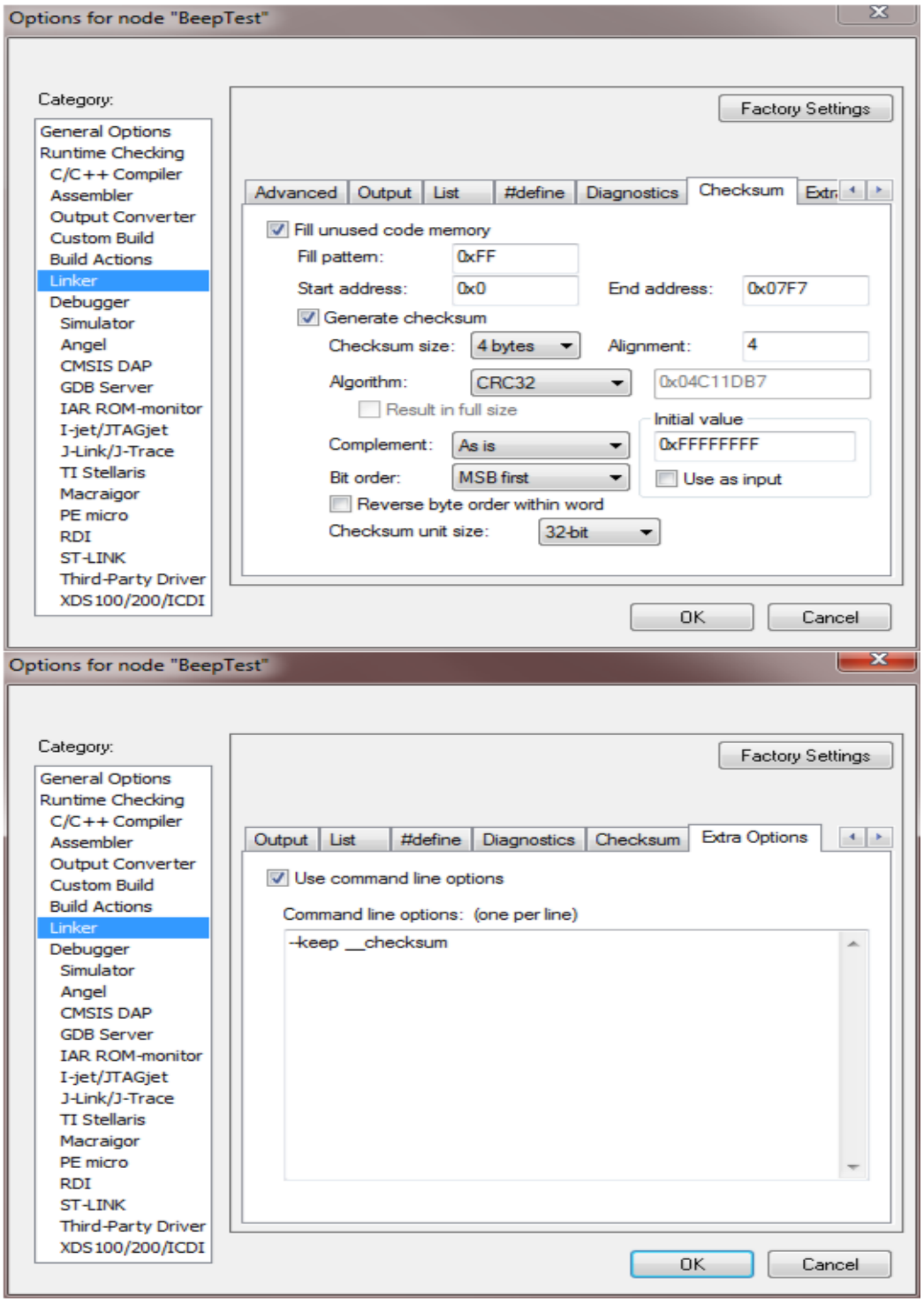


Figure 11 Configuring Pre-Boot Checksum

Note: All the examples delivered with the ADuCM350BBCZ Software Package implement the CRC generation steps to brand each executable with correct CRC codes for boot-time CRC validation.

4.3 Run-Time Parity Validation

The ADuCM350BBCZ processor includes an option to perform runtime test parity testing. Please consult the HRM (Hardware Reference Manual) for details of the flash parity feature.

Parity for application images may be generated and implanted into the executable at build-time. At system run-time, real-time hardware parity checking may be enabled. If enabled, the parity hardware compares the run-time generated parity of executing code against the build-time parity stored in the code image. On parity faults, a parity error interrupt is issued which should be configured (by the user code) to have the highest priority possible.

Please refer to the detailed `ParityTest` example in the ADuCM350BBCZ Software Package for a detailed treatment of generating and comparing parity on the ADuCM350BBCZ processor.

Note: All the examples delivered with the ADuCM350BBCZ Software Package implement the parity generation steps to brand each executable with correct parity codes for run-time parity validation.

4.4 Generating CRC and Parity Info

Generating CRC and parity information is done using a modified version of the IAR ELF Utility tool: iElfTool. The modified iElfTool executable, modified sources and build project are provided in the ADuCM350BBCZ Software Package under the `<ADuCM350BBCZ_root>\tools\IELFTool` directory. The modifications include supporting the ability to generate *reverse CRC* encoding (in addition to forward CRC encoding, already present in the native iElfTool and the IAR project GUI), and to generate flash *parity* (not present in the native iElfTool).

The modified iElfTool is run as a post-build step after the linker has fully resolved the executable.

Special symbols are used to convey both CRC signatures and parity information. These symbols must be preserved and protected against linker elimination.

The following steps are required to generate and embed parity information.

4.4.1 Specify Linker KEEP Directives

Add the following lines to the project: Options->Linker->Input->Keep symbols:

__checksum0
__checksum1
__parity0
__parity1

4.4.2 Specify Linker Command Line Settings

Add the following lines to the project: Options->Linker->Extra options->Command line options:

```
--keep __checksum
--place_holder __checksum0,4,.checksum0,4
--place_holder __checksum1,4,.checksum1,4
--place_holder __parity0,0x1F00,.parity0,4
--place_holder __parity1,0x0F80,.parity1,4
```

Note that the first line (--keep __checksum) is used for Page0 checksum and needs to be present regardless of the overall CRC/parity features of the flash.

4.4.3 Specify iElfTool Post-Build Settings

Add the following to the project: Options->Build Actions->Post-build command line. The individual commands are shown here on separate lines for clarity, but the entire command should be entered as a single line:

```
<ADuCM350BBCZ_root>\tools\IElfTool\Debug_Win32_9\ielftool.exe
 --fill 0xFF;0x0-0x5ffff
 --parity __parity0:0x1F00,even:0x00000000:L;0x0-0x3dfff
 --parity __parity1:0x0F80,even:0x00040000:L;0x40000-0x5efff
 --checksum __checksum0+0:4,crc32:LiR,0xffffffff;0x0-0x3fff7
 --checksum __checksum0+4:4,crc32:Li,0xffffffff;0x0-0x3fff7
 --checksum __checksum1+0:4,crc32:LiR,0xffffffff;0x40000-0x5fff7
 --checksum __checksum1+4:4,crc32:Li,0xffffffff;0x40000-0x5fff7
 $TARGET_PATH$
 $TARGET_PATH$
```

Where,

<ADuCM350BBCZ_root> is the ADuCM350BBCZ Software Package install location. This can also be `$PROJ_DIR$` provided that the ielftool.exe executable has been copied to the project directory. Alternatively, an absolute path for the location of the executable can be used, or the executable can be placed in a directory which is in `$PATH` and then referred to by name alone.

$TARGET_PATH$ is a pre-defined IAR project file macro that names the produced executable file, which is usually also the name of the project, i.e. `ParityTest` in this example. Note that `$TARGET_PATH$` is named twice in this example; once as the input file and again as the output file.

4.4.4 Generic iElfTool Command Line Description

The native arguments to the iElfTool command are described in the IAR C/C++ Development Guide, and also in the online help, apart from the checksum '**R**' option and the "—**parity**" command, which are new and are described here.

The 'R' option simply instructs the --checksum command to traverse the specified memory region in high-to-low address order. This is used above to compute the reverse signature for each flash memory region.

The description of the --parity command is as follows:

Syntax:
--parity {symbol[+offset]|address}:size,algorithm:base[:flags];range[;range...]

Parameters:

- **symbol**
  The name of the symbol where the parity array should be stored. Note that it must exist in the symbol table in the input ELF file.

- **offset**
  An offset to the symbol.

- **address**
  The absolute address where the parity array should be stored.

- **size**
  The number of bytes in the parity array; must be a multiple of 4 and must not be larger than the size of the parity symbol.

- **algorithm**
  The parity algorithm used, one of:
  - **even**, the parity bit is zero if, and only if, the number of 1s in the source word is even
  - **odd**, the parity bit is zero if, and only if, the number of 1s in the source word is odd

- **flags**
  The parity word size size, one of:
  - **L** calculates one parity bit per 32-bit word
  - **W** calculates one parity bit per 16-bit halfword
  - **B** calculates one parity bit per 8-bit byte
  - **r r**everses the order of bytes or halfwords within a 32-bit word parity calculation. Only applicable with the W and B flags.

- **base**

  The memory address of the start of the flash region for which the parity is being calculated.

- **range**

  An address range on which the parity should be calculated. Hexadecimal and decimal notation is allowed (for example, 0x8002–0x8FFF). The range does not have to begin at the start of the flash region.

4.5 System Reset Strategy

All projects require the “Halt after bootloader” reset strategy 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 tool chain and emulator specific (see Figure 9 J-Link Setup and Connection).

To set/verify the reset strategy on the IAR tool chain, right-click on the top-most project element in the project’s “Workspace” tree-view window for each configuration (typically, “Debug” and “Release”) and select pop-up menu item “Options…” (alternatively, ALT+F7). Then browse to the sub-dialog for Debugger->J-Link/J-Trace” and select the “Setup” tab. Under the “Reset” drop-down listbox, select the “Halt after bootloader” option.

5 Build Configurations

5.1 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. Within 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.

5.1.1 Application Initialization

The functions `SystemInit()` and `adi_initpinmux()` are used to initialize an application. `SystemInit()` is required to initialize the ARM Cortex CMSIS infrastructure. `adi_initpinmux()` initializes the peripheral pin multiplexing if static pin multiplexing is used (see section 5.1.2 Static Pin Multiplexing). Figure 12 - Application Initialization, shows these functions being called from the user application `main()`.

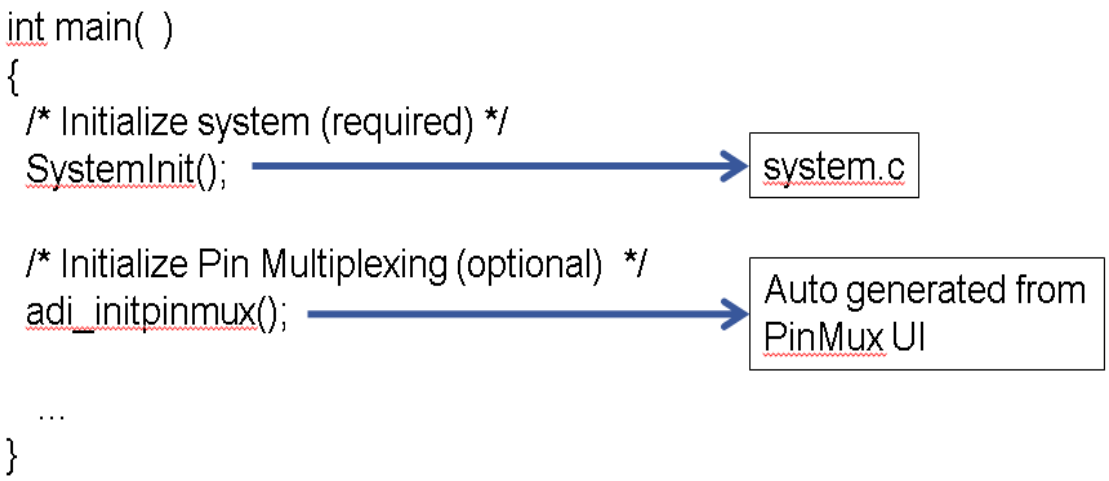


Figure 12 - Application Initialization

5.1.2 Static Pin Multiplexing

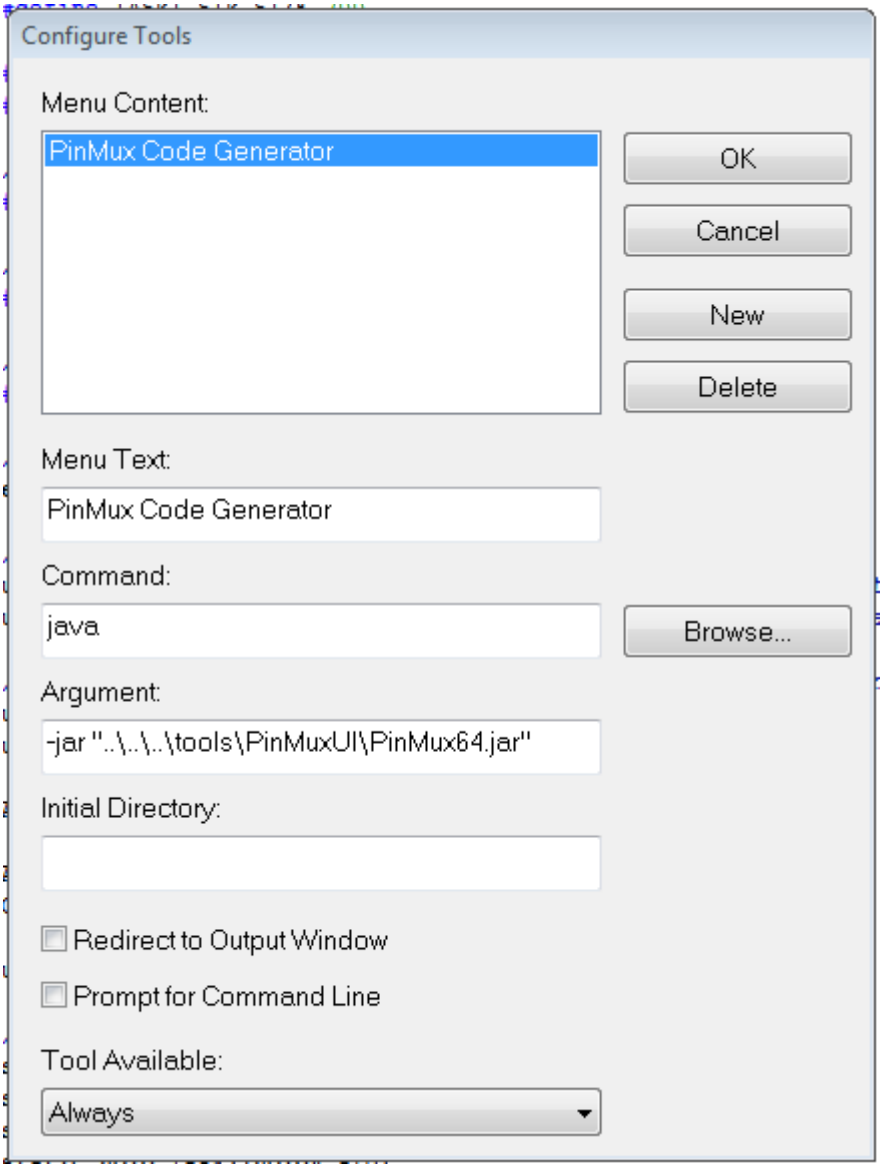
Included with the ADuCM350BBCZ Software Package is a Pin Multiplexing application which is capable of generating code to set all the port MUX and FER registers statically for all peripherals in a single call. The Pin Multiplexing application is a Java application which is run from a command prompt:

Location: `<ADuCM350BBCZ_root> \tools\PinMuxUI`

- 32-bit version: `java -jar PinMux32.jar`
- 64-bit version: `java -jar PinMux64.jar`

You will need to find the location of your java.exe executable on your machine. For example, version 6 installs into `C:\Program Files (x86)\Java\jre6\bin` by default.

The PinMux of choice may be added to the IAR “Tools” menu by adding the following information (adjusted for your install path and version) to the “Configure Tools” dialogue, thus:



This way, the PinMux GUI may be started from within the IAR GUI.

After starting the PinMux application you must first select the correct processor type (top-right drop-down listbox). You then 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 file which sets the GPIO port configuration registers based on the peripherals and functions selected. This C file should be manually added to your project. This C file has a function `adi_initpinmux()` which can be called from the application source (see Figure 13 - Pin Multiplexing Application).

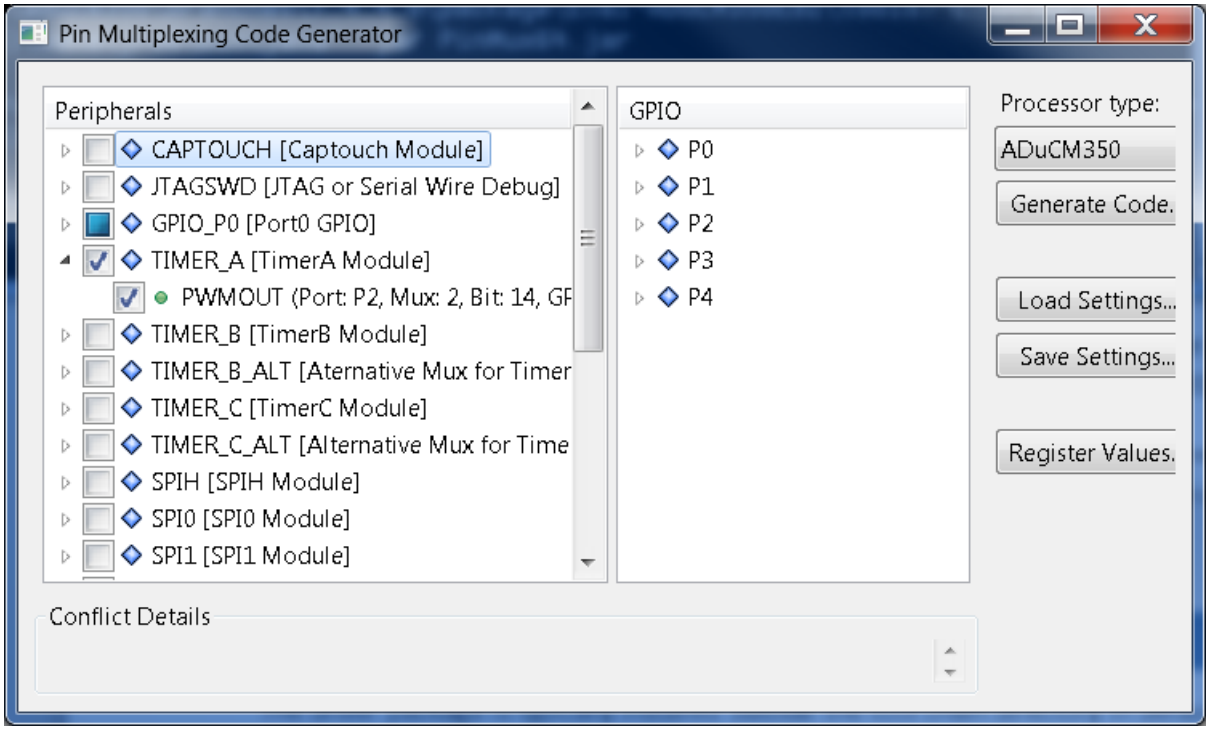


Figure 13 - Pin Multiplexing Application

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

5.1.3 Dynamic Pin Multiplexing

Dynamic pin multiplexing capability has been deprecated for the last few releases and has been entirely removed with this release. Please use the GUI pin multiplexing methodology as explained in section 5.1.2 Static Pin Multiplexing.

5.1.4 Driver Include Files

The C/C++ Compiler Preprocessor tab is used to define the additional include directories needed to build the project. If the application isn't using an RTOS, the device drivers only require the

```
<ADuCM350BBCZ_root>\inc
<ADuCM350BBCZ_root>\inc\config
```

directories from the software package installation. Applications may need to augment the preprocessor search path with their own requirements.

5.1.5 uC/OS-II Include Files

If an application is using the Micrium uC/OS-II RTOS, additional include directories from the Micrium software installation must be added to the preprocessor additional include paths project options dialog (see Figure 14).

To identify the Micrium root directory (which is disjoint from the ADuCM350BBCZ Software Package), the `MICRIUM_DIR` system environment variable is defined to point to the Micrium software installation folder. This environment variable is then added to the additional include directory dialog of a project using the syntax `$_MICRIUM_DIR_$` (the leading "$_" and trailing "_$" are name decoration IAR defines to access system environment variables within a project).

Complete details of the Micrium source interface are provided in both the "Examples" section of this document, and in the USB examples delivered with the ADuCM350BBCZ Software Package.

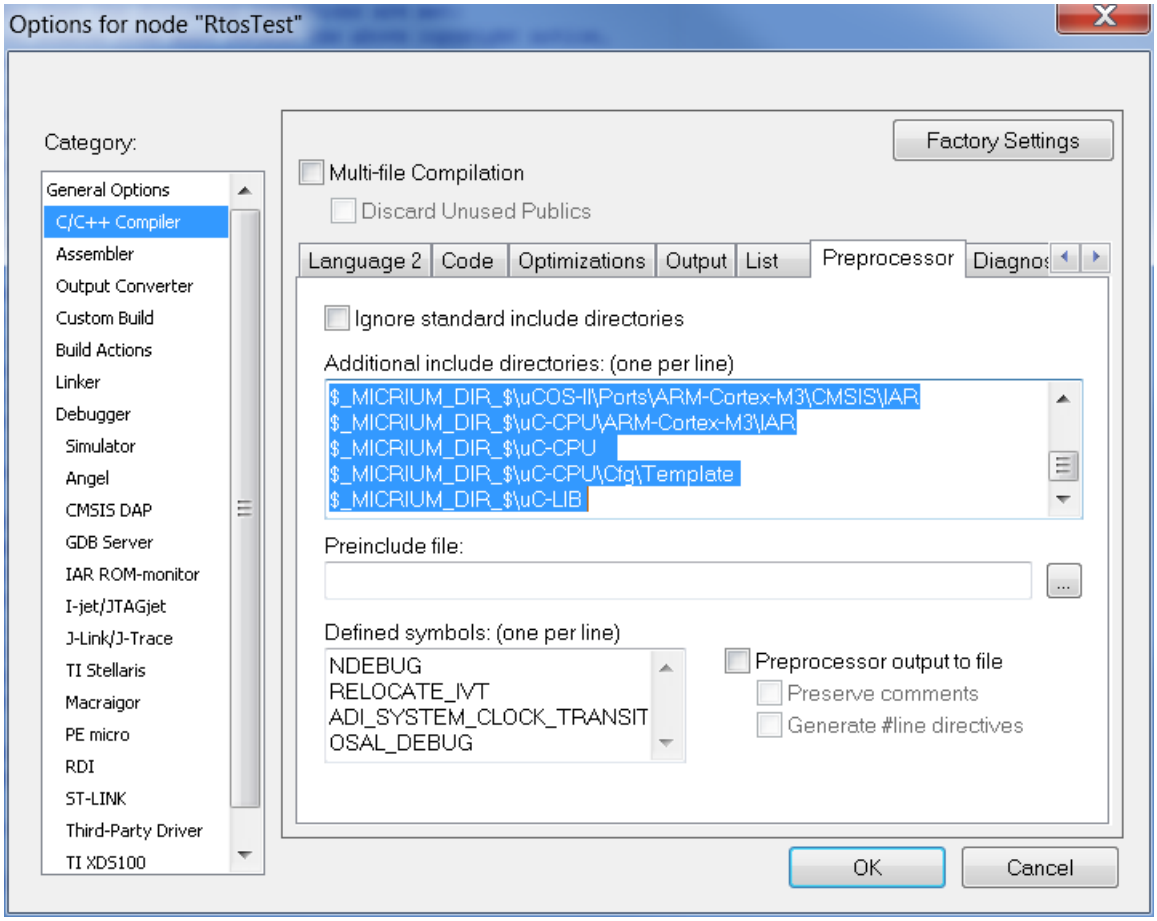


Figure 14 – IAR: C/C++ Compiler Includes for uC/OS-II RTOS

The full list of include paths needed is:

```
$PROJ_DIR$\..\..\..\osal\uCOS-II\Source
$PROJ_DIR$\..\..\..\osal\uCOS-II\Ports
$_MICRIUM_DIR_$\uCOS-II\Source
$_MICRIUM_DIR_$\uCOS-II\Ports\ARM-Cortex-M3\CMSIS\IAR
$_MICRIUM_DIR_$\uC-CPU\ARM-Cortex-M3\IAR
$_MICRIUM_DIR_$\uC-CPU
$_MICRIUM_DIR_$\uC-CPU\Cfg\Template
$_MICRIUM_DIR_$\uC-LIB
```

Please note that the OSAL sources shown here are relative to a specific project's location.

5.2 Driver Configuration

Most of the drivers are statically configurable. Configuration is controlled via C preprocessor 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.

5.2.1 Global Configuration

There is a single, global configuration file, `<ADuCM350BBCZ_root>\inc\config\adi_global_config.h`. To override the global feature set ADI recommends the same approach for overriding the driver-specific configuration files, as described above. Content of the adi_global_config.h is shown below in Figure 15:

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

/*! Set this macro to 1 to enable soon-to-be-DEPRECATED dynamic pin muxing by the drivers */
#define ADI_CFG_ENABLE_DEPRECATED_DYNAMIC_PIN_MUX_SUPPORT 1
```

Figure 15 Global Configuration File Contents

5.2.2 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 `<ADuCM350BBCZ_root>\inc\config` directory, which may be edited for global changes or overridden for localized changes.

It is recommended that the default configuration files are not modified, but overridden by copying to the local project directory, and modified as needed.

5.2.3 Configuration Overrides

The default factory configuration files can be edited directly for global changes (for all applications). Individual overrides can be made by copying/modifying 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, are three ways to override the default static configuration values:

1. Globally: Modify the default factory default files for global changes.
2. Locally: Copy the driver's default configuration file into the application's source folder. Local edits can then be applied. In so doing, you must ensure the application's source folder appears before the default `inc\config` folder in the project's preprocessor include path option settings.

   Note: Local overrides (if any) are the recommended override method.
3. Dynamically: Use the dynamic APIs to modify the configuration at run time. The configuration APIs may be called at runtime 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.

5.2.4 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 preprocessor macro **`RELOCATE_IVT`** is used enable IVT relocation.

The default, statically-linked IVT placement and content 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.

For dynamic IVT content, the IVT may be relocated (during system startup) from ROM to RAM. Applications may need to modify the IVT content (such as in dynamically hooking/replacing interrupt handlers or running an RTOS that requires patching interrupt handlers through a common interrupt dispatcher). To support dynamic IVT relocation, assert the **`RELOCATE_IVT`** macro in the compiler preprocessor option tab. This causes the IVT to be relocated during system reset (see details within `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 code in system files `startup.c` and `system.c` (bracketed by 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 may be modified dynamically.

5.2.5 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, 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, describing what caused the interrupt. Some device drivers support event filtering, allowing 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 const 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.

5.3 RTOS Configuration

This release provides a Real Time Operation System (RTOS) Operating System Abstraction Layer (OSAL) that provides a wrapper separating the application from the underlying RTOS implementation. Currently, the RTOS implementation is limited to Micrium's uCOS-II preemptive, real-time multitasking kernel, which is licensed software available separately from Micrium; ADI does not provide the RTOS itself, only the OSAL wrapper.

Each application that uses the Micrium uC/OS-II RTOS requires the following additions to the project file:

1. RTOS configuration files: These files contain application configuration data and functions. Refer to the *Micrium uC/OS–II User's Manual*[V,b] for information regarding these configuration files. Some of the RTOS examples within this release illustrate these files.
2. OSAL sources or library: The operating system abstraction layer (OSAL) API provides a common API for a small subset of RTOS functionality. The sources may be used directly, or the OSAL library may be added to the linker additional libraries options dialog.

The RTOS source files can be added (as needed) to the application project file as shown in Figure 16 - RTOS Project Files.

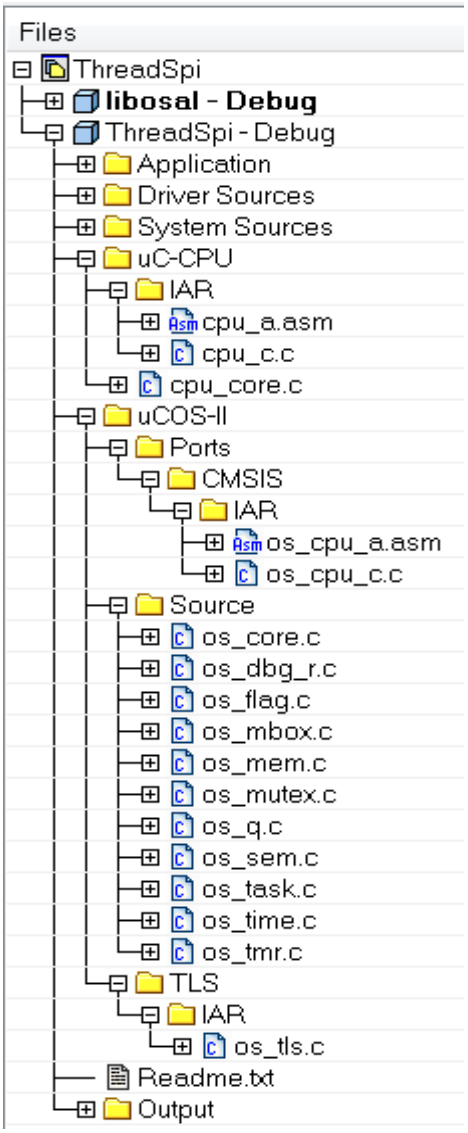


Figure 16 - RTOS Project Files

All the device drivers are setup to cooperate within an RTOS, but some special steps are required to properly activate RTOS-awareness within the drivers. The drivers do not plug directly into any one

underlying RTOS, but use an abstraction layer to shield the application space and drivers from directly playing into any one RTOS implementation.

5.3.1 RTOS Aware Drivers

The macro **`ADI_CFG_ENABLE_RTOS_SUPPORT`** controls whether or not drivers are compiled in an RTOS-aware manner. By default drivers are configured to be RTOS-Unaware. The **`ADI_CFG_ENABLE_RTOS_SUPPORT`** macro is managed in the global configuration file, described above.

5.3.2 RTOS Aware Drivers and the need for OSAL support

When compiled in an RTOS-aware manner, an application will need to either include the source code for the Operating System Abstraction Layer (OSAL) in the project source set, or add an OSAL library ("Debug" or "Release", e.g., `<ADuCM350BBCZ_root>\osal\uCOS-II\Lib\iar\Debug\Exe\lobosal.a`) to the application's project linker "Additional Libraries…" options dialog.

If using the OSAL library approach, the library must be either prebuilt, or the OSAL library project (e.g., `<ADuCM350BBCZ_root>\osal\uCOS-II\Lib\iar\libosal.ewp`) must be included (and built) in the application's project space. Either way, the OSAL library build is a prerequisite to the applications final link.

5.3.3 RTOS Aware Driver and IVT Table Location

When drivers are configured to be RTOS aware, the IVT table must be relocated to RAM as explained in section 0. This allows the interrupt handlers to be intercepted (through redirecting the IVT entries) by the RTOS common interrupt dispatcher for context switching tasks before and after calling the underlying (default) handler that was replaced. OSAL interrupt registration accomplishes this task.

6 Examples

The ADuCM350BBCZ Software Package contains numerous example and test programs illustrating use of the ADuCM350BBCZ 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 debugger "Terminal I/O" console display (for most examples) or a PC based terminal application session (or equivalent console app for hardware-based UART output) to capture example output.

The examples are located in the `<ADuCM350BBCZ_root>\examples` directory. 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 `<ADuCM350BBCZ_root>` directory. The evaluation board is the only target to which the examples are ported.

6.1 Build and Run

Each example may be launched into the toolchain IDE by launching/opening the associated example project (for IAR, this is the "Embedded Workbench Workspace" file, *.eww). Each project contains an "iar" sub-directory within which the toolchain-specific project build files reside.

Locate the ".\iar" directory (under any example directory) and launch the appropriate project file by double-clicking it (or loading it through the IDE project file browser). Build the project and launch the debugger to download the executable to the target.

The example project release-mode builds are configured for linker elimination and aggressive optimizations set to prefer minimal code size to speed.

6.2 Self-Test Debug Code

The device driver sources contain special self-test code in the debug build mode that performs internal state validation and test various assertions at run-time. The debug code is enabled with the `ADI_DEBUG` macro. The `ADI_DEBUG` macro is set by the preprocessor predefines for the "Debug" build configuration, and *not* set for release mode builds. It is recommended to enable this macro during application development and debug to help catch run-time driver and application errors early. The extra debug code increases the executable image size as well as execution time, so be sure to undefine this macro for release mode builds.

The rule-of-thumb is that release-mode builds exclude all debug code, such that only essential code for correct and successful operation of the driver is included.

6.3 Capturing STDIO from Examples

Most of the example projects provided in this release are pre-configured to report status to STDIO, which is redirected to the host PC debugger.

To use this facility in the IAR Embedded Workshop: enable the "Semihosted" low-level interface (in the runtime library) and enable steering of STDIO "Via semihosting". Both these settings are located under the project's "General Options->Library Configuration" options dialog. To view the captured output, open the debugger's "View->Terminal I/O" display. Other development environments may have similar configuration and viewing facilities.

Semi-hosting is slow because it adds extra low-level runtime library code to the linked target image in order to redirect STDIO from the core to the host PC debug console. This is accomplished with a combination of the Cortex M3 core Debug Monitor Exception (exception #12) and the added runtime library target code that talks to the Host PC debugger via the debugger interface (SWD/JTAG). *Note: semi-hosted output speed by be greatly increased under IAR by enabling "Buffered terminal output" in the "General Options->Library Options" project options dialog.*

WARNING: Semi-hosted projects MUST have an attached debugger to process the debug exception, allowing it to route the redirected STDIO to the debug interface. If an active emulator session is not present to handle the debug monitor exception, the core will generate a hard fault. This means that production code intended for release in "turn-key" systems (no emulator attached),

MUST be built with semi-hosting disabled! Never expect semi-hosted code to run properly without an active debugger attached.

6.4 RTOS Builds

The Analog Devices "ThreadTest" (and other) examples demonstrate direct use of the Micrium uC/OS-II RTOS and thread creation (without the OSAL). Any example that employs the Micrium RTOS relies on a Windows System environment variable called `MICRIUM_DIR` to point to the Micrium install location (see example setting, Figure 17 - Micrium Environment Variable Windows dialogue). This is an Analog Devices convention to facilitate linking against the Micrium components and users must manually create/edit the `MICRIUM_DIR` environment variable to point to the Micrium install directory on *their* computer for these examples to build successfully.

IAR project files (*.ewp) reference the Micrium install directory location through this environment variable, using a decorated version of it as "`$_MICRIUM_DIR_$`" (note the leading and trailing underscore name decorating which is the IAR method of accessing user-defined system environment variables within the IAR project space) within the project files (*.ewp). This is similar to the use of IAR-defined `$PROJ_DIR$` and `$TOOLKIT_DIR$` macros (and others) that reference the project root directory and tool chain install directory, respectively.

See the *IAR Embedded Workbench IDE Project Management and Build Guide*[III.b] for details on the IAR-defined project macros.

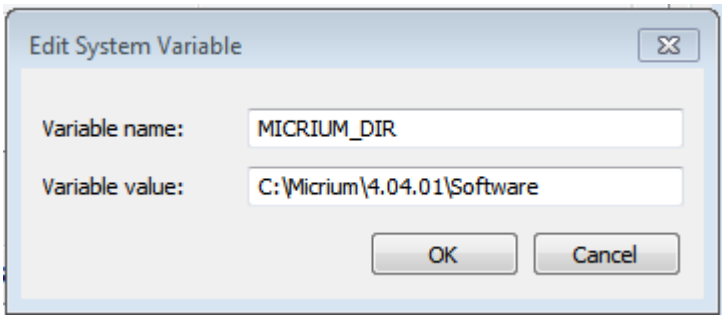


Figure 17 - Micrium Environment Variable

7 USB Software

ADuCM350BBCZ Software Package includes USB peripheral device driver and board support functionality that enable Full Speed USB Device operation on the Eval-ADUCM350EBZ board. The USB drivers and examples work in conjunction with the Micrium μC/OS-II and μC/USB-Device and device class driver products, obtained and licensed from Micrium.

7.1 Software Requirements for USB

USB Functionality on ADuCM350BBCZ evaluation board (Eval-ADUCM350EBZ) requires the following software packages:

- ADI's ADuCM350BBCZ Software Package
- ADI's OSAL library (optional)
- Micrium's uC/OS-II (see release notes for version)
- Micrium's uC/USB-Device (see release notes for version)
- Micrium's uC/USB-Device-CDC (see release notes for version)
- Micrium's uC/USB-Device-PHDC (see release notes for version)
- Micrium's uC/USB-Device-MSC (see release notes for version)

Note: Applications based on other class driver must get the class drivers from Micrium.

7.2 USB Examples

Three examples are included with the ADuCM350BBCZ Software Package which demonstrates USB communications between the ADuCM350EBZ evaluation board and a Windows host.

In all three examples, it is best to run the terminal emulator with a clean target start. If either the USB cable is reinserted or the target software is restarted, the terminal emulator generally requires restarting as well.

- **Personal Healthcare Device Class (PHDC)**

  The `<ADuCM350BBCZ_root>\examples\uCOS-II-USBD-PHDC` example has two modes of operation, designed to demonstrate the different Quality of Service (QoS) levels defined in the PHDC specification.

  - The first, or *basic*, mode of operation implements a single *High Latency/Reliable* QoS level. The basic mode of operation illustrates a USB use case in which the device transfers either the firmware details or a set of pseudo temperature values on demand from the host application (also provided with the example). The example represents the scenario of connecting a healthcare device to a PC to download its logged content.
  - The second, or *multiple*, mode of operation implements a *Low Latency/Good* QoS level. The multiple mode of operation illustrates a USB use case in which the device

transfers the CPU percentage usage statistics to the host application at 100ms intervals. Additional QoS levels can be added to transmit sample data at intervals. This example represents the scenario of a healthcare device continuously transmitting data to a host PC for monitoring purposes. The example is also the same as supplied by Micrium with the uC/USB-Device-PHDC product.

Both modes use the same WinUSB Windows device driver, for which the signed installation information file and co-installers are supplied. The required Windows device driver is supplied in the path `<ADuCM350BBCZ_root>\examples\uCOS-II-USBD-PHDC\host\driver`. The Readme.txt files present in the example project directories contain detailed information about each example. The directory location and structure of the examples are shown in Figure 18 - USB PHDC Components.

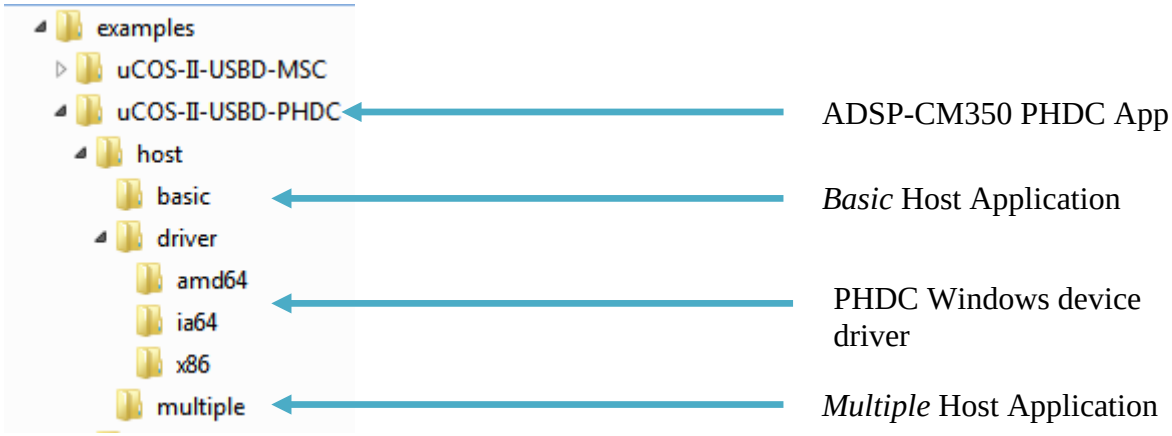


Figure 18 - USB PHDC Components

- **Mass Storage Device Class (MSC)**

  The `<ADuCM350BBCZ_root>\examples\uCOS-II-USBD-MSC` example exports a small device-based RAM disk to the USB MSC interface which can be accessed by a host PC that supports the USB MSC standard. As such there are no host device drivers or host applications required or provided; the OS-supplied file system is able to display the contents of the media, which is empty on reset.

  For example, upon device connection, a driver letter will appear under "Devices with Removable Storage" on Windows' Explorer. The usual rules apply for removable storage media and the RAM Disk can be ejected, removed and re-inserted to demonstrate persistence of data while the USB lead is unplugged.

  Please refer to the Readme.txt file present in the project directory of the example for further details. The directory location and structure of the example are shown in Figure 19 - USB MSC Components.

  Note: On restarting the application or power cycling the evaluation board, any files created on the RAM disk will be destroyed.

Figure 19 - USB MSC Components

- **Communication Class Device Class (CDC)**

  The `<ADuCM350BBCZ_root>\examples\uCOS-II-USBD-CDC` example demonstrates the Communication Device Class (CDC). It converts the USB device into a serial communication device, and the target is recognized by the host as a serial interface (USB2COM, virtual COM port). The Windows device driver is supplied in the path `<ADuCM350BBCZ_root>\examples\uCOS-II-USBD-CDC\host\driver`. The signed installation information file and co-installers are supplied.

  Upon device detection a new COM port named as ADI/Micrium USB Virtual COM Device (CDC/ACM) (COMX) will be seen in the device manager. A terminal application with this COM port can be opened and used to send and receive the data over USB Virtual COM Device.

  Please refer to the Readme.txt file present in the main project directory of the example for further details and operational notes. The directory location and structure of the example are shown in Figure 20 - USB CDC Components.

Figure 20 – USB CDC Components

Note: Before building any of the USB examples, users must install the required Micrium components and set the `MICRIUM_DIR` environment variable as specified in section 6.4 RTOS Builds.

8 Device Driver API Documentation

The complete documentation for the ADuCM350BBCZ Software Package 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 21 - Device Driver Documentation. The html documentation is located in the `<ADuCM350BBCZ_root>\doc\html` folder. To open the html documentation, double-click on the index.html file.

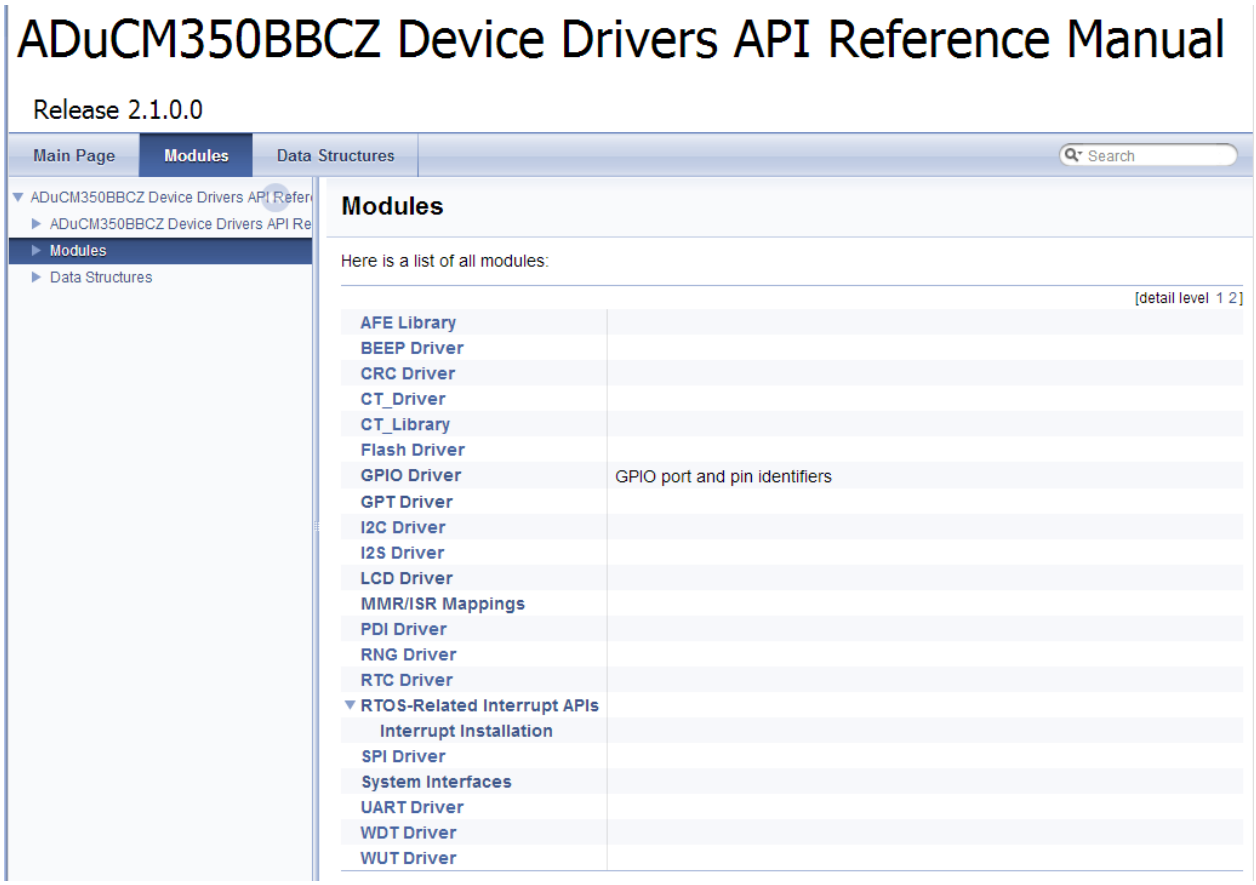


Figure 21 - Device Driver Documentation

9 Appendix

9.1 CMSIS

The ADuCM350BBCZ Software Package 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 convenient Cortex core access functions. These CMSIS assess functions are used throughout the ADuCM350BBCZ Software Package 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 function for ITM messaging, etc. CMSIS also defines a number of naming conventions and various typedefs that are used throughout the ADuCM350BBCZ Software Package.

Please consult *The Definitive Guide to the ARM Cortex-M3*[IV] reference or the www.arm.com website for complete CMSIS details.

9.2 Interrupt Vector Table

The interrupt vector table is typically located at the start of flash memory. This placement is managed by the default linker control file:
`<EWARM_root>\arm\config\linker\AnalogDevices\ADuCM350BBCZ.icf.`

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 priority of which of which 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.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).

  The default MSP value is initialized in the startup code using the IAR-specific, `__sfe("CSTACK")` C-Language directive, thusly creating linkage between the C Compiler used to generate the IVT initialization data and the linker control file (*.icf) which defines the C-Runtime Stack placement.

- 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.c`. The location is initialized with the reset interrupt handler function, also defined in `startup.c`. 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. Typically, weak bindings give a fail-safe method that should be overridden to provide user-specific exception handling. Overriding weakly-bound functions is a simple matter of just re-implementing the same handler function name without the `WEAK` binding attribute. Handlers that are not overridden will keep the weakly-bound fail-safe implementation.

  Note: see the IAR Development Guide for documentation on `WEAK` binding.

- 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 ADuCM350BBCZ processor, there are 61 (0-60) such peripheral interrupts. Each peripheral interrupt has a dedicated interrupt priority register that may be programmed at run-time to manage interrupt dispatching.

9.3 Startup.c Content

The `<ADuCM350BBCZ_root>\src\startup.c` file is required for every ADuCM350BBCZ application. This file (and companion `<ADuCM350BBCZ_root>\src\startup.h` include file) is largely defined by the CMSIS standard and contains:

- Compiler Abstractions

  Macros that abstract various compiler-specific directives into a general form that can be used by the source code without regard to the underlying code-generation implementation. Examples of compiler abstractions include: `ATTRIBUTE_INTERRUPT, KEEP_VAR, WEAK_PROTO, WEAK_FUNC, VECTOR_SECTION, SECTION_PLACE`, etc.

- Default (Weak) Interrupt Handler Prototypes and Implementations for all Exceptions and Peripheral interrupts.

- IVT allocation and Initialization constructs.

- System Reset Handler Implementation

  The `startup.c` file contains the `ResetISR()` interrupt handler function. This is any program's entry point and is called directly by the boot kernel after the various pre-boot tasks are complete.

  There are only a few tasks the reset handler performs:

  - Reinitialize the main stack pointer (MSP) by copying the value from IVT[0] to the MSP register. This is done to reset the MSP after the boot kernel has made the call the the reset vector, in case it has left any garbage on the stack. The CMSIS accessor function `__set_MSP()` is used to reset the MSP.

  - Enable the entire 16k of Bank0 SRAM so that it is preserved during system hibernation. This may be disabled to save power during hibernation, but at the cost of less preserved SRAM. See the ADuCM350BBCZ HRM for details on hibernation and SRAM.

  - Conditionally relocate the IVT to SRAM. This is dependent on the **`RELOCATE_IVT`** build macro. Generally, the IVT is better left in flash to save on SRAM, but if the table content needs to be run-time-modified (as in running the OSAL and/or an RTOS), the table must be relocated to SRAM.

    If the table *is* relocated, the new IVT must then be activated by setting the core "interrupt vector table offset register" (`VTOR`) to point to the new IVT location.

    Note: The reserved area for the IVT relocation is uninitialized at startup and must not be activated until <u>after</u> the default (ROM-based) IVT has been copied. The relocated IVT activation is handled during `SystemInit()` (see below).

- Various compiler-specific C Run-Time Startup libraries are called (or not), depending on the compiler being used. For IAR, this includes expanding out any packed initialization data generated by linker compression, asserting global data, initializing internal run-time libraries, and finally call user `main()`.
- The reset vector handler is not supposed to return because the boot kernel is not expecting this.

The `startup.c` file is a required and integral component for every ADuCM350BBCZ application.

9.4 System.c Content

The file `<ADuCM350BBCZ_root>\src\system.c` is another CMSIS prescribed file implementing a number of required CMSIS APIs (`SystemInit(), SystemCoreClockUpdate(),` etc.) In addition, the `system.c` file for the ADuCM350BBCZ includes a number of advanced features for managing:

- clock PLLs,
- clock enables,
- clock dividers,
- clock multiplexing,
- clock frequencies,
- clock state transitions,
- low-power modes,
- etc.

The `system.c` file is a required and integral component for every ADuCM350BBCZ application.

The critical `system.c` file lays the framework for all system clocking control and for system startup. The critical system functions implemented by `system.c` include:

- `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 Runtime 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 (see `startup.c` description) 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 ROM-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.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.

- `SystemClockEnable(), SetSystemClockDivider(), SystemEnableClockSource(), etc.`

  The various clock control APIs provide the means for managing all system clock behavior. The various examples illustrate the use of these APIs.

- `SystemEnterLowPowerMode() & SystemExitLowPowerMode()`

  This function pair allows the ADuCM350BBCZ to be put into various levels of low-power operation, including active, core sleep, system sleep, and full hibernation. Each mode progressively powers down more and more of the ADuCM350BBCZ. During full hibernation mode, up to half of the internal system RAM content may be preserved during hibernation, so what upon wakeup, normal application operation may resume without having to reinitialize things (e.g., application data, stack, heap, etc., *provided they are placed in the non-volatile RAM region*). Please read the Power Management Unit chapter in the HRM and the detailed API description in the Device Driver API documentation for more information.

- `SystemTransitionClocks()`

  This function supports safe clock transitions between different modes of operation. The measurement section of the ADuCM350BBCZ (AFE) requires a high clock for correct operation… the USB subsystem has even more demanding requirements for the clocking schema.

  Regardless of the task, running the ADuCM350BBCZ in fast clock mode is power hungry and so applications prefer to optimize clock usage depending on mode of operation, especially when a resource is not in an active mode. Reducing clock speed and/or gating off inactive clock domains (peripherals) greatly extend battery life. But transitioning the clock controls can be a bit tricky and must be handled with care.

  The `SystemTransitionClocks()`API is provided to encapsulate the complex task of transitioning the internal system clock controls between the various modes of operation under a simple API. The underlying implementation hides a state machine that is driven by input events passed as API parameters. Internally, the state machine executes the appropriate transition actions depending on the current state and the requested transition.

  It is highly recommended that all system clocking transitions be managed through this API.