Application Note

Debugging Sitara AM2x Microcontrollers

Eyal Cohen and Sahin Okur

ABSTRACT

This application note covers the various debugging tools and techniques available to users developing applications with Sitara™ AM2x microcontrollers (MCUs).

Table of Contents

Trademarks

Sitara™ and Code Composer Studio™ are trademarks of Texas Instruments.

1 Building for Debug

This section covers how to build your application for optimal debugging.

1.1 Disable Code Optimization

Before debugging your code, disable any compiler optimization. When compiler optimization is enabled, stepping through code can become unpredictable, and breakpoints sometimes cannot be set to the exact line in the C source code. This is because the optimizer can condense code and impact the correlation between the assembly instruction and the C source. **Due to this, the recommendation is to turn off compiler optimization when stepping through code.**

To disable compiler optimization, go to your project's properties > Build > Arm Compiler > Optimization, and set the optimization level to none or 0. If building with makefiles, this can be done by modifying the makefiles directly. For MCU+ SDK, this is typically done within the submodule's makefile (not the top-level makefile).

1.2 Using the Debug SDK Libraries

The SDK provides two versions of libs: "Debug" and "Release". The "Debug" version is built with optimization disabled, while the "Release" version is built with optimization enabled. **The recommendation is to use the "Debug" version of an SDK lib when debugging.**

The SDK libs follow the naming convention: "{library name}.{device}.{core}.{compiler}.{version}.lib"

For example, "drivers.am243x.m4f.ti-arm-clang.release.lib" implies the following:

- Library Name: Drivers Library
- Device: AM243x MCU
- Core: Arm® Cortex®-M4F Core
- Compiler: TI Arm Clang Compiler
- Version: Release

The libraries that are linked to your project can be configured in the project properties under Build > Arm Linker > File Search Path.

2 Code Composer Studio Stop-Mode Debugging

This section covers the key debug features offered by Code Composer Studio™ (CCS) for stop-mode debugging. Stop-mode debugging refers to debugging where stopping or halting the core is involved, as opposed to real-time debugging, which involves debugging without stopping the core (real-time debug is discussed in this application note in a later section).

2.1 Configuring the Debugger

The chip can be debugged using CCS via the JTAG port. If using a Sitara AM2x MCU evaluation board, you can use the on-board JTAG debug probe. There is also typically support for using your own JTAG debug probe via an external header.

To setup your board and debugger for CCS debug, see the *EVM Setup* section of the MCU+ SDK Getting Started guide for your device to setup your board and debugger for CCS debug.

2.2 Breakpoints and Watchpoints

This section covers breakpoints and watchpoints and how to use them.

Breakpoints are program locations where the processor must halt so that debugging can occur. Both hardware and software break points allow the core to halt at a given PC location.

Watchpoints are breakpoints that can be triggered to halt program execution when a particular memory read or write occurs. Watchpoints are extremely useful to catch exceptions, invalid memory boundary accesses, overrun buffers, and so forth and can be set to access any memory region, including Memory Mapped Registers (MMRs).

2.2.1 Software Breakpoints

A Software breakpoint is implemented as an opcode replacement. The debugger modifies the opcode by inserting an *estop_0* instruction where the previous instruction was. The program counter stops immediately before it executes the software breakpoint instruction. In general, this instruction is hidden from the main interface, but in certain instances this instruction is displayed in the Disassembly View. Software breakpoints can only be set in memory regions with write access (RAM), therefore, there is no theoretical limit to the number of software breakpoints that can be used.

To toggle a breakpoint, either double-click on the left side of either the line number in the source code view or the address in the disassembly view, or right-click → Toggle Breakpoint.

CCS allows you to single-step through the code in your program. With the breakpoint set, select Run > Step Into to step into a given function.

You can also select Run > Step Over that executes the function in a single step. This is useful when you do not want to enter a certain function when single-stepping through code.

2.2.2 Hardware Breakpoints

A Hardware breakpoint is implemented internally by the target hardware. The method used to do this is heavily dependent on the device or core, but typically the debugger writes the address to a register on the device and sets a flag to enable breakpoints. These registers are not exposed to the IDE. A hardware breakpoint can be set in any memory type (RAM, Flash or ROM), but it is limited by the number of registers on the device. This is mandatory for the types of console I/O devices. Hardware breakpoints can also have a count, which determines the number of times a location is encountered before a breakpoint is generated. For example, if the count is 2, a breakpoint is generated every second time. Hardware breakpoints make use of dedicated registers and hence are limited in number. The AM243x supports 8 hardware breakpoints and 8 watchpoints. To see how many hardware breakpoints and watchpoints are supported per device family, see the device-specific technical reference manual.

2.2.3 Watchpoints

Watchpoints are a special category of hardware breakpoints that can be triggered for a particular memory read or write. Watchpoints are extremely useful to catch exceptions, invalid memory boundary accesses, overrun buffers, and so forth and can be set to access any memory region, including Memory Mapped Registers (MMRs).

To set a Watchpoint, highlight a variable in the source code editor, right click and select Breakpoint → Hardware Watchpoint. For example, right click on the variable *gGpioIntrDone* and add a watchpoint. Whenever you press the general-purpose input/output (GPIO) push button, a breakpoint will trigger when gGpioIntrDone increments at the line *gGpioIntrDone++;* in *GPIO_bankIsrFxn().*

A common issue causing software instability is stack overflow. When building a project, the stack size is typically specified in the project linker, that corresponding size is allocated for the stack by the linker. A hardware watchpoint can be set to monitor when the location __STACK_END - 2 is written to, which indicates that a stack overflow has occurred.

2.3 Inspecting Device Registers

To open the Registers view: View → Registers.

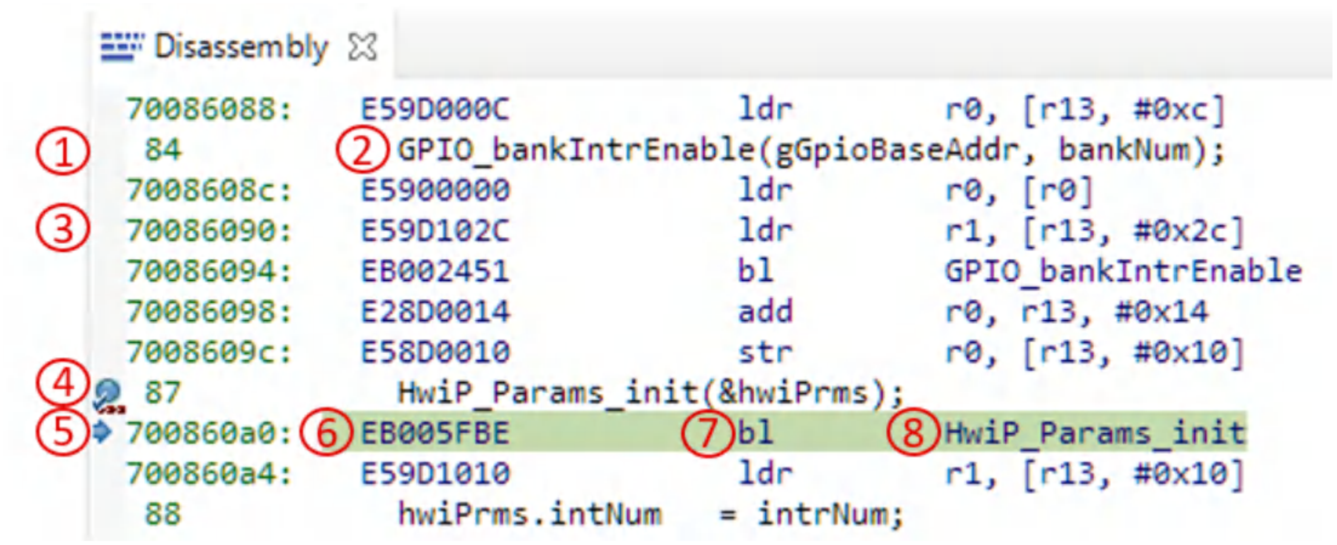
The Registers view allows for viewing and editing the contents of core and peripheral registers of the device, including bitfields and individual bits.

Type Ctrl+F or right click → Find to search for any register in the register window.

2.4 Inspecting Disassembly

This section covers how to use the CCS Disassembly view to get further insight into your running software.

The CPU opcodes of the software executing on the target can be viewed in the Disassembly window (View → Disassembly) of the core. Assembly step into and step over buttons in the debug window can be used to step through the disassembly.



```
70086088:    E59D000C        ldr       r0, [r13, #0xc]
 84            GPIO_bankIntrEnable(gGpioBaseAddr, bankNum);
7008608c:    E5900000        ldr       r0, [r0]
70086090:    E59D102C        ldr       r1, [r13, #0x2c]
70086094:    EB002451        bl        GPIO_bankIntrEnable
70086098:    E28D0014        add       r0, r13, #0x14
7008609c:    E58D0010        str       r0, [r13, #0x10]
 87            HwiP_Params_init(&hwiPrms);
700860a0:    EB005FBE        bl        HwiP_Params_init
700860a4:    E59D1010        ldr       r1, [r13, #0x10]
 88            hwiPrms.intNum    = intrNum;
```

The Disassembly view contains several points of information:

- Line numbers of the source file in context
- C source code at the line number
- Program addresses
- Breakpoint indicators
- Program Counter - the next instruction to be executed
- Opcodes in hex format
- Disassembled instructions
- If the opcode references a function or variable, their names are also shown

3 Debug Logging

This section covers the various options supported for debug (printf-style) logging.

3.1 Logging Methods

The Driver Porting Layer (DPL) of the SDK contains a Debug Log module that provides APIs for debug logging and is the recommended method for providing "printf" style logging. These options can be configured in the SysConfig GUI under TI Driver Porting Layer (DPL) → Debug Log.

There are 3 logging methods supported by the Debug Log module:

- **CCS Console Logging (via JTAG)** – To enable logging to the CCS Console, check "Enable CCS Log" in the SysConfig window. To open the console in CCS, click View → Console.
- **UART Logging** - To enable UART logging, check "Enable UART Log." This automatically adds a universal asynchronous receiver/transmitter (UART) driver instance to be used with the logger.

To view the UART Log output, open the CCS Terminal by going to View → Terminal. In the Terminal window, click "Open a Terminal" and configure the Terminal based on the UART settings in SysConfig.

The default UART terminal settings are:

Parameter	Value
Baud Rate	115200
Data Length	8-bit
Parity Type	None
Stop Bit	1-bit
Flow Control	None

- **Shared Memory Logging** - The Shared Memory Logger allows you to share and output logs with other cores in the device.
  - Shared Memory Log Writer - Logs the debug output to shared memory. Another core should have the Shared Memory Log Reader enabled to read the debug log strings.
  - Shared Memory Log Reader (FreeRTOS only) - Reads the string logged by another core and outputs it to the logs selected on the core. This needs task support so the reader core needs to be running FreeRTOS

A core can only be designed as a reader or writer, not both. When a core has the Shared Memory Log Reader enabled, the option to use the Shared Memory Writer is ignored. When the Shared Memory Logger is enabled, *SysConfig* generates the shared memory section to be used for logging in *ti_dpl_config.c*.

```
`DebugP_ShmLog gDebugShmLog[CSL_CORE_ID_MAX] __attribute__((aligned(128),
section(".bss.log_shared_mem")));`
```

This section *.bss.log_shared_mem* from the above code snippet needs to be reserved for shared memory logging and needs to be allocated at the same shared memory address location for all cores. This section is reserved in the linker command file. This section also needs to be marked as non-cache in the MPU/MMU module within SysConfig.

3.2 Log Zones

The Debug Log module has support for "Log Zones", which allows for enabling and disabling different types of debug log messages. These Log Zones are used in the SDK drivers but can also be used in the application. The following Log Zones are supported:

- Error Log Zone
- Warning Log Zone
- Info Log Zone

Having the debug log messages separated into different zones provides the ability to easily change the verbosity of the drivers.

3.3 Asserts

The Debug Log module allows for assert testing, If a given expression is evaluated to 0, the application disables the interrupts and loop forever. The application also logs the file name and line number where the assert occurred.

3.4 Example Usage

Include the below file to access the APIs:

```
#include <stdio.h>
#include <kernel/dpl/DebugP.h>
```

Example usage for assert:

```
void *addr = NULL
/* This will assert when address is NULL */
DebugP_assert(addr!=NULL);
```

Example usage for logs:

```
Uint32_t value = 10;
char *str = "Sitara AM2x debugging";
/* Use snprintf to format the string and then call logging function */
DebugP_log("This is %s and value is = %d", str, value);
```

Example usage for scanf:

```
DebugP_log("Enter a 32b number \r \n");
value32 = 0;
DebugP_scanf("%d", &value32);
DebugP_log("32b value = %d\r\n", value32);
```

4 Multi-Core Debug

The Debug view displays the stack frames for each debuggable core on a multi-core target. Most of the various views in the Debug perspective (Register, Variables, Disassembly, Memory Browser, and so forth) reflect the context of the highlighted stack frame for the specified core.

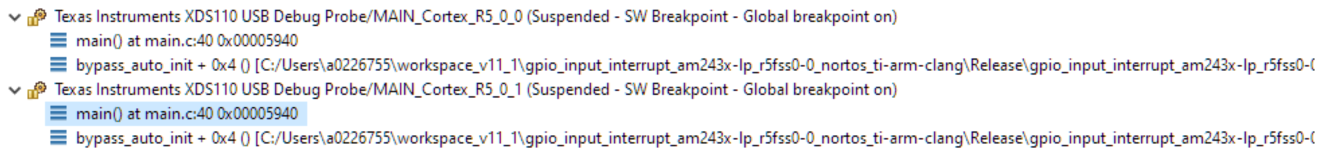
4.1 Grouping Cores

Commands can be sent to a specific set of cores at the same time. This can be done by "grouping" the cores of interest in the **Debug** view.

4.1.1 Fixed Group

Once a debug session is started, you can create a more permanent group. This **Fixed Group** has a specific node in the **Debug** view that has its own debug context. Selecting this group debug context causes debug commands to be sent to all group members without the need to select them individually. Note, that while the commands are sent simultaneously, how synchronously the commands are executed depends on if the hardware target itself supports synchronous execution.

In the screenshot below, a **Fixed Group** is created for just the first and second CPUs in the **Debug** view by multi-selecting them and then using the **Group core(s)** option:



```
Texas Instruments XDS110 USB Debug Probe/MAIN_Cortex_R5_0_0 (Suspended - SW Breakpoint - Global breakpoint on)
    main() at main.c:40 0x00005940
    bypass_auto_init + 0x4 () [C:/Users\a0226755\workspace_v11_1\gpio_input_interrupt_am243x-lp_r5fss0-0_nortos_ti-arm-clang\Release\gpio_input_interrupt_am243x-lp_r5fss0-(
Texas Instruments XDS110 USB Debug Probe/MAIN_Cortex_R5_0_1 (Suspended - SW Breakpoint - Global breakpoint on)
    main() at main.c:40 0x00005940
    bypass_auto_init + 0x4 () [C:/Users\a0226755\workspace_v11_1\gpio_input_interrupt_am243x-lp_r5fss0-0_nortos_ti-arm-clang\Release\gpio_input_interrupt_am243x-lp_r5fss0-(
```

This causes a new group called **Group 1** to appear in the **Debug** view, with the CPUs members.

4.1.2 Hiding Cores

It is possible to filter the list of CPUs visible in the **Debug** view that will not be used. This helps to avoid clutter in the view to avoid accidentally selecting a debug context of a CPU that is not to be used. To hide CPUs:

1. Multi-select the CPUs to hide, right-click and select **Hide core(s)** in the context menu.
2. Selected CPUs will disappear from the view,
3. Unhide all CPUs with **Show all cores** option,

Note that you can set this filter before starting a debug session by specifying which CPUs to display in the **Debug Configuration**

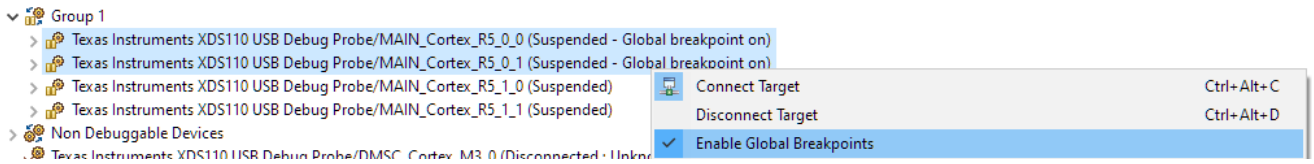
4.2 Using Multiple Workbench Windows

Multiple main windows (called **Wordbench** windows) can be treated to have each window dedicated to a specified core during the debug session. A new window can be created using the **Window → New Window** option. A **Workbench** window has its own debug context (**Workbench** window 1 can show data for core 1 while **Workbench** window 2 shows data for core 2, and so forth). Note that creating a new window does not mean a new debug session. Each window is associated with the same debug session but can be specified, using the **Debug** view, to reflect the context of a different core.

4.3 Global Breakpoints

Each debug context can be configured for **Global Breakpoints**. This feature essentially makes breakpoints global across all debug contexts that have it enabled. For example, if the debug contexts for CPU 1 and 3 have global breakpoints enabled, and CPU 1 hits a breakpoint, CPU 3 will also be halted (if it was running). And vice versa.

The screenshot below shows an example of enabling global breakpoints for the whole group. This enables global breakpoints for each CPU in the group:


Group 1
Texas Instruments XDS110 USB Debug Probe/MAIN_Cortex_R5_0_0 (Suspended - Global breakpoint on)
Texas Instruments XDS110 USB Debug Probe/MAIN_Cortex_R5_0_1 (Suspended - Global breakpoint on)
Texas Instruments XDS110 USB Debug Probe/MAIN_Cortex_R5_1_0 (Suspended)
Texas Instruments XDS110 USB Debug Probe/MAIN_Cortex_R5_1_1 (Suspended)
Non Debuggable Devices
Texas Instruments XDS110 USB Debug Probe/DMSC_Cortex_M3_0 (Disconnected : Unkn
Connect Target Ctrl+Alt+C
Disconnect Target Ctrl+Alt+D
✓ Enable Global Breakpoints

5 Debugging Arm Cortex-R5 Exceptions

This section covers the various exceptions that can occur on the Arm Cortex-R5 core and the techniques to debug them.

An “Exception” is an event that makes the processor temporarily halt the normal flow of program execution, for example, to service an interrupt from a peripheral. Before attempting to handle an exception, the processor preserves the critical parts of the current processor state so that the original program can resume when the handler routine has finished. In practical situations, exceptions can be mainly categorized into the following:

- Interrupts (Normal Interrupts IRQs and Fast Interrupts FIQs/NMIs)
- Aborts (Data Abort, Prefetch Abort)
- Undefined Instruction (UNDEF) exceptions

5.1 Exception Priority Order

When several exceptions occur simultaneously, they are serviced in a fixed order of priority. Each exception is handled in turn before execution of the user program continues. It is not possible for all exceptions to occur concurrently. For example, the Undefined Instruction and SVC exceptions are mutually exclusive because they are both triggered by executing an instruction. Because the Data Abort exception has a higher priority than the FIQ exception, the Data Abort is actually registered before the FIQ is handled. The Data Abort handler is entered, but control is then passed immediately to the FIQ handler. When the FIQ has been handled, control returns to the Data Abort handler. This means that the data transfer error does not escape detection as it would if the FIQ were handled first.

Exception	Priority
Reset	1 (Highest)
Data Abort	2
FIQ	3
IRQ	4
Prefetch Abort	5
SVC	6
Undefined Abort	6 (lowest)

5.2 Aborts

When an abort happens, the program gets halted at the Exception Vector Table in address 0xFFFF00##:

Value of V bit	Exception vector base location
0	0x00000000
1 (HIVECS)	0xFFFF0000

The last two nibbles in the address (0xFFFF00##) indicates the type of abort as shown.

Exception	Offset From Vector Base
Reset	0x00
Undefined Instruction	0x04
Software Interrupt	0x08
Abort (prefetch)	0x0C
Abort (data)	0x10
IRQ	0x18

Exception	Offset From Vector Base
FIQ	0x1C

All exceptions end up in the address specified in the Exception Vector Table. The program execution can then branch to application-specific handlers. The branch target address differs based on the application. Two such examples are discussed below:

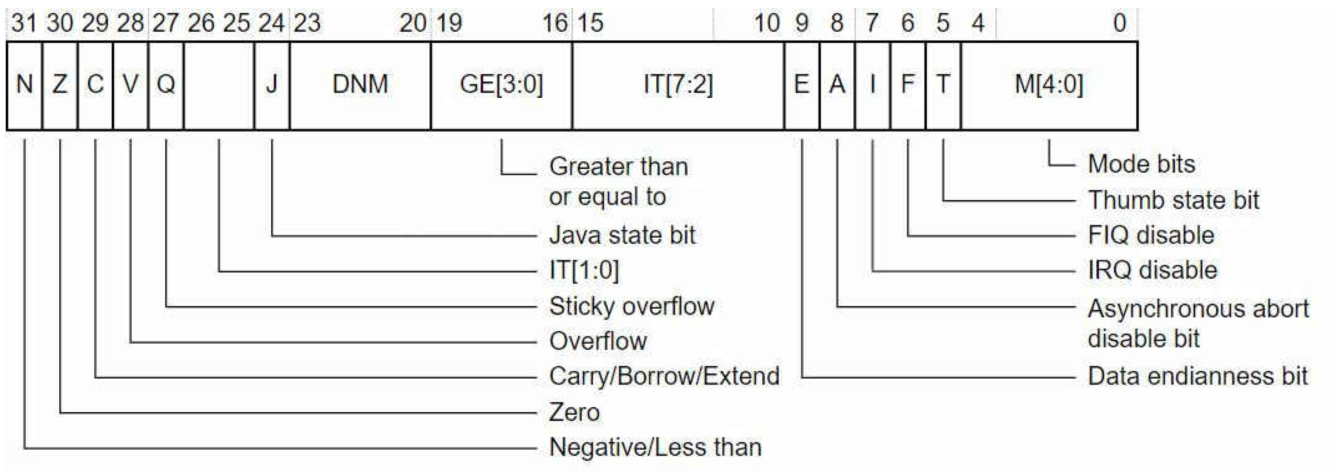
- An application can use the default exception handlers designed as a “trap” where the execution gets stuck. Below is an example to a prefetch abort handler (an infinte loop):

```
Void __attribute__((interrupt("ABORT"), section(".text.hwi"))) HWiP_prefetch_abort_handler(void)
{
   volatile uint32_t loop = 1;
   while(loop)
      ;
}
```

- An implementation can have advanced exception handling capabilities in the OS, where the details of the exception will be read and a corresponding error code are notified by the custom OS error handler. In such cases, details of the error codes can be checked to understand the actual exception that was triggered. R13, R14, and SPSR registers of the corresponding exception can be read for debugging the issue.

There are three important Arm Cortex-R5 registers that can also be used to confirm the current state of the processor.

CPSR:



The CPSR can be used to verify the current mode of the processor. The mode bits of the CPSR register can be used to check if the current mode is Abort:

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	**Abort**
11011	Undefined
11111	System

SPSR:

The SPSR can be used to check the previous mode just before entering the exception. For example, if the processor moves from System to Abort Mode, SPSR shows the mode as “System” while CPSR shows the mode as “Abort”. The bit definitions of SPSR register are the same as that of the CPSR register.

R14 Register (Link Register):

The R14 register is used to find the actual instruction or function call that caused the synchronous abort. The actual address of the instruction that triggered the Exception is R14 - x, where “x” depends on the type of exception.

Aborts are usually unintended exceptions resulting due to invalid or unsuccessful access of memory. Some of the causes for aborts are as follows.

- Permission fault indicated by the Memory Protection Unit (MPU)
- Error detected in the data by the ECC checking logic

If the exception is confirmed to be a Data Abort, as the first step check the value of the Data Fault Status Register (DFSR) of the Cortex-R CPU. The DFSR holds status information about the last data fault.

Figure 5-1 shows the **DFSR** register bit assignments.

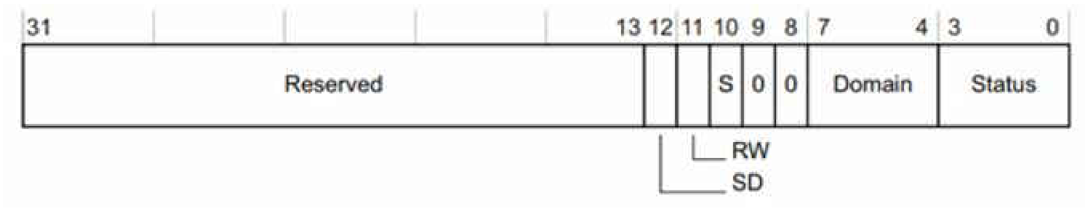


Figure 5-1. DFSR bits

Use the “S” Bit [10] and “Status Bits” [0:3] to understand the nature of the Data Abort. For status description, see Table 5-1.

Table 5-1. Status Description

Priority	Sources	FSR[10,3:0]	FAR
Highest	Alignment	0b00001	Valid
	Background	0b00000	Valid
	Permission	0b01101	Valid
	Synchronous external abort	0b01000	Valid
	Asynchronous external abort	0b10110	Unpredictable
	Synchronous parity or ECC error	0b11001	Valid
	Asynchronous parity or ECC error	0b11000	Unpredictable
	Debug event	0b00010	Unpredictable

SD Bit:

The SD Bit distinguishes between an AXI Decode or Slave error on an external abort. This bit is valid only for external aborts. For all other types of abort, this bit is set to zero.

- **0 = AXI** Decode error (DECERR) or AHB error caused the abort, generated, typically by an interconnect component, to indicate that there is no slave at the transaction address (The address you requested is not valid)
- **1 = AXI** Slave error (SLVERR) or unsupported exclusive access caused the abort. Used when the access has reached the slave successfully, but the slave wishes to return an error condition to the originating master an error condition to the originating master. (Valid address, but slave is unable to do the requested operation)

RW Bit:

The RW bit indicates whether a read or write access caused the abort.

- **0 =** read access caused the abort

- **1 =** write access caused the abort

5.2.1 Data Aborts

5.2.1.1 Alignment

This indicates that the memory access does not follow alignment requirements, which differ according to the memory attribute of the region:

- “Normal” supports unaligned support (this is configurable)
- “Device” / “Strongly Ordered” supports only aligned access

This means that if a region is configured as “Strongly Ordered” and you try to do an Un-Aligned Memory access, an Alignment Data Abort occurs. The Memory Map of the Cortex R5 has address range 0xEFFFFFFF to 0xFFFFFFFF configured as “Strongly Ordered” by default.

5.2.1.2 Background Aborts

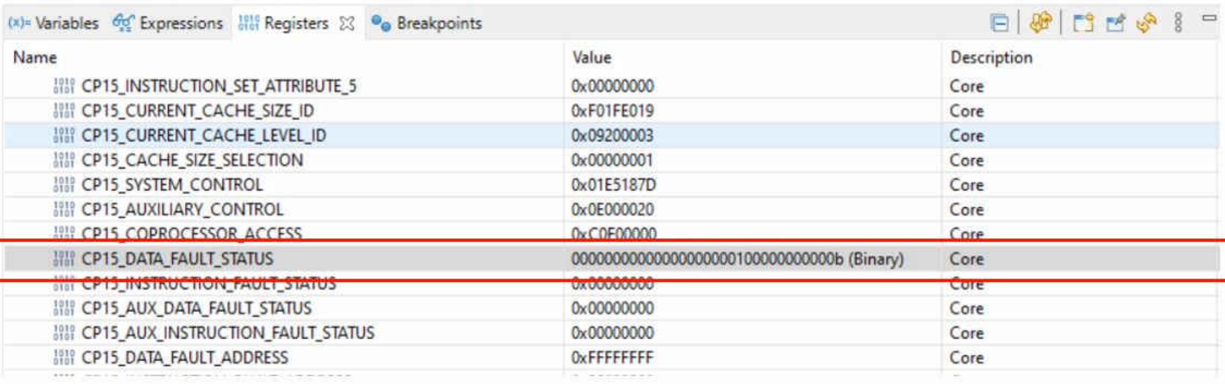
Memory Protection Unit (MPU) settings must be correct for any region that the CPU is going to access. If the address that the CPU issues falls outside any of the defined regions and the MPU is enabled, the MPU is hard-wired to abort the access. That is, all accesses for an address that is not mapped to a region in the MPU generate a background fault. A background fault does not occur if the background region is enabled and the access is Privileged. An MPU background fault might indicate a stack overflow, and be rectified by allocating more stack.

In the example below, in line 55, an address that falls outside any of the defined regions is trying to be written to.

Example:

```
void empty_main(void *args)
{
   Drivers_open();
   Board_driversOpen();
   *((volatile uint32_t*) 0xFFFFFFFF = 0x12;

   Board_driversClose();
   Drivers_close();
}
```

By checking the DATA_FAULT_STATUS register (DFSR), bits [10, 3:0], you can see that it fits to Background Abort.



Name	Value	Description
CP15_INSTRUCTION_SET_ATTRIBUTE_5	0x00000000	Core
CP15_CURRENT_CACHE_SIZE_ID	0xF01FE019	Core
CP15_CURRENT_CACHE_LEVEL_ID	0x09200003	Core
CP15_CACHE_SIZE_SELECTION	0x00000001	Core
CP15_SYSTEM_CONTROL	0x01E5187D	Core
CP15_AUXILIARY_CONTROL	0x0E000020	Core
CP15_COPROCESSOR_ACCESS	0xC0E00000	Core
CP15_DATA_FAULT_STATUS	00000000000000000000100000000000b (Binary)	Core
CP15_INSTRUCTION_FAULT_STATUS	0x00000000	Core
CP15_AUX_DATA_FAULT_STATUS	0x00000000	Core
CP15_AUX_INSTRUCTION_FAULT_STATUS	0x00000000	Core
CP15_DATA_FAULT_ADDRESS	0xFFFFFFFF	Core

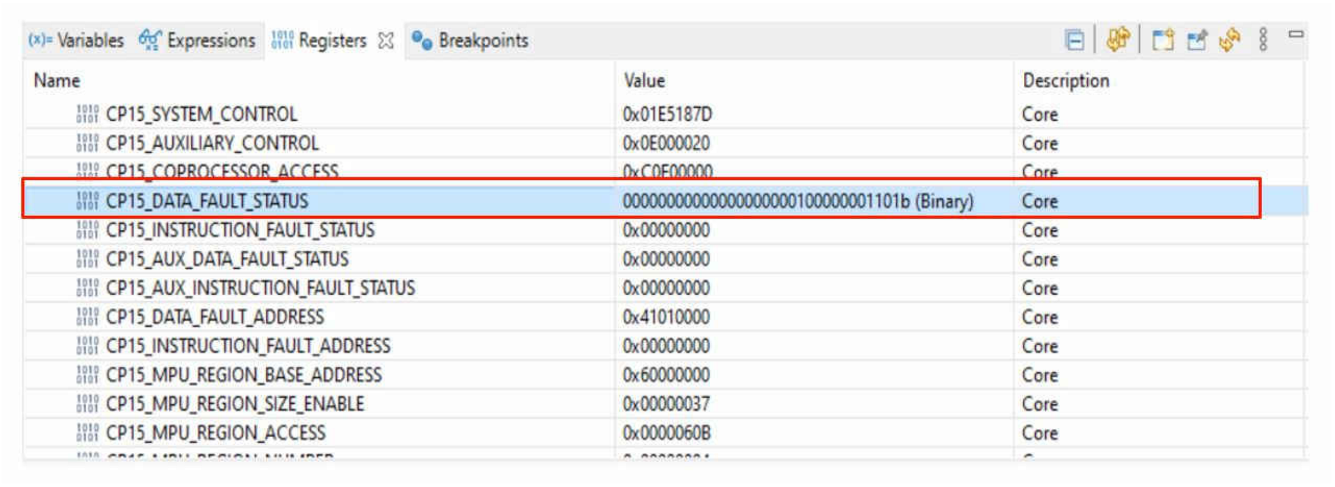
5.2.1.3 Permission

This can happen when MPU settings prevent the access of a region. For example, if a User mode application attempts to access a Privileged mode access only region a permission fault occurs.

Example:

```
__attribute__((section("mySection"))) Bool test;

void empty_main(void *args)
{
   Drivers_open();
   Board_driversOpen();
   test = 1;

   Board_driversClose();
   Drivers_close();
}
```

A read-only memory section was created by configuring the MPU access permission attributes. Then, a Boolean variable *test* is placed in this read-only section (line 48). When trying to write to this variable (line 56), an MPU permission fault was triggered.



Name	Value	Description
CP15_SYSTEM_CONTROL	0x01E5187D	Core
CP15_AUXILIARY_CONTROL	0x0E000020	Core
CP15_COPROCESSOR_ACCESS	0xC0F00000	Core
CP15_DATA_FAULT_STATUS	00000000000000000000100000001101b (Binary)	Core
CP15_INSTRUCTION_FAULT_STATUS	0x00000000	Core
CP15_AUX_DATA_FAULT_STATUS	0x00000000	Core
CP15_AUX_INSTRUCTION_FAULT_STATUS	0x00000000	Core
CP15_DATA_FAULT_ADDRESS	0x41010000	Core
CP15_INSTRUCTION_FAULT_ADDRESS	0x00000000	Core
CP15_MPU_REGION_BASE_ADDRESS	0x60000000	Core
CP15_MPU_REGION_SIZE_ENABLE	0x00000037	Core
CP15_MPU_REGION_ACCESS	0x0000060B	Core

5.2.1.4 Synchronous/Asynchronous External

This happens when the access has been transferred from the CPU to the AXI/AHB Bus and encountered an error. This is the most common fault type that happens with Data Abort. If the Abort is Synchronous, you can check the actual memory address that when accessed resulted in Data Abort using Data Fault Address Register (DFAR), which holds the address of the fault when a synchronous abort occurred.

5.2.1.5 Synchronous/Asynchronous ECC

This happens if an ECC error is detected at TCM interfaces or in the cache.

5.2.2 Synchronous/Asynchronous Aborts

5.2.2.1 Changing an Asynchronous Abort to a Synchronous Abort

If store instructions to peripheral areas cause the exceptions, you can configure the corresponding peripheral area as “Strongly-Ordered” via the Arm MPU (using SysConfig). A write to “Strongly-Ordered Memory” can complete only when it reaches the peripheral or memory component accessed by the write. However, this may impact the performance because a “Strongly Ordered” MPU attribute waits for the access to be completed before processing the next data access.

5.2.2.2 Synchronous Abort

Also known as a precise abort, is one for which the exception is ensured to be taken on the instruction that generated the aborting memory access. The abort handler can use the value in the Link Register (*R14*) to determine which instruction generated the abort, and the value in the Saved Program Status Register (*SPSR*) to determine the state of the processor when the abort occurred. This happens when the access has been transferred from the CPU to the AXI/AHB Bus and encountered an error. This is the most common fault type that happens with Data Abort. If the Abort is Synchronous, you can check the actual memory address that when accessed resulted in Data Abort using Data Fault Address Register (*DFAR*), which holds the address of the fault when a synchronous abort occurred.

5.2.2.3 Asynchronous Abort

Also known as an imprecise abort, is one for which the exception is taken on a later instruction than the instruction that generated the aborting memory access. Asynchronous faults are comparatively difficult to analyze because you cannot trace the exact location that resulted in the abort unlike the *DFAR* register that is used in Synchronous Faults. In general, “store” instructions (STB, STH, STR, STM/PUSH) to areas with “Normal” or “Device” memory attributes causing an error are asynchronous.

5.2.2.4 Debugging Asynchronous Abort

From the *DFSR* Register, you can check status bits, *SD* bit, and *RW* bit.

As mentioned above, SD indicated whether it is an internal AXI decode error or external AXI slave error, and RW indicates whether a read or write access caused an abort.

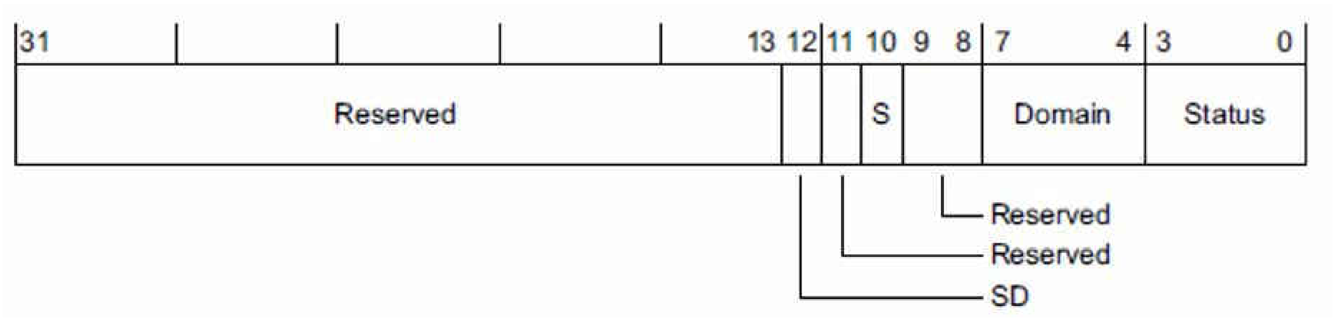
After the relevant information was extracted from the **DFSR**, the information can track the instruction that causes the abort:

- R14 – 8 is a location near the instruction that caused the exception
- Find a “store” instruction near R14 – 8, which can likely cause the exception

5.2.3 Prefetch Abort

Prefetch Abort (PABT) Exception occurs when an instruction fetch causes an error. When a Prefetch Abort occurs, the processor marks the prefetched instruction as invalid, but does not take the exception until the instruction is to be executed. If the instruction is not executed, for example because a branch occurs while it is in the pipeline, an abort does not occur. **All prefetch aborts are synchronous.** The difference between Undefined Instruction Abort and Prefetch Abort exception is that in case of prefetch, CPU is unable to fetch the instruction from the address; in an Undefined Instruction Exception, the CPU does not know what the instruction does.

The reason for Prefetch Abort can be analyzed by reading the Instruction Fault Status Register (**IFSR**), the Instruction Fault Address Register (**IFAR**), and the Auxiliary Instruction Fault Status Register (**AIFSR**).



IFAR contains the address where the CPU was trying to fetch an instruction from. The contents of **IFAR** is always valid for a Prefetch Abort, because all Prefetch Aborts are synchronous. **AIFSR** records additional information about the nature and location of the fault, for example ATCM or BTCM.

5.2.3.1 Possible Reasons for Prefetch Abort

- Improper MPU setting: If a permission fault has occurred based on the IFSR status, it is possible that one of the following conditions has occurred:
  - An instruction is being fetched from a location for which “Execute Never” attribute is set.
  - The target address read from IFAR has “Device” or “Strongly-Ordered” memory attribute. This implicitly means that these areas do not have executable code.
- ECC Error on the instruction read: ECC error is detected on the instruction reads. The IFAR register provides the address that caused the error to be detected. The auxiliary IFSR indicates source of the ECC error.
- Wrong return address or branch address - Return address being corrupted - Branch address is corrupted

5.2.3.2 Handling Prefetch Abort Exception

- Confirm whether the CPU control is stuck in Prefetch Abort Exception by checking the halt address. If the Offset is 0x0C, it indicates that the control has ended in a Prefetch Abort Handler.
- Check the status from **IFSR** and **IFAR** to determine the type of fault and the address leading to the abort.
- In the case of a “permission” fault, find the region in which the address read from the **IFAR** register falls under. The region can be checked for MPU violations for code area. (Execute Never setting, Strongly-ordered memory...).

5.2.4 Undefined Instruction

Undefined instruction exception can occur if the CPU does not understand the fetched instruction. There are no Fault Status and Fault address registers associated with this exception; only Link register (*R14*) provides relevant information. The instruction that caused the *UNDEF* abort is at *R14– 4*.

5.2.4.1 Possible Reasons for Undefined Instruction Exception

- Branch to RAM code that has been corrupted or not yet initialized with required functions
- Return address on the stack has been corrupted (for example, stack overflow or pop/push count mismatch)
- Function pointer is not initialized or corrupted

5.2.4.2 Handling Undefined Instruction Exception

- Confirm whether the CPU control is stuck in an Undefined Instruction exception by checking the halt address. If the address is 0x04, then the control has ended in an Undefined Instruction Exception.
- Check the value of the *R14* register. *R14 – 4* provides the address of the instruction that caused the undefined instruction exception. “X” depends on the mode (X=4 for ARM mode, and X=2 for Thumb mode).
- Check the instruction at the address read from *R14 - X*.
  - If it is a valid instruction, check whether the mode used (ARM or THUMB) for execution is correct (A mode mismatch for a valid instruction can cause undefined instruction exception).
  - If the instruction is invalid, check for address corruption or RAM corruption.

5.3 Fetching Core Registers Inside an Abort Handler

It is very useful to fetch core registers whenever an abort handler occurs. By analyzing the core registers, you can better understand what caused the abort handler. Below is an example of fetching the relevant registers whenever a data abort occurs. For this example, the "empty" project was used from the SDK.

By looking at the vector table that is located at ***HwiP_armv7r_vectors_nortos_asm.s*** file, you can see that all data aborts go to **HwiP_data_abort_handler.** This handler is a C function that is defined in **HwiP_armv7r_handlers_nortos.c** file.

In our SDK, abort handlers are written in C code. For every C function, the compiler creates prologue and epilogue sequences that manipulate some of the core registers (Stack Pointer register). Thus, if you fetch the Stack Pointer inside the C function, a wrong value will be stored. To avoid that, you must use the **"naked"** attribute and write a **naked function** in assembly. This attribute tells the compiler that the function is an embedded assembly function, and then prologue and epilogue sequences is not generated for that function by the compiler, and the Stack Pointer will point to the right value.

Inside **HwiP_armv7r_handlers_nortos.c**, a few modifications are done:

1. Change HwiP_data_abort_handler to be a **naked function**.
2. Create a new C handler (HwiP_data_abort_handler_**c**) that will be called from the naked function.

```
volatile uint32_t lrVal, pcVal, spVal;

__attribute__((naked)) void HwiP_data_abort_handler(void)
{
    //When data abort occurs, the processor first halts here

    /* Store Core Registers */
    __asm volatile ( "mov %0, lr" : "=r" ( lrVal ) );
    __asm volatile ( "mov %0, pc" : "=r" ( pcVal ) );
    __asm volatile ( "mov %0, sp" : "=r" ( spVal ) );


    /* Call the C version of the abort handler */
    __asm ("B HwiP_data_abort_handler_c");
}


void __attribute__((interrupt("ABORT"), section(".text.hwi"))) HwiP_data_abort_handler_c(uint32_t
*pMSP)
{
    printf("\nLR(r14): %x, PC:%x, SP:%x\n" lrVal, pcVal, spVal);

    volatile uint32_t loop = 1;
    while(loop)
        ;
}
```

Inside the naked function, assembly instructions were used to store the core registers. Then, the C version that was created of the abort handler was called. Inside the C version, the core registers can be printf or do other handling depends on the use-case.

In order to test our code, a data fault inside **empty_main()** was generated by forcing the processor to write to an undefined memory region (writing 0x12 to pMem variable):

```
void empty_main(void *args)
{
    volatile uint32_t lrVal, pcVal, spVal;
    /* Open drivers to open the UART driver for console */
    Drivers_open();
    Board_driversOpen();

    volatile uint32_t* pMem;
    pMem = 0xFFFFFFFF;
    *pMem = 0x12;

    while(1);

    Board_driversClose();
    Drivers_close();
}
```

Once this code was compiled and executed, a data fault is generated and makes the processor to first go through our naked function, store the core registers, call the C function and print the registers to the console.

6 Debugging Arm Cortex-M4 Exceptions

Arm Cortex M series have two types of exceptions:

- **System Exceptions** - Internally to the processor and generated by the processor itself. There are in total 15 system exceptions that are defined by Arm and supported by the Cortex-M processors.
- **Interrupts** - External to the core itself. Those interrupts are usually vendor specific and they are routed to the Nested Vectored Interrupt Controller (NVIC), that is responsible for their configuration. The M4F core of the Sitara 24X devices supports up to 64 interrupts. Using NVIC registers you can Enable / Disable / Pend various interrupts and read the status of the active and pending interrupts.

6.1 Exception Entry and Exit Sequence

6.1.1 Entry Sequence

Whenever there is an interrupt:

- Pending bit set - the according pending bit of the interrupt will be set on the register of the NVIC
- Stacking and Vector Fetch (Push xPSR, PC, LR, R12, R3, R2, R1, R0)
- The processor makes an entry into the handler (Active bit is set inside NVIC register)
- Clearing the pending bit of the NVIC register
- The processor mode is changed from Thread to Handler mode
- Handler code is executed
- The MSP ia used for any stack operations inside Handler mode

6.1.2 Exception Exit Sequence

The exception return mechanism is triggered using a special return address (EXC_RETURN). This address is generated during exception entry and it is stored in the Link Reigster (LR). In example, when the processor is in thread mode and the processor uses the Processor Stack Pointer (PSP) as its stack pointer, whenever an exception occurs, the processor does the stacking operations mentioned using PSP, and LR register is loaded with the EXC_RETURN. The way exception return happens, is by writing the value LR into the program counter (PC). By writing the LR value into the PC, exception return can be triggered. When that happens, the processor actually does an unstacking operation and comes to a normal execution. The EXC_RETURN is not an actual address, it includes different fields that are decoded by the processor (Stack frame type, return to thread mode or handler mode, return with MSP or PSP and so on...) so that the processor knows to which mode it should return.

6.1.3 Decoding EXC_RETURN Value

Table 6-1. Decoding EXC_RETURN Value

Bits	Descriptions	Values
31:28	EXC_RETURN indicator	0xF
27:5	Reserved	0xEFFFFF
4	Stack Frame Type	When FPU is not available, this bit is always 1
3	Return Mode	1 = Return to Thread Mode 0 = Return to Handler Mode
2	Return Stack	1 = Return with PSP 0 = Return with MSP
1	Reserved	0
0	Reserved	1

Figure 6-1 shows the flow of the exit sequence.

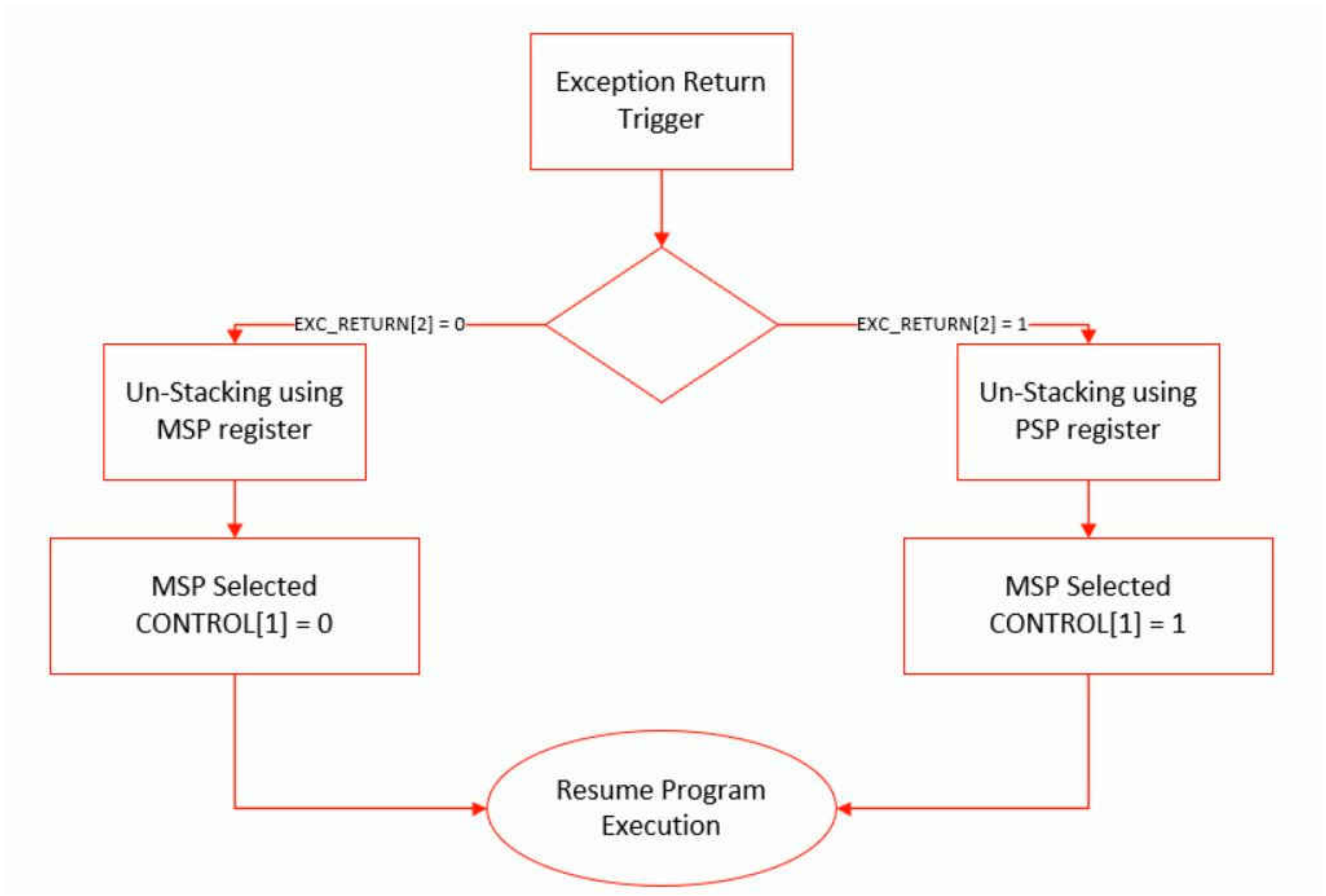

Exception Return Trigger
EXC_RETURN[2] = 0
EXC_RETURN[2] = 1
Un-Stacking using MSP register
Un-Stacking using PSP register
MSP Selected CONTROL[1] = 0
MSP Selected CONTROL[1] = 1
Resume Program Execution

Figure 6-1. Exit Sequence Flow

6.2 Faults Handling

Faults are system exceptions. They happen because of programmers handling processor by violating the design rules. Whenever a fault happens, the internal processor register is updated to record the type of fault, the address of the instructions that caused the fault, and if the associated fault is enabled, the exception handler is called by the processor. In example, if your code tries to divide by zero, then divide by zero fault is raised from the hardware that invokes the usage fault exception handler, if enabled. If not enabled, then the code ends up in a general hard fault handler.

6.2.1 There are 15 System Exceptions by Arm Cortex M Processors

Table 6-2. Arm Cortex-M Exceptions

Exception Number	IRQ Number	Exceptione Type	Priority	Vector Address or Offset	Activation
1	-	Reset	-3, Highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000 C	-
4	-12	MemManage	Configurable	0x00000010	Synchronous
5	-11	BusFault	Configurable	0x00000014	Synchronous when precise, Asynchronous when imprecise
6	-10	UsageFault	Configurable	0x00000018	Synchronous
7-10	-	Reserved	-	-	-

Table 6-2. Arm Cortex-M Exceptions (continued)

Exception Number	IRQ Number	Exceptione Type	Priority	Vector Address or Offset	Activation
11	-5	SVCall	Configurable	0x0000002C	Synchronous
12-13	-	Rserved	-	-	-
14	-2	PendSV	Configurable	0x00000038	Asynchronous
15	-1	SysTick	Configurable	0x0000003C	Asynchronous
16	0	Interrupt (IRQ)	Configurable	0x00000040	Asynchronous

Out of these system excpetions, HardFault, MemManage, BusFault and UsageFault are the only faults. There are exceptions that are responsible for reporting a fault. Hard fault exception is always enabled by the processor and its priority is hard coded and not configurable. You can disable it by the FAULTMASK register. Other faults need to be enabled by the user.

6.2.1.1 Causes of Faults

- Undefined Instruction
- Divide by zero
- Attemp to execute code from memory region which is marked as XN (Executed Never) to prevent code injection. In example, peripheral region is always marked as XN area, if the processor will try to execute code from this region it will cause a fault.
- MPU (Memory Protect Unit) - region access violation by the code, in example writing to a "read only" area
- Unaligned data access
- Returning to thread mode keeping active interrupt alive - when the interrupt is in active state you must finish it, you can't go back to thread mode if not finishing handling an interrupt in the handler mode.
- Debug monitor setting and related exceptions
- Bus Error, in example no respone from memory device like SDRAM and so on...
- Executing SVC instruction inside SVC handler

6.2.2 HardFault Exception

A HardFault exception occurs because of an error during exception processing, or whenever any other exception can not be managed by other exception handler. This exception has a fixed priority (-1) after reset and NMI. This means it has higher priority than any exception with configurable priority.

6.2.2.1 Causes of HardFault Exception

- Escalation of configurable fault exceptions, in example, if a program tries to divide by zero, and the divide by zero trap is disabled - then the processor enda up in the hard-fault handler instead of Usage fault handler.
- Bus error returned during a vector fetch
- Execution of a break point instruction in the software when both halt mode and debug mode are disabled
- Executing SVC instruction inside the SVC handler causea the processor to end in the HardFault handler.

6.2.3 Configurable Fault Exceptions

The priority of the following system faults are configurable in contrast to the hard fault exception.

6.2.3.1 Mem Manage Fault Exception

This is a configurable fault exception that is disabled by default (can be enabled by the SHCSR register). This fault detects memory access violations to regions that are defined in the Memory Management Unit (MPU). For example, code execution from a memory region with read/write access only. Another example is when unprivileged thread mode code (such as user application or RTOS task) tries to access memory region that is marked as "privileged access only" by the MPU. Also, this fault can occur due to XN (eXecute Never) issues - whenever the processor tries to execute an instruction from a XN area.

Table 6-3 shows the behavior of accesses to each region in the memory map.

Table 6-3. Mem Manage Fault Exception

Address Range	Memory Region	Memory Type	XN	Description
0x00000000 - 0x1FFFFFFF	Code	Normal	-	Executable region for program code. You can also put data here.
0x20000000 - 0x3FFFFFFF	SRAM	Normal	-	Executable region for data. You can also put code here. This region includes bit band area.
0x40000000 - 0x5FFFFFFF	Peripheral	Device	XN	This region include bit band area
0x60000000 - 0x9FFFFFFF	External RAM	Normal	-	Executable region for data
0xA0000000 - 0xDFFFFFFF	External Device	Device	XN	External Device Memory
0xE0000000 - 0xE00FFFFF	PPB - Private Peripheral Bus	Strongly-Ordered	XN	This region include the NVIC, System Timer and System Control Block
0xE0100000 - 0xFFFFFFFF	Device	Device	XN	Implementation-specific

Note

The Code, SRAM, and external RAM regions can hold programs. However, Arm recommends that programs always use the Code region. This is because the processor has separate buses that enable instruction fetches and data accesses to occur simultaneously.

6.2.3.2 BusFault Exception

Detects memory access errors on instruction fetch, data read/write, interrupt vector fetch, and register stacking (save/restore) on interrupt (entry/exit). This fault can also occur due to unprivileged access to the Private Peripheral Bus (PPB).

6.2.3.3 Usage Fault Exception

Can be caused by:

- Execution of undefined instruction. Cortex M series support only thumb ISA, so executing any instruction outside this ISA (like Arm ISA) would result in a fault.
- Executing an FPU instruction when the FPU is disabled
- Trying to return to Thread mode when an interrupt is still handled in the background
- Dividing by zero
- Clearing the 'T' bit to 0 instead of keeping it '1' (The T bit of the processor decides Arm state or THUMB state. For Cortex M series, it should be maintained at '1' since the since the processor does not support the Arm ISA)

6.2.4 Control Registers

The System control block (SCB) provides system implementation information, and system control. This includes configuration, control, and reporting of the system exceptions. Some of its registers are used to control fault exceptions:

- **CCR** - Configuration and Control Register, controls the behavior of the UsageFault for divideby-zero and unaligned memory accesses
- **SHP** - System Handler Priority Registers, control the exception priority
- **SHCRS** - System Handler Control and State Register, enables the system handlers, and indicates the pending status of the BusFault, MemManage fault, and SVC exceptions

Register	Address	Description
CCR	0xE000ED14	Configuration and Control Register: contains enable bits for trapping of divide-by-zero and unaligned accesses with the UsageFault.
SHP	0xE000ED18	System Handler Priority registers: control the priority of exception handlers.
SHCRS	0xE000ED24	The SHCSR enables the system handlers, and indicates: • The pending status of the BusFault, MemManage fault, and SVC exceptions • The active status of the system handlers

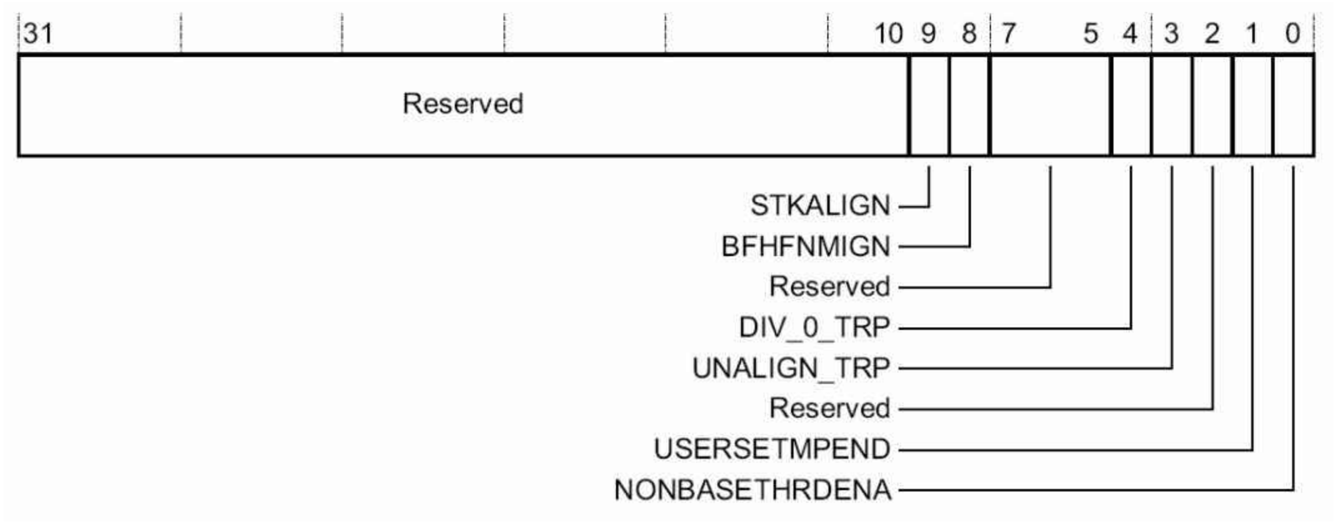


Figure 6-2. CCR - Configuration and Control Register

The following bits of the CCR register control the behavior of the Usage Fault:

DIV_0_TRP: Enables UsageFault when the processor executes an SDIV or UDIV instruction with a divisor of 0:

- When '0' = do not trap divide by 0; a divide by 0 returns a quotient of 0.
- When '1' = trap divide by 0.

UNALIGN_TRP: Enable UsageFault when a memory access to unaligned addresses are performed:

- When '0' = do not trap unaligned halfword and word accesses
- When '1' = trap unaligned halfword and word accesses; an unaligned access generates a UsageFault

6.2.4.1 SHP - System Handler Priority Register

The SHP registers set the priority level of exception handlers that their priorities are configurable:

- SHP[0]: Priority of the Memory Management Fault
- SHP[1]: Priority of the Bus Fault
- SHP[2]: Priority of the Usage Fault

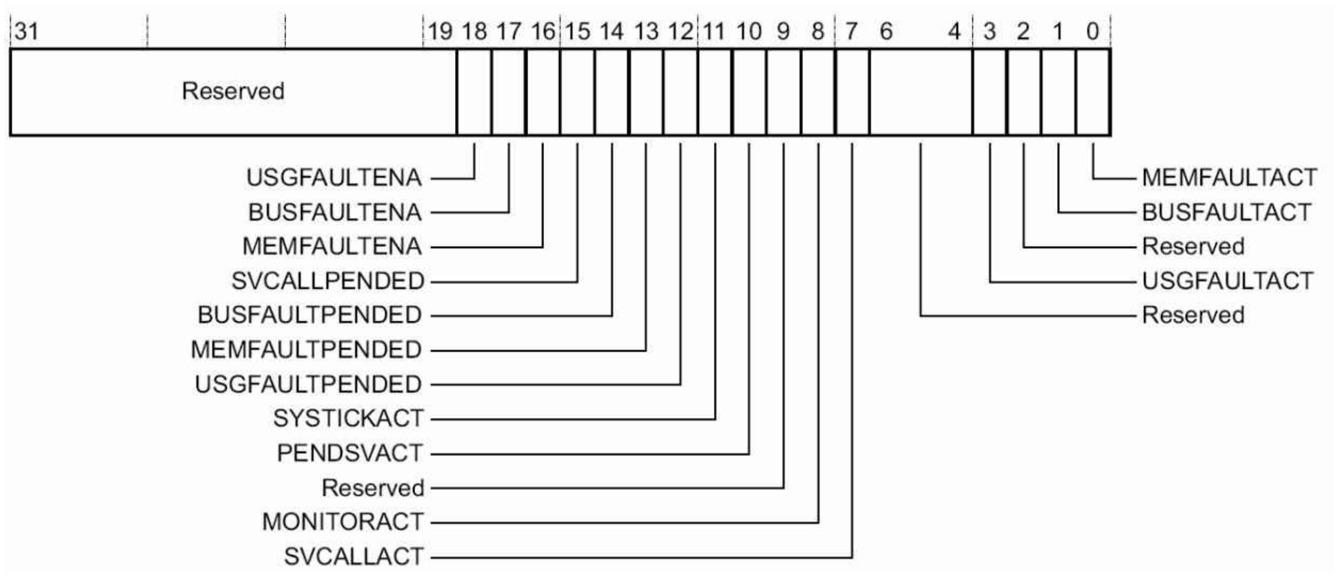


Figure 6-3. SHCSR - System Handler Control and State Register

The following bits of the SHCSR register belong to fault exceptions:

Active Bits:

- MEMFAULTACT: Memory Management Fault exception active bit, reads as 1 if exception is active.
- BUSFAULTACT: BusFault exception active bit, reads as 1 if exception is active.
- USGFAULTACT: UsageFault exception active bit, reads as 1 if exception is active.

Pending Bits:

- USGFAULTPENDED: UsageFault exception pending bit, reads as 1 if exception is pending.
- MEMFAULTPENDED: Memory Management Fault exception pending bit, reads as 1 if exception is pending.
- BUSFAULTPENDED: BusFault exception pending bit, reads as 1 if exception is pending.

Enable Bits: *(If not enabling, the exceptions below will be escelated directly to the hard fault handler)*

- MEMFAULTENA: Memory Management Fault exception enable bit, set to 1 to enable; set to 0 to disable.
- BUSFAULTENA: BusFault exception enable bit, set to 1 to enable; set to 0 to disable.
- USGFAULTENA: UsageFault exception enable bit, set to 1 to enable; set to 0 to disable.

6.2.5 Status Registers

Status Register	Handler	Address	Description
HFSR	HardFault	0xE000ED2C	HardFault Status Register
MMFSR	MemManage	0xE000ED28	MemManage Fault Status Register
BFSR	BusFault	0xE000ED29	BusFault Status Register
UFSR	UsageFault	0xE000ED2A	UsageFault Status Register
AFSR		0xE000ED3C	Auxiliary Fault Status Register. Implementation defined content

For information regarding the Status Registers bit fields, see the Arm Cortex M4 Technical Reference Manual.

6.2.5.1 Undefined Instruction Handling Example

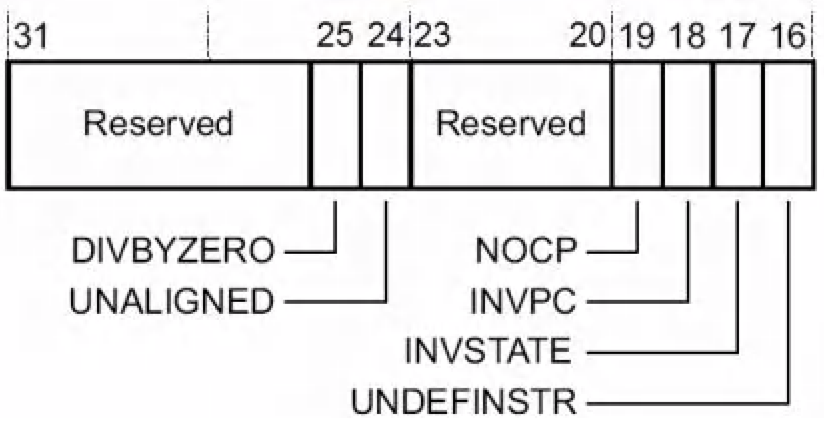
In order to detect what caused an exception, you can use the registers above inside the exception handlers themselves. Here is an example of simulating a usage fault exception inside the empty example of the MCU+ SDK and handling it with the registers above:

1. Enable all configurable exceptions via the SHCSR.
2. Create a random address in DRAM (0x00030000), with the value of 0xFFFFFFFF.
3. Create a pointer to a function that points to the address you created.
4. Execute the function by dereferencing the pointer you created.

```
void empty_main(void *args)
{
    /* Open drivers to open the UART driver for console */
    Drivers_open();
    Board_driversOpen();

    //Enable all configurable exceptions:

    uint32_t *pSHCSR = (uint32_t*)0xE000ED24;
    *pSHCSR |= ( 1<< 16); //Memory Manage Fault
    *pSHCSR |= ( 1<< 17); //Bus Fault
    *pSHCSR |= ( 1<< 18); //Usage Fault

    //Force the processor to execute an undefined instruction from DRAM
    uint32_t* pADDR = (uint32_t*)0x00030001;
    *pADDR = 0xFFFFFFFF;
    void (*temp_address) (void);
    temp_address = pADDR;
    temp_address();

    while(1);

    Board_driversClose();
    Drivers_close();
}
```

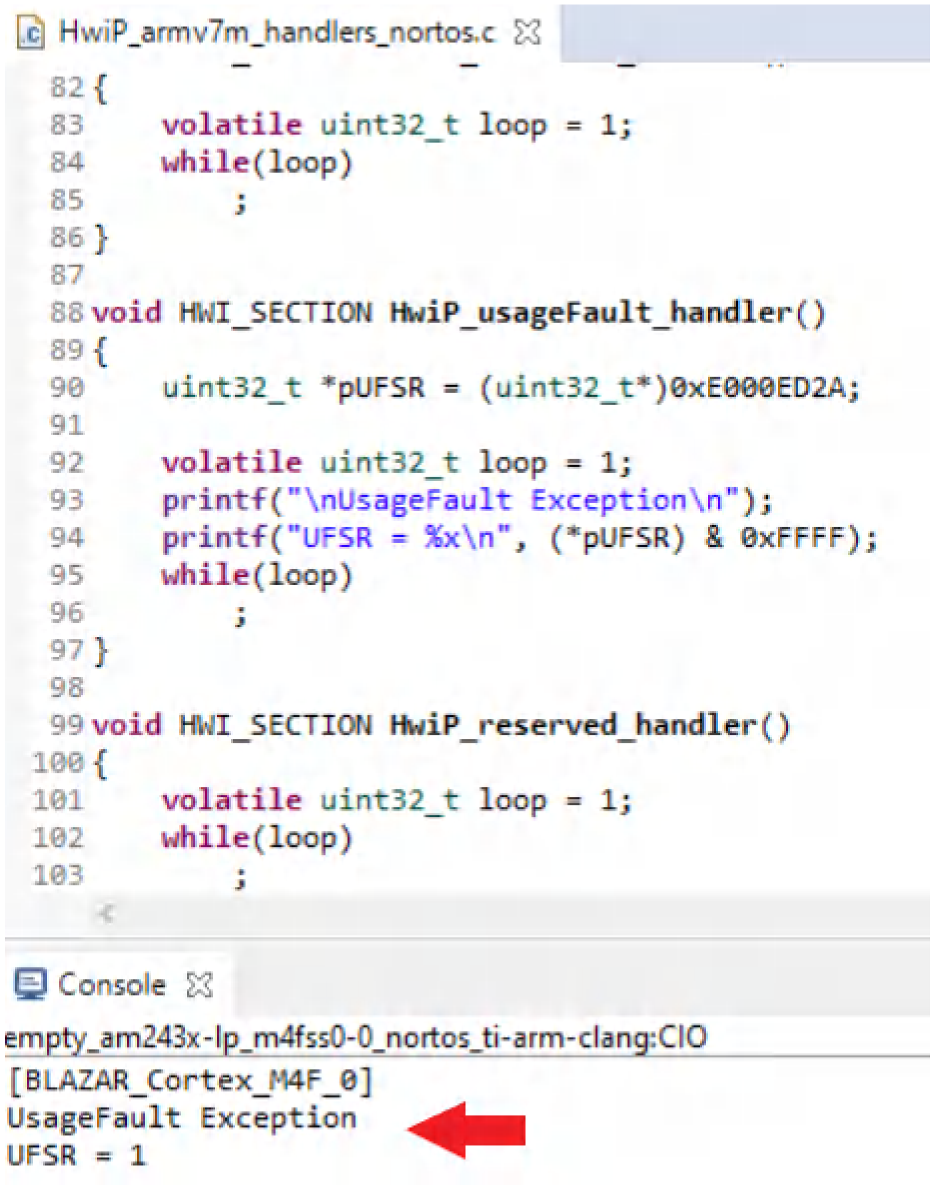
Running this code causes the processor to execute an undefined instruction and end in the UsageFault handler.

Inside the UsageFault handler (HwiP_armv7m_handlers_nortos.c), you can use the **Usage Fault Status Register (UFSR)** by reading and printing its value & 0XFFFF.



```
void HWI_SECTION HwiP_usageFault_handler()
{
    uint32_t *pUFSR = (uint32_t*)0xE000ED2A;

    volatile uint32_t loop = 1;
    printf("UsageFault Exception\n");
    printf("UFSR = %x\n", (*pUFSR) & 0xFFFF);
    while(loop)
        ;
}
```

Since an undefined instruction is executed, you can expect this register to be equal to '1':



```
{
    volatile uint32_t loop = 1;
    while(loop)
        ;
}

void HWI_SECTION HwiP_usageFault_handler()
{
    uint32_t *pUFSR = (uint32_t*)0xE000ED2A;

    volatile uint32_t loop = 1;
    printf("\nUsageFault Exception\n");
    printf("UFSR = %x\n", (*pUFSR) & 0xFFFF);
    while(loop)
        ;
}

void HWI_SECTION HwiP_reserved_handler()
{
    volatile uint32_t loop = 1;
    while(loop)
        ;
```

```
empty_am243x-lp_m4fss0-0_nortos_ti-arm-clang:CIO
[BLAZAR_Cortex_M4F_0]
UsageFault Exception
UFSR = 1
```

6.2.5.2 Invalid State Handling Example

You can also simulate a different UsageFault exception of invalid state **(UFSR** = 0x2). This fault means that the processor has attempted to execute an instruction that makes illegal use of the Execution Program Status Register (EPSR).

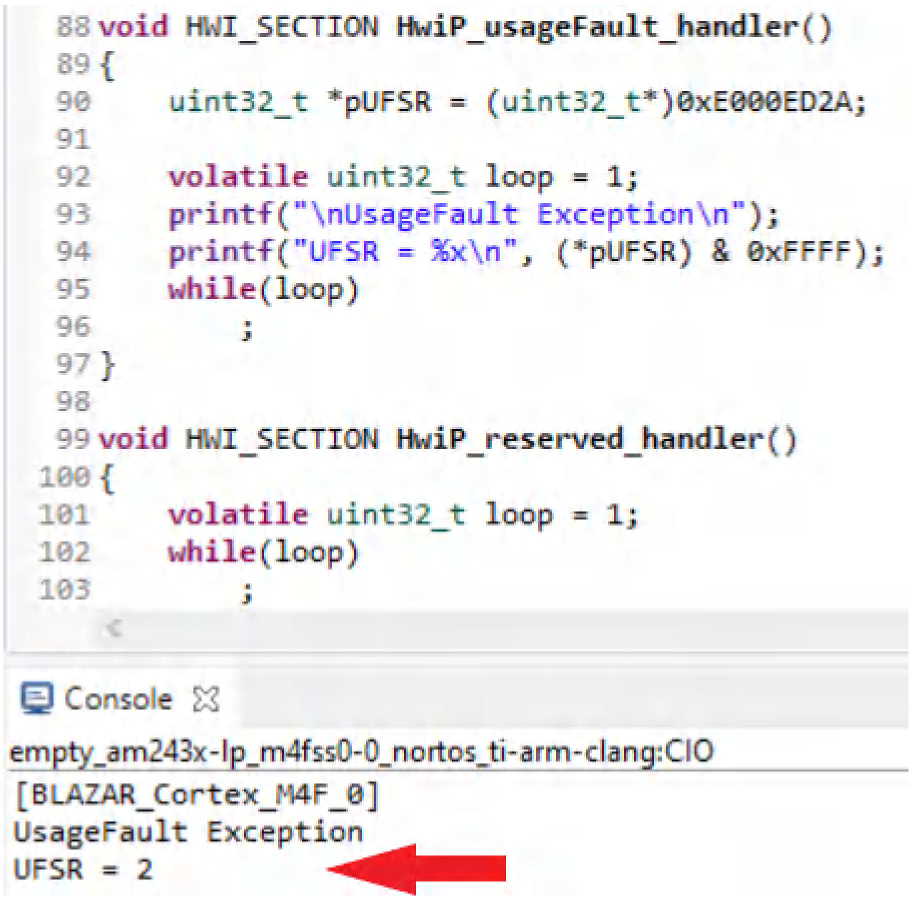
When the **INVSTATE** bit of the **UFSR** is set, the PC value stacked for the exception return points to the instruction that attempted the illegal use of the EPSR. Potential reasons:

- Loading a branch target address to PC with LSB=0
- Stacked PSR corrupted during exception or interrupt handling
- Vector table contains a vector address with LSB=0

The LSB of the address is loaded to the 'T' bit of the **EPSR**. If this bit is set, it means that the processor is in Thumb mode, if this bit is cleared, it means that the processor is in Arm mode. Since Arm Cortex M Series support only Thumb ISA, then this bit must be set during execution. This is something that is done by the compiler, but if you assign an address manually to the PC, then you need to take care of this bit by ourselves.

The code below causes the processor to halt in the UsageFault handler, but this time with UFSR = 0x2 (Invalid State):

```
void empty_main(void *args)
{
    /* Open drivers to open the UART driver for console */
    Drivers_open();
    Board_driversOpen();

    //Enable all configurable exceptions:

    uint32_t *pSHCSR = (uint32_t*)0xE000ED24;
    *pSHCSR |= ( 1<< 16); //Memory Manage Fault
    *pSHCSR |= ( 1<< 17); //Bus Fault
    *pSHCSR |= ( 1<< 18); //Usage Fault

    //Address LSB = 0
    uint32_t* pADDR = (uint32_t*)0x00030000;
    *pADDR = 0xFFFFFFFF;
    void (*temp_address) (void);
    temp_address = pADDR;
    temp_address();

    while(1);

    Board_driversClose();
    Drivers_close();
}
```



```
void HWI_SECTION HwiP_usageFault_handler()
{
    uint32_t *pUFSR = (uint32_t*)0xE000ED2A;

    volatile uint32_t loop = 1;
    printf("\nUsageFault Exception\n");
    printf("UFSR = %x\n", (*pUFSR) & 0xFFFF);
    while(loop)
        ;
}

void HWI_SECTION HwiP_reserved_handler()
{
    volatile uint32_t loop = 1;
    while(loop)
        ;
```

```
empty_am243x-lp_m4fss0-0_nortos_ti-arm-clang:CIO
[BLAZAR_Cortex_M4F_0]
UsageFault Exception
UFSR = 2
```

6.2.6 Printing the Stack Frame

As mentioned in the previous sections, when the processor handles an exception, during the entry sequence, it stacks *R0 – R3, R12, LR, PC, and xPSR.*

When debugging outside of CCS, printing the stack frame to the console might be very useful in order to explore why you end up in an exception. In example, knowing the value of the Link Register (R14) is important to understand where in memory the exception occurred. To print the Stack Frame, you have to know the value of the Main Stack Pointer (MSP) right when the exception happened, so you will be able to print the whole block that contains R0 – R3, R12, LR, PC, and xPSR.

When an exception occurs:

1. Go to an Handler code
2. Save the MSP
3. Print R0 - R3, R12, LR, PC and xPSR (MSP[0] to MSP[7])

In our SDK, exceptions handlers are written in C code. For every C function, the compiler creates a prologue and epilogue sequences that manipulates the MSP register. Thus, if you save the MSP inside the C function, a wrong MSP value is stored. To avoid that, you must use the "naked" attribute and write a naked function in assembly. This attribute tells the compiler that the function is an embedded assembly function, and then prologue and epilogue sequences is not generated for that function by the compiler, and the MSP points to the right value.

Following the above, the sequence of handling a fault is as follows:

1. Once an exception occurs, execute the **naked** handler function (assembly):
   a. Save MSP in R0
   b. Call the C version of the handler code
2. Inside the C function:
   a. Print R0 - R3, R12, LR, PC and xPSR (MSP[0] to MSP[7])
   b. while(1)

Here is a code example for the modified usageFault_handler() function of the SDK:

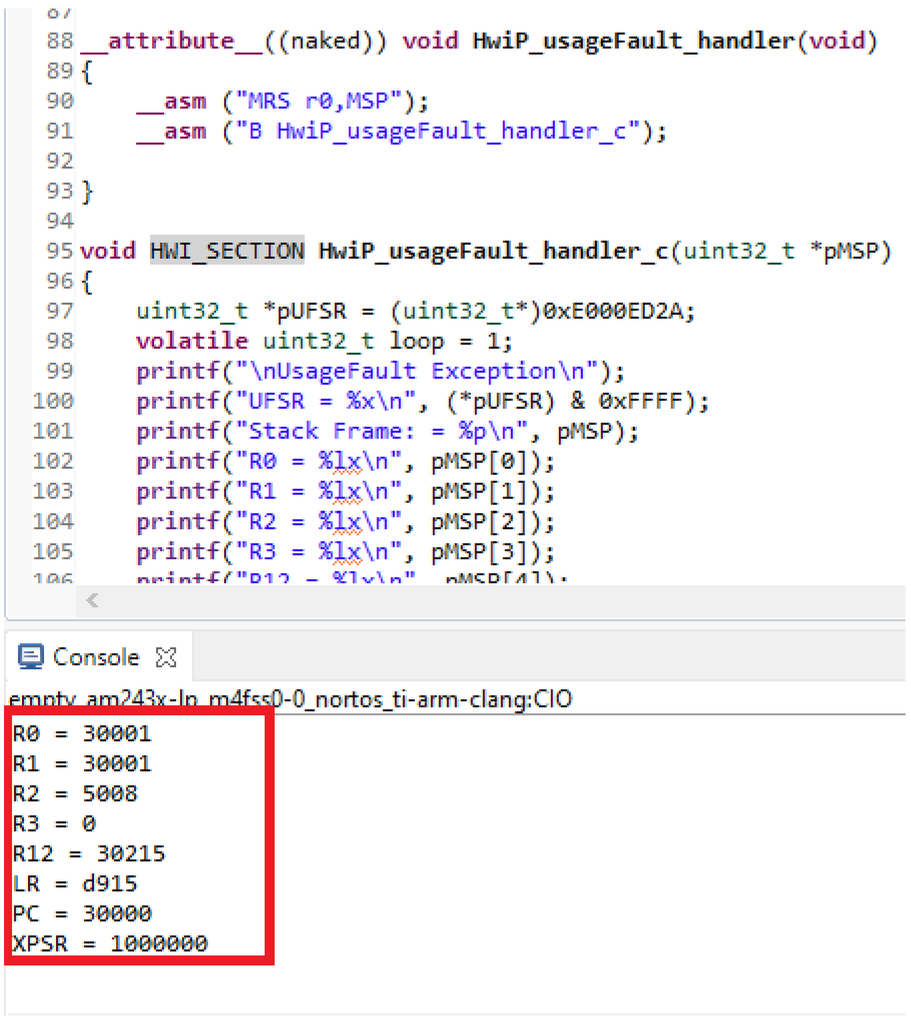
```
__attribute__((naked)) void HwiP_usageFault_handler(void)
{
    __asm ("MRS r0,MSP");
    __asm ("B HwiP_usageFault_handler_c");
}

void HWI_SECTION HwiP_usageFault_handler_c(uint32_t *pMSP)
{
    uint32_t *pUFSR = (uint32_t*)0xE000ED2A;
    volatile uint32_t loop = 1;
    printf("\nUsageFault Exception\n");
    printf("UFSR = %x\n", (*pUFSR) & 0xFFFF);
    printf("Stack Frame: = %p\n", pMSP);
    printf("R0 = %lx\n", pMSP[0]);
    printf("R1 = %lx\n", pMSP[1]);
    printf("R2 = %lx\n", pMSP[2]);
    printf("R3 = %lx\n", pMSP[3]);
    printf("R12 = %lx\n", pMSP[4]);
    printf("LR = %lx\n", pMSP[5]);
    printf("PC = %lx\n", pMSP[6]);
    printf("xPSR = %lx\n", pMSP[7]);

    while(loop)
        ;
}
```

Example

In order to simulate an usageFault, you can use the previous example where you were trying to execute an undefined instruction from DRAM:

```
void empty_main(void *args)
{
    /* Open drivers to open the UART driver for console */
    Drivers_open();
    Board_driversOpen();

    //Enable all configurable exceptions:

    uint32_t *pSHCSR = (uint32_t*)0xE000ED24;
    *pSHCSR |= ( 1<< 16); //Memory Manage Fault
    *pSHCSR |= ( 1<< 17); //Bus Fault
    *pSHCSR |= ( 1<< 18); //Usage Fault

    uint32_t* pADDR = (uint32_t*)0x00030001;
    *pADDR = 0xFFFFFFFF;
    void (*temp_address) (void);
    temp_address = pADDR;
    temp_address();

    while(1);

    Board_driversClose();
    Drivers_close();
}
```

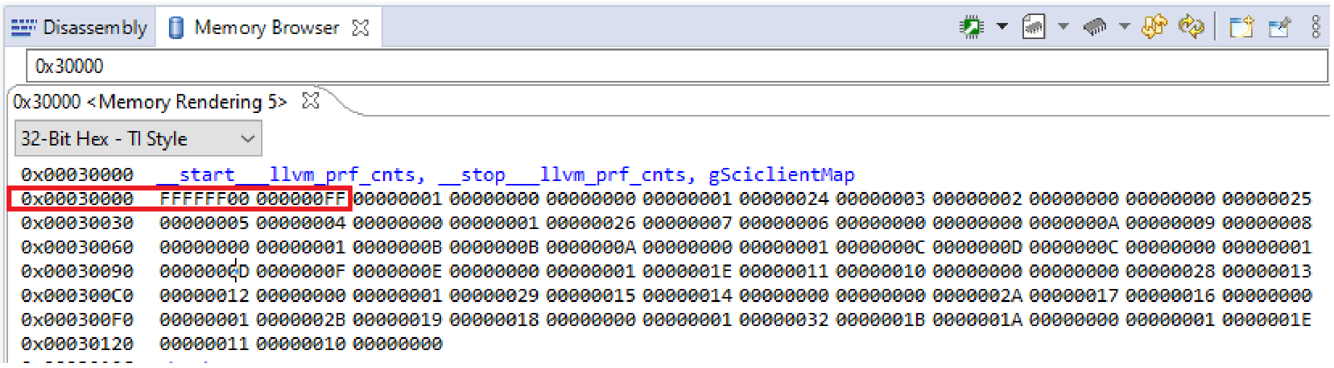
Once you execute the code above, you will go through the **naked** usageFault_handler (assembly version) and then jump to the C version and print the whole stack frame:



```
__attribute__((naked)) void HwiP_usageFault_handler(void)
{
    __asm ("MRS r0,MSP");
    __asm ("B HwiP_usageFault_handler_c");

}

void HWI_SECTION HwiP_usageFault_handler_c(uint32_t *pMSP)
{
    uint32_t *pUFSR = (uint32_t*)0xE000ED2A;
    volatile uint32_t loop = 1;
    printf("\nUsageFault Exception\n");
    printf("UFSR = %x\n", (*pUFSR) & 0xFFFF);
    printf("Stack Frame: = %p\n", pMSP);
    printf("R0 = %lx\n", pMSP[0]);
    printf("R1 = %lx\n", pMSP[1]);
    printf("R2 = %lx\n", pMSP[2]);
    printf("R3 = %lx\n", pMSP[3]);
```

```
empty_am243x-lp_m4fss0-0_nortos_ti-arm-clang:CIO
R0 = 30001
R1 = 30001
R2 = 5008
R3 = 0
R12 = 30215
LR = d915
PC = 30000
XPSR = 1000000
```

In the image above, you can see that **LR = 0x30000,** by searching this address in the Disassembly window. You can see the undefined instruction (**0xFFFFFFFF)** that you were trying to execute:



```
0x00030000   __start___llvm_prf_cnts, __stop___llvm_prf_cnts, gSciclientMap
0x00030000   FFFFFF00 000000FF 00000001 00000000 00000000 00000001 00000024 00000003 00000002 00000000 00000000 00000025
0x00030030   00000005 00000004 00000000 00000001 00000026 00000007 00000006 00000000 00000000 0000000A 00000009 00000008
0x00030060   00000000 00000001 0000000B 0000000B 0000000A 00000000 00000001 0000000C 0000000D 0000000C 00000000 00000001
0x00030090   0000000D 0000000F 0000000E 00000000 00000001 0000001E 00000011 00000010 00000000 00000000 00000028 00000013
0x000300C0   00000012 00000000 00000001 00000029 00000015 00000014 00000000 00000000 0000002A 00000017 00000016 00000000
0x000300F0   00000001 0000002B 00000019 00000018 00000000 00000001 00000032 0000001B 0000001A 00000000 00000001 0000001E
0x00030120   00000011 00000010 00000000
```

7 Debugging Memory

7.1 Viewing Device Memory

To open the Memory Browser view: View → Memory Browser

The Memory Browser view is another integral part of the debugger that shows the contents of the target memory starting at a specified address and with various data formatting features.

To use it, enter an address in the Address Text box at the top of the view. The contents of a selected memory location can be edited by double-clicking the value.

It features multiple viewing formats and types:

Chars, integers (signed/unsigned), floats, multiple hexadecimal data sizes (8 through 64-bits).

Additional capabilities include:

- Memory fill with predefined values
- Saving to and loading from files on the host PC
- Viewing of all variables and functions
- A context-sensitive information box for every memory position

7.2 Linker Command File (linker.cmd)

The linker, not the compiler, defines the memory map and allocates code and data into target memory. The linker command (linker.cmd) is the code generation development tool responsible for linking together all the object files and libraries into the final executable form. The linker offers many features, including some recent additions, which make it easy to use system memory efficiently. Inside the linker there are two important directives:

- The MEMORY directive
- The SECTIONS directive

Those directives appear in every linker command file.

7.2.1 The Memory Directive

The purpose of the MEMORY directive is to assign names to ranges of memory. These memory range names are used in the SECTIONS directive. Here is part of the MEMORY directive from a typical Am243x system:

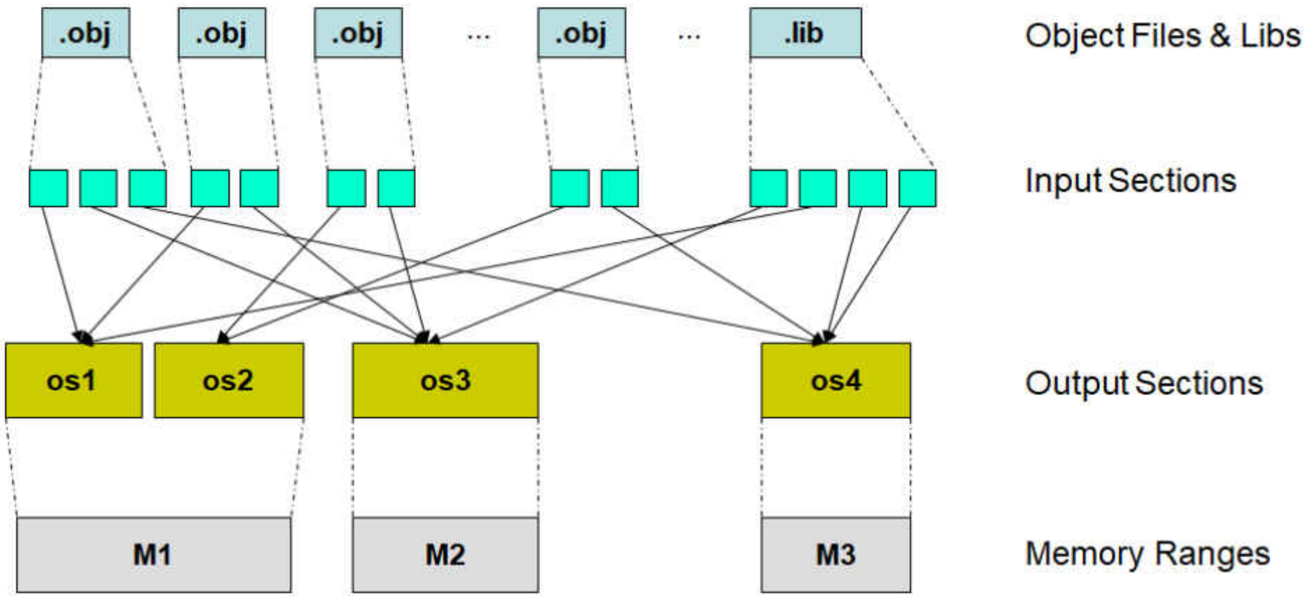
```
MEMORY
{
    R5F_VECS                : ORIGIN = 0x00000000 , LENGTH = 0x00000040
    R5F_TCMA                : ORIGIN = 0x00000040 , LENGTH = 0x00007FC0
    R5F_TCMB0               : ORIGIN = 0x41010000 , LENGTH = 0x00008000
    NON_CACHE_MEM           : ORIGIN = 0x70060000 , LENGTH = 0x8000
    MSRAM                   : ORIGIN = 0x70080000 , LENGTH = 0x40000
    FLASH                   : ORIGIN = 0x60100000 , LENGTH = 0x80000
    USER_SHM_MEM            : ORIGIN = 0x701D0000, LENGTH = 0x180
    LOG_SHM_MEM             : ORIGIN = 0x701D0000 + 0x180, LENGTH = 0x00004000 - 0x180
    RTOS_NORTOS_IPC_SHM_MEM : ORIGIN = 0x701D4000, LENGTH = 0x0000C000
}
```

The line that begins with MSRAM defines a memory range named MSRAM. It starts at address 0x7008000 and has a length of 0x4000.

7.2.2 The Sections Directive

The SECTIONS directive contains most of the interesting code. The key thing to understand is that the SECTIONS directives does two things at once:

- It forms output sections from input sections
- It allocates those output sections to memory



Describing the SECTIONS directive requires an understanding of these terms:

- Object file - For the purposes of this document, an object file is a collection of input sections. An object file can be presented directly to the linker (via the command line or in a command file), or it can come from a library.
- Input Section - One section from one object file. An input section can be initialized or uninitialized. It can contain code or data.
- Output Section - A collection of one or more input sections
- Memory Range - A range of system memory specified in the MEMORY directive

In theory, you cannot know anything about the contents of an input section based on the name alone. Nonetheless, input sections with these names usually have these contents:

Name	Initialized	Notes
.text	Yes	Executable code
.bss	No	Global variables
.cinit	Yes	Tables which initialize global variables
.data (EABI)	Yes and No	Initialized coming out of the assembler; changed to uninitialized by the linker
.data (COFF ABI)	Yes	Initialized data
.stack	No	System stack
.sysmem or .heap	No	Malloc heap (malloc(), calloc(), realloc()..)
.const	Yes	Initialized global variables
.switch	Yes	Jump tables for certain switch statements
.init_array or .pinit	Yes	Table of C++ constructors called at startup
.cio	No	Buffer for stdio functions
.rodata	Yes	Contains read-only data, typically string constants and static-scoped objects.

Here is a part of the sections directive of the Am243x:

```
SECTIONS
{
    .vectors:{} palign(8) > R5F_VECS
    GROUP {
        .text.hwi: palign(8)
        .text.cache: palign(8)
        .text.mpu: palign(8)
        .text.boot: palign(8)
        .text:abort: palign(8) /* this helps in loading symbols when using XIP mode */
    } > MSRAM
    GROUP {
        .text:   {} palign(8)   /* This is where code resides */
        .rodata: {} palign(8)   /* This is where const's go */
    } > MSRAM
```

Grouping and aligning is done quite often inside linker.cmd file. The assembler generally has a .align directive embedded in code, but doing **palign(8)** in linker ensures proper alignment. **palign(8)** is not necessary for code sections, code is typically aligned to 4 byte boundary so that Arm instructions are aligned correctly. It is necessary for sections like the heap and stacks. Other data sections are 8-byte aligned to ensure max data size of **long long** (64 bits) are aligned. Note that **palign** only ensures section start and size are 8 byte aligned, it doesn't ensure individual members in sections are aligned.

Grouping is done for various reasons, sections like BSS need grouping so that a symbol for start and end get defined, which is used in C runtime init phase before **main** to memset BSS section to 0.

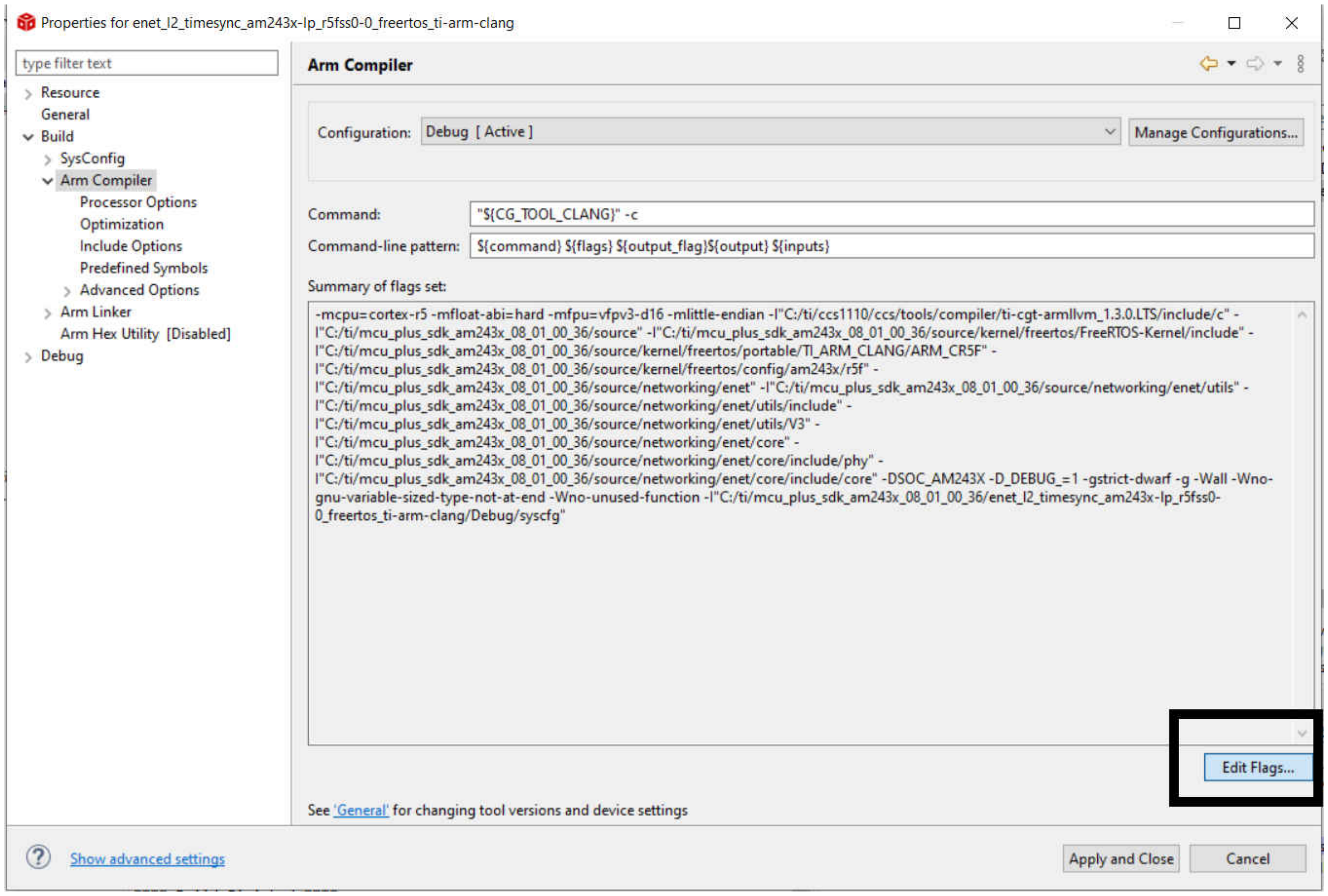
Some sections are grouped to enable overlay with other specific sections. For example, ICSS firmware is needed only at initialization and is overlaid with ICSS_MEM section as ICSS_MEM section is used only after firmware is loaded, after which the firmware is no longer required and can be overwritten.

Grouping is generally done as it is easy to check in map file the size of section if is grouped.

In order to understand more of the syntax inside linker.cmd, see the TI Linker Command File Primer

7.3 Stack Overflow

There are few methods to debug the stack. One of them is using the stack peak of the ROV (described in detailed in the ROV section). Another method that can be useful is to set a watchpoint to the end of stack, as described in the Watchpoints section. In addition, the compiler provides stack protection functionality in the form of the following options. To be able to use them, enable those "flags" in **Project** → **Properties** → **Build** → **Arm Compiler** → **Edit flags**.



Configuration: Debug [Active]

Command: "${CG_TOOL_CLANG}" -c

Command-line pattern: ${command} ${flags} ${output_flag}${output} ${inputs}

Summary of flags set:

```
-mcpu=cortex-r5 -mfloat-abi=hard -mfpu=vfpv3-d16 -mlittle-endian -I"C:/ti/ccs1110/ccs/tools/compiler/ti-cgt-armllvm_1.3.0.LTS/include/c" -
I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source" -I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source/kernel/freertos/FreeRTOS-Kernel/include" -
I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source/kernel/freertos/portable/TI_ARM_CLANG/ARM_CR5F" -
I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source/kernel/freertos/config/am243x/r5f" -
I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source/networking/enet" -I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source/networking/enet/utils" -
I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source/networking/enet/utils/include" -
I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source/networking/enet/utils/V3" -
I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source/networking/enet/core" -
I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source/networking/enet/core/include/phy" -
I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/source/networking/enet/core/include/core" -DSOC_AM243X -D_DEBUG_=1 -gstrict-dwarf -g -Wall -Wno-
gnu-variable-sized-type-not-at-end -Wno-unused-function -I"C:/ti/mcu_plus_sdk_am243x_08_01_00_36/enet_l2_timesync_am243x-lp_r5fss0-
0_freertos_ti-arm-clang/Debug/syscfg"
```

See 'General' for changing tool versions and device settings

7.3.1 -fstack-protector

By adding the flag *-fstack-protector*, you instruct the compiler to emit extra code to check for buffer overflows, such as stack-smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions with buffers larger than or equal to 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error handling function is called. The error handling function can be made to indicate the error in some way and exit the program. Only variables that are actually allocated on the stack are considered, optimized away variables or variables allocated in registers are not considered.

7.3.2 -fstack-protector-strong

By adding the flag *-fstack-protector-strong* , you instruct the compiler to behave as if -fstack-protector were specified, except that a stronger heuristic is used to determine the functions for which the compiler will emit stack buffer overflow checking code.

7.3.3 -fstack-protector-all

By adding the flag *-fstack-protector-all*, you instruct the compiler to behave as if -fstack-protector were specified, except that the compiler emits stack buffer overflow checking code for all functions instead of limiting protection as -fstack-protector does.

7.3.4 Enabling Stack Smashing Detection

To enable stack smashing detection in your application, you need to provide definitions of:

__stack_chk_fail() - This function is called from an instrumented function when a check against the stack guard value, *__stack_chk_guard*, fails. A simple definition of this function might look like this:

```
void __stack_chk_fail(void) {
printf("__stack_chk_guard has been corrupted\n");
exit(0);
}
```

__stack_chk_guard - This is a globally visible symbol whose value can be copied into a location at the boundary of a function's allocated stack on entry into the function, and loaded just prior to function exit to perform a check that the local copy of the *__stack_chk_guard* value has not been overwritten. A simple definition of this symbol might look like this:

```
unsignedlong__stack_chk_guard=0xbadeebad;
```

You can then compile a file containing both of these definitions to preoduce an object file that can be linked into an application that is instrumented for stack smashing detection.

7.3.5 Enabling Stack Smashing Detection

Here is a simple example to summarize and demonstrate how the stack smashing detection capability can be used. The first source file presents the definitions of *__stack_chk_fail()* and *__stack_chk_guard* (stack_check.c):

```
#include <stdlib.h>
#include <stdio.h>
void __stack_chk_fail(void);
unsigned long __stack_chk_guard = 0xbadeebad;
void __stack_chk_fail(void)
{
printf("ERROR: __stack_chk_guard has been corrupted\n");
exit(0);
}
```

The second source file presents a use case where a function, *foo*, writes past the end of a local buffer (stack_smash.c):

```
#include <string.h>
void foo(void);
int main() {
foo();
return 0;}void foo(void)
{
char buffer[3];
strcpy(buffer, "Oi! I am smashing your stack");
}
```

7.4 Variables and Expressions View in CCS

To open the Variables view: *View → Variables*

The Variables view shows only **Local** variables that belong to the function currently being executed.

To see an example of this, import gpio_interrupt example and set a breakpoint inside **gpio_interrupt.c** in **gpio_input_interrupt_main()** and in the ISR **GPIO_bankIsrFxn()**.

Run the program and let it halt at the first breakpoint in **gpio_input_interrupt_main()**. Take a look a the variables listed in the Variables view - notice it shows the variables local to this function. You can modify the value of a variable directly from the Variables view. Variables whose values have changed since the last time they were seen are highlighted in yellow.

To open the Expressions view: *View → Expressions*

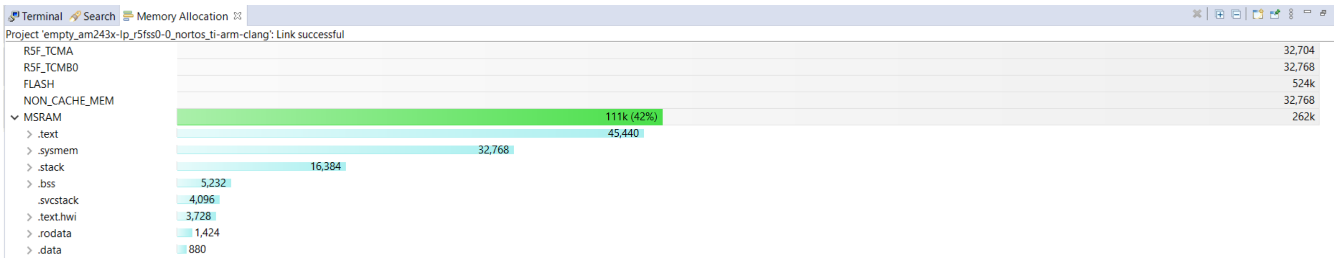
The Expressions view allows you to watch **local, global,** and **static variables, C-valid expressions,** and **registers**.

Expressions whose values have changed since the last time they were seen are highlighted in yellow. You can modify the value of an expression directly from the Expressions view.

7.5 Understanding Your Application's Memory Allocation

The Memory Allocation View in Code Composer Studio provides a graphical representation of how much memory is consumed by your application.

To open the view, go to the CCS menu View -> Memory Allocation.



Project 'empty_am243x-lp_r5fss0-0_nortos_ti-arm-clang': Link successful

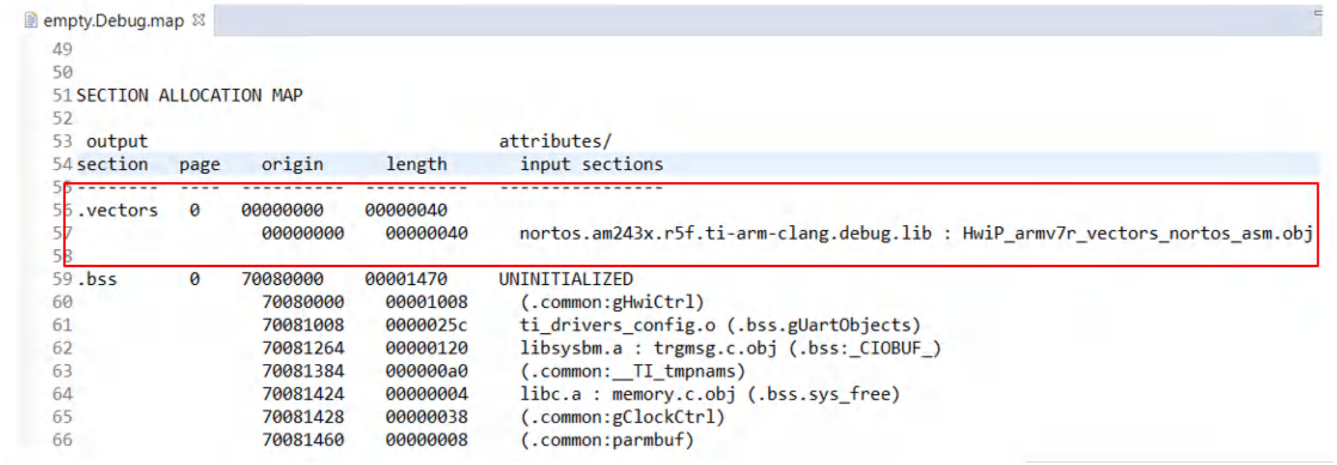
R5F_TCMA		32,704
R5F_TCMB0		32,768
FLASH		524k
NON_CACHE_MEM		32,768
MSRAM	111k (42%)	262k
.text	45,440	
.sysmem	32,768	
.stack	16,384	
.bss	5,232	
.svcstack	4,096	
.text.hwi	3,728	
.rodata	1,424	
.data	880	

By default, the view shows the memory used relative to the total available memory for the project that is active in the Project Explorer view. You can expand each memory region to see how much memory each individual section or sub-section is using. Another method to debug memory allocation to understand where the memory sections you configured in linker.cmd are actually allocated, is to look at the *Debug.map* file under Debugà*"project_name ".Debug.map.*

For example, in the Am243x linker.cmd file, there is a default section called "vectors":

```
SECTIONS
{
.vectors:{} palign(8) > R5F_VECS
...
...
...
}
```

To understand where exactly this section is allocated and what it contains, check the *Debug.map* file.



```


SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.vectors    0    00000000    00000040
                  00000000    00000040     nortos.am243x.r5f.ti-arm-clang.debug.lib : HwiP_armv7r_vectors_nortos_asm.obj

.bss        0    70080000    00001470     UNINITIALIZED
                  70080000    00001008     (.common:gHwiCtrl)
                  70081008    0000025c     ti_drivers_config.o (.bss.gUartObjects)
                  70081264    00000120     libsysbm.a : trgmsg.c.obj (.bss:_CIOBUF_)
                  70081384    000000a0     (.common:__TI_tmpnams)
                  70081424    00000004     libc.a : memory.c.obj (.bss.sys_free)
                  70081428    00000038     (.common:gClockCtrl)
                  70081460    00000008     (.common:parmbuf)
```

There, this is a specific case, the vectors are located in address 0x00, and contains an object called "*HwiP_armv7r_vectors_nortos_asm.obj*" from a library named "*nortos.am243x.r5f.ti-arm-clang.debug.lib*".

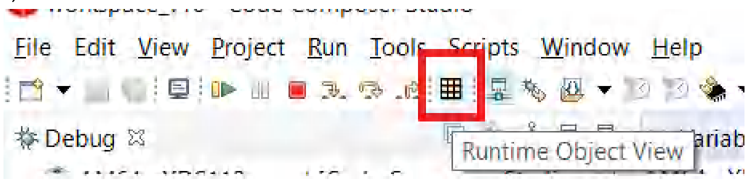
7.6 FreeRTOS ROV

The Runtime Object View (ROV) provides tools that enable developers to quickly visualize the state of embedded applications. ROV reads memory from the target and intelligently displays data.

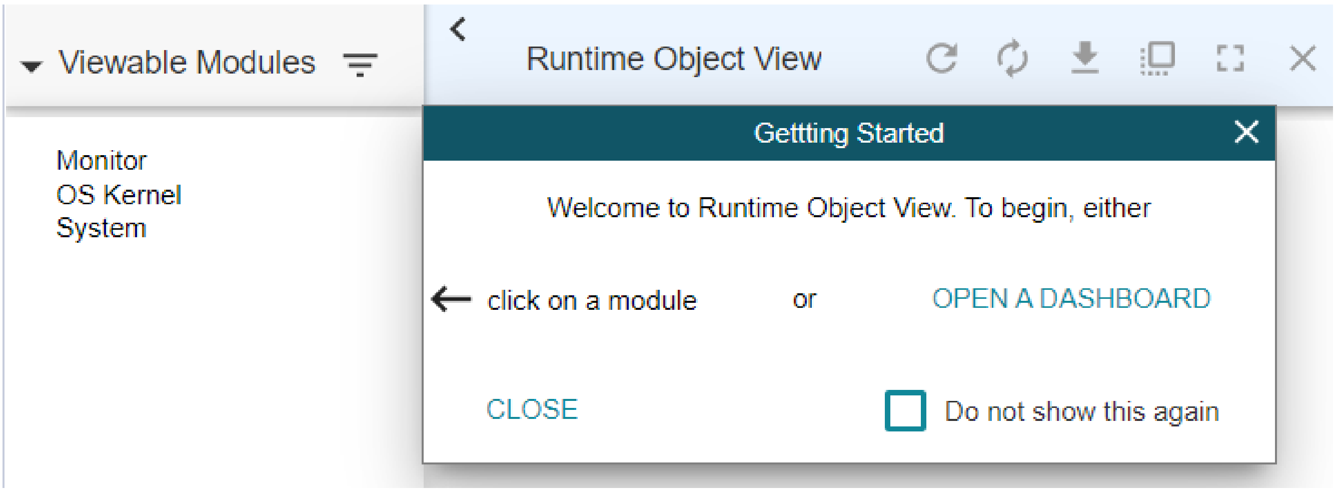
- ROV does not disturb the run-time behavior of the application on the target (when using a JTAG connection). ROV can read current memory from a running target. ROV also automatically refreshes all of its views whenever the target is stopped–for example when single stepping, when the target hits a breakpoint, or when you halt the target asynchronously.
- ROV adds zero footprint to the target code (when using a JTAG connection).
- ROV provides visual tools that show changes in the target state, memory use, and data structures on the host computer. The tool shows high-level information needed by embedded application developers.
- ROV is provided automatically with the TI-RTOS Kernel component of the Sitara SDK. If you use Code Composer Studio (CCS) you have an access to ROV. Nothing needs to be enabled in the application code.

ROV can be extended to work with any embedded library. The examples on this page use the TI-RTOS Kernel, which includes ROV support for its entities including threads, heap memory, and CPU load.

1. To launch ROV views, click on the “ROV” button in the CCS toolbar as shown below:



2. Click on various “Viewable Modules”. “OS Kernel” is the one which has useful views:


Viewable Modules

Monitor
OS Kernel
System

Runtime Object View

Gettting Started

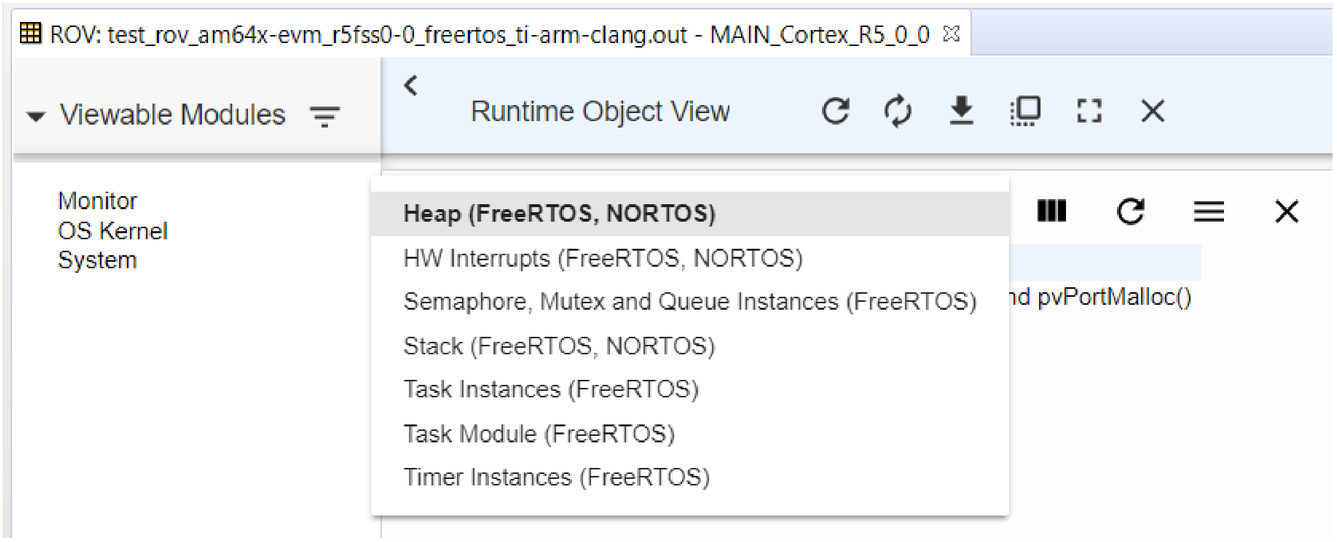
Welcome to Runtime Object View. To begin, either

click on a module or OPEN A DASHBOARD

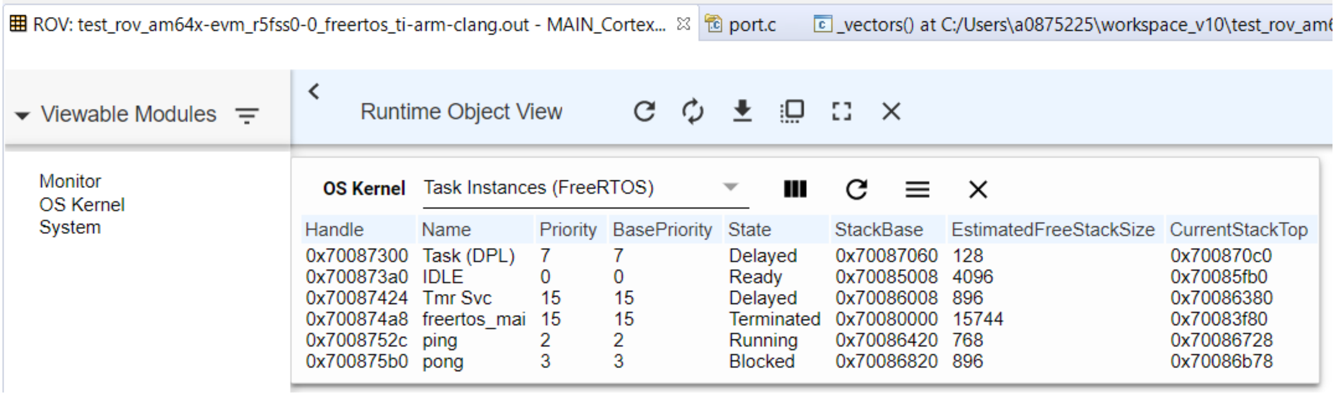
CLOSE

Do not show this again

3. After clicking on "OS Kernel", click on the drop down to see all the supported ROV views:


ROV: test_rov_am64x-evm_r5fss0-0_freertos_ti-arm-clang.out - MAIN_Cortex_R5_0_0
Viewable Modules
Runtime Object View
Monitor
OS Kernel
System
Heap (FreeRTOS, NORTOS)
HW Interrupts (FreeRTOS, NORTOS)
Semaphore, Mutex and Queue Instances (FreeRTOS)
Stack (FreeRTOS, NORTOS)
Task Instances (FreeRTOS)
Task Module (FreeRTOS)
Timer Instances (FreeRTOS)
nd pvPortMalloc()

4. Shown below is a sample after "Task Instances" view is selected.


ROV: test_rov_am64x-evm_r5fss0-0_freertos_ti-arm-clang.out - MAIN_Cortex...
port.c
_vectors() at C:/Users\a0875225\workspace_v10\test_rov_am6
Viewable Modules
Runtime Object View
Monitor
OS Kernel
System
OS Kernel
Task Instances (FreeRTOS)

Handle	Name	Priority	BasePriority	State	StackBase	EstimatedFreeStackSize	CurrentStackTop
0x70087300	Task (DPL)	7	7	Delayed	0x70087060	128	0x700870c0
0x700873a0	IDLE	0	0	Ready	0x70085008	4096	0x70085fb0
0x70087424	Tmr Svc	15	15	Delayed	0x70086008	896	0x70086380
0x700874a8	freertos_mai	15	15	Terminated	0x70080000	15744	0x70083f80
0x7008752c	ping	2	2	Running	0x70086420	768	0x70086728
0x700875b0	pong	3	3	Blocked	0x70086820	896	0x70086b78

The ROV can be very helpful for debugging stack overflow issues. When an error such as a stack overflow occurs, the corresponding field is highlighted in red. Hover your cursor over a red field to see a brief description of the error.

8 Debugging Boot

Booting user-defined applications on a System-on-Chip (SoC) involves multiples steps as listed below:

1. There are multiple steps involved to convert a user application, created using a compiler+linker toolchain, into a binary format that is suitable to be booted by the SoC.
2. You need to flash this binary to the EVM flash.
3. When the SoC is powered on, the previously flashed binary is executed.
4. After powering on the EVM, the bootflow takes place mainly in two steps:
   a. **ROM boot**, in which the ROM bootloader boots a secondary bootloader or an SBL.
   b. **SBL boot** in which the secondary bootloader boots the application.

8.1 ROM Boot

- As soon as the EVM is powered ON, the ROM bootloader or RBL starts running. The RBL is the primary bootloader.
- Depending on which boot mode is selected on the EVM, the RBL loads the **secondary bootloader** or SBL from a boot media (OSPI flash, SD card or via UART).
- Rest of the booting is done by the SBL
- The RBL expects the image it boots (SBL in our case) to always be signed

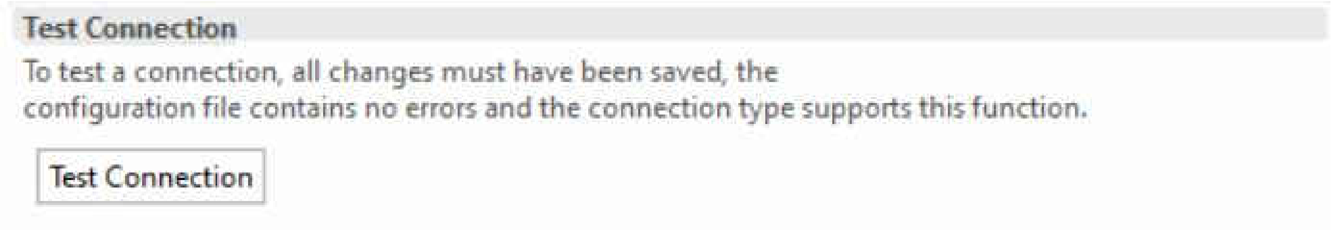
8.2 SBL Boot

- The SBL is essentially an example application of the bootloader library.
- It is considered to be a secondary bootloader because it is booted by the RBL, which is the primary bootloader.
- An SBL typically does a bunch of specific initializations and proceeds to the application loading.
  – For example, in the case of AM243x, the SBL loads the SYSFW to the Cortex M3 and sends the board cfg to the SYSFW once the M3 core is booted.
- Depending on the type of SBL loaded, SBL looks for the **multicore appimage** of the application binary at a specified location in a boot media.
- If the appimage is found, the multicore appimage is parsed into multiple **RPRCs**. These are optimized binaries that are then loaded into individual CPUs.
- Each RPRC image has information regarding the core on which it is to be loaded, entry points and multiple sections of that application binary
- The SBL uses this information to initialize each core that has a valid RPRC. It then loads the RPRC according to the sections specified, sets the entry points and releases the core from reset. Now the core will start running.

8.3 GEL Files

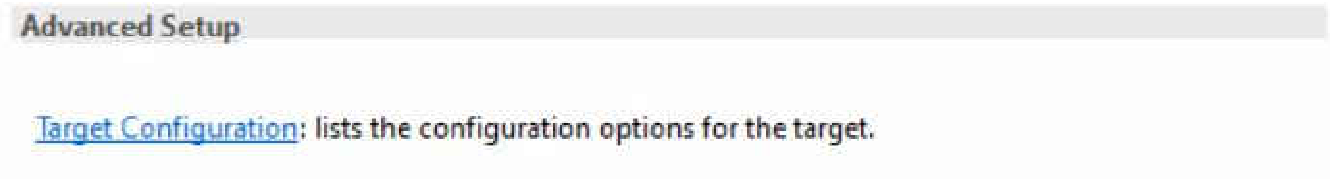
Startup GEL files are used to automate device initialization when Code Composer Studio starts up. The General Extension Language (GEL) can be used to configure the Code Composer Studio development environment and to initialize the target CPU. GEL is an interpreted language, and its syntax is similar to that of C. A rich set of built-in GEL functions is available, or you can create your own GEL functions. When develop and debugging with CCS, the GEL files replace the functionality of the Secondary Bootloader. In the field, the booting sequence happens as described above, and most times the code will be imported from Flash to RAM by the SBL. The SBL is a C code which is part of the Sitara MCU SDK and it can be modified / implemented by the developer. If important parts are missing from the SBL (such as un-itiliaziations of clocks and so on), unexpected problems can occur. In those cases, where the booting process fails, it is difficult to understand where in the code you are stuck. In order to debug such cases, it can be easier to connect to target without GEL files or loading any image.

1. Configure a new target by **new->Target Configuration file**, insert a file name and click on **finish.** Then look for your device (Am243x Launchpad was used) and click on **save**.
2. You can click on **Test Connection** button to verify that all connection tests pass.


Test Connection

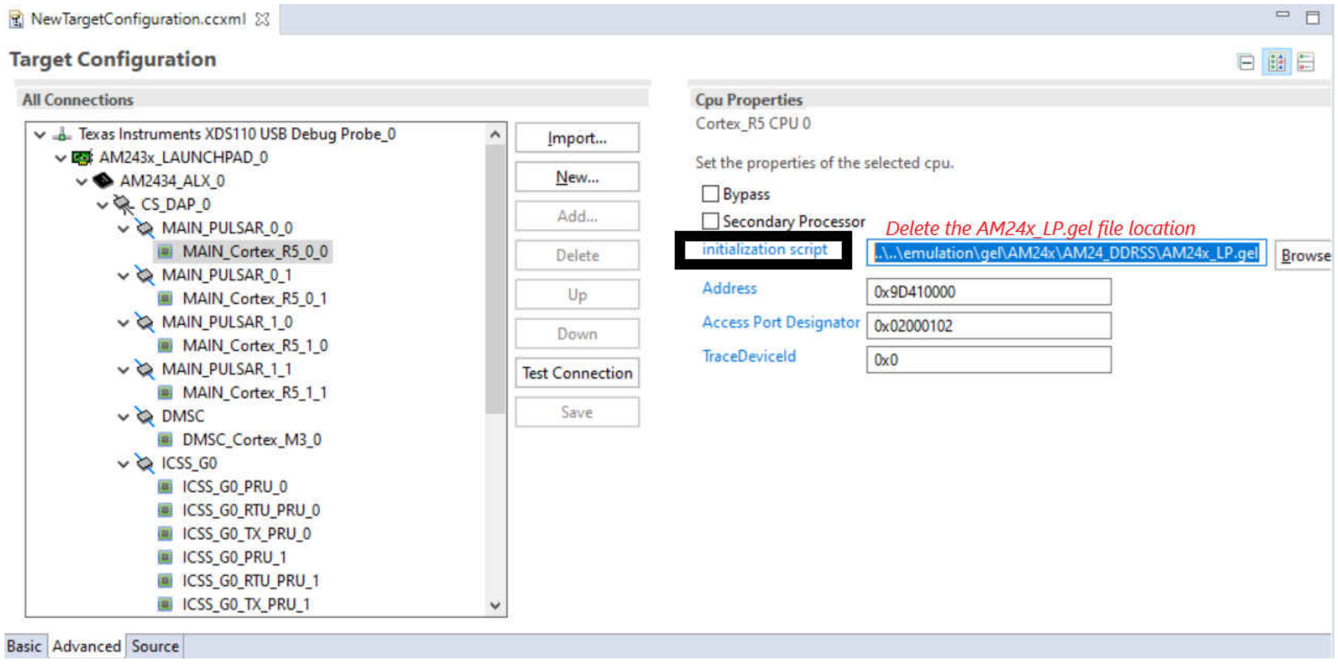
To test a connection, all changes must have been saved, the configuration file contains no errors and the connection type supports this function.

3. Click on Target Confication under **Advanced Setup**:


Advanced Setup

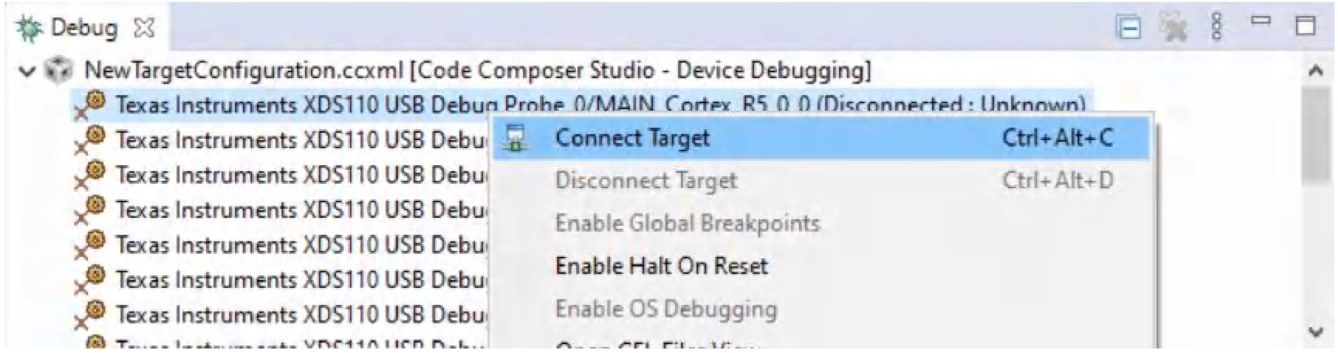
Target Configuration: lists the configuration options for the target.

4. Delete the initialization script from **each core** and hit on **Save**.

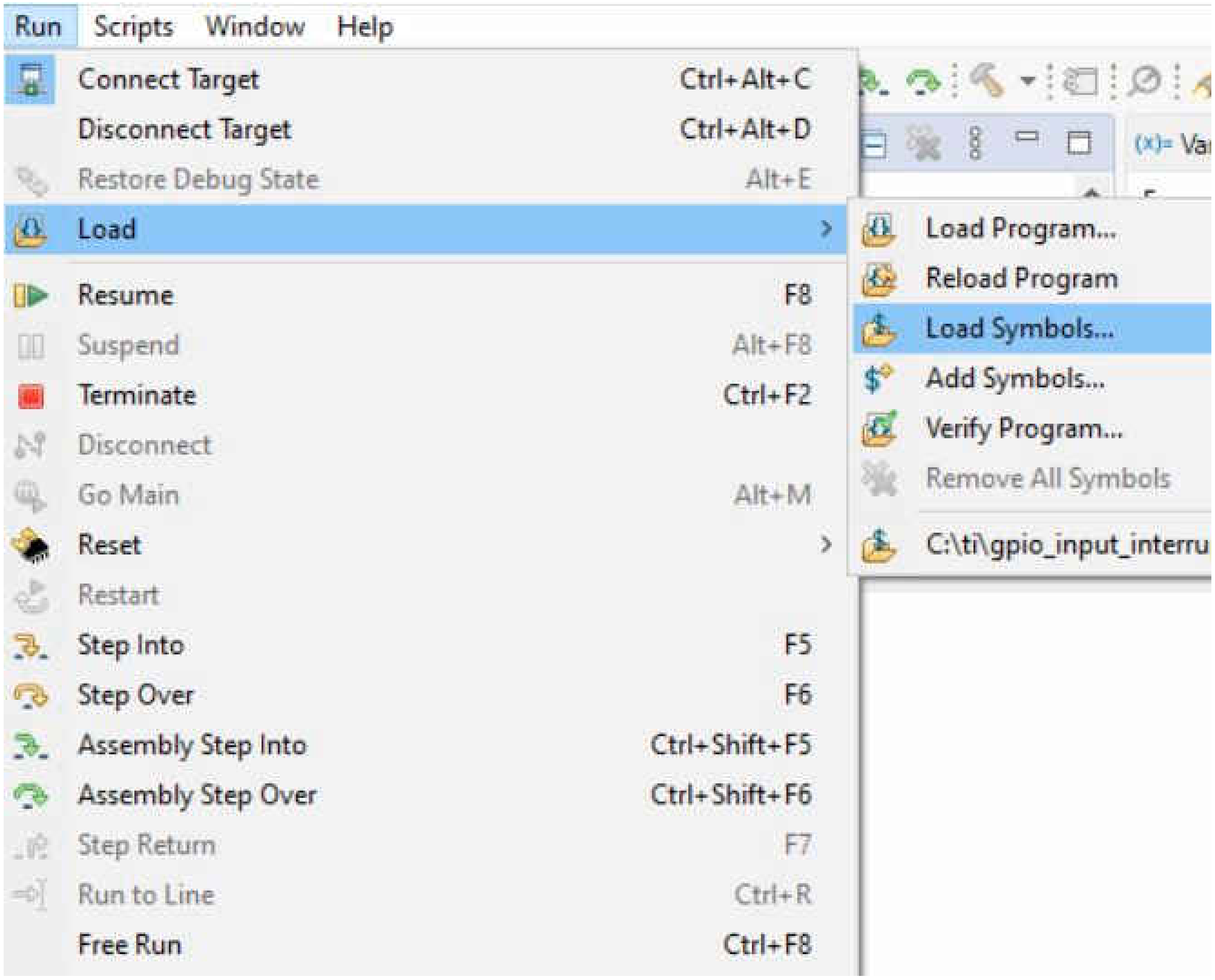


initialization script: ..\..\emulation\gel\AM24x\AM24_DDRSS\AM24x_LP.gel
Address: 0x9D410000
Access Port Designator: 0x02000102
TraceDeviceId: 0x0

5. Run the created target configuration file by right click on it and then select **Launch Selected Configuration**.
6. Connect to one of the cores by right click on it and select **Connect Target (Ctrl+Alt+C)**.


Debug
NewTargetConfiguration.ccxml [Code Composer Studio - Device Debugging]
Texas Instruments XDS110 USB Debug Probe_0/MAIN_Cortex_R5_0_0 (Disconnected : Unknown)
Texas Instruments XDS110 USB Debu
Connect Target Ctrl+Alt+C
Disconnect Target Ctrl+Alt+D
Enable Global Breakpoints
Enable Halt On Reset
Enable OS Debugging

7. If you halt the program, you would stop on a random address with "No symbols are defined". To be able to see some code and understand exactly where you are, you should load symbols with CCS by clicking on **Run->Load->Load Symbols** and choose the right project.



8.3.1 Debugging Init Code

8.3.1.1 Disable Auto-Run to Main

CCS by default automatically runs to main() upon loading a program. This option must be disabled in order to debug the init code of your application as described in the CCS User Guide.

9 Debugging Real-Time Control Loops

9.1 Trace

Arm tracing describes an advanced debug feature set of Arm devices that are able to stream out compressed core instruction information so a data stream of executed instructions can be reconstructed.

There are two types of trace:

- **Processor Trace:** Inspects the code execution and performs real-time gathering of instructions being executed in a processor
- **System Trace (STM):** A set of built-in capabilities on the device that monitors synchronization and timing between cores and on-chip peripherals. STM oversees system behavior.

CCS supports the Arm trace and this architecture is useful to detect complex, intermittent bugs and profile and fine tune code performance. If you are stuck with an intermittent or complex problem in the code, Processor Trace is usually your last line of defense to see the execution history. If you happen to have runtime problems but cannot identify what is causing the missed real-time deadlines, both Processor and System Trace help evidence it. If the system does not meet the expected or calculated power requirements, System Trace helps.

9.1.1 Processor / Core Trace

The Core Trace is responsible for the next following things:

- Capture all instructions that go through the CPU and copy them to a memory buffer:
  - Trace program address execution
  - Trace data writes to a specific location or range of locations
- Attach timestamps to each instruction
- Send data back to Code Composer Studio for post processing analysis:
  - Code Coverage
  - Profile

Expanding on the concept of Core Trace, the idea behind it is pretty straightforward: simply capture all the assembly instructions that ever get executed by the CPU and send them to the host PC for analysis. These are stored together with timestamps. Once this data is available, CCS can correlate the assembly instructions with the source code and thus allow looking at the code execution more easily. In addition to that it can also perform a multitude of other operations. The most relevant are: code coverage analysis, which means finding out which routines were actually executed, and profiling, which means knowing how many times and for how long each instruction and routine executed. However, one important detail defines its availability: since the execution speed of modern processors can reach billions of instructions per second, it is impossible to gather all this information without special hardware and some buffering between the device and the host PC. That is the reason why core trace is not available in all devices, and for the ones who have this feature there are two implementations with different levels of complexity:

ETB - Embedded Trace Buffer:

- Buffer size is limited (typically 2k to 8k)
- No modifications to the hardware are needed
- Any XDS emulator can be used
- Can run at core frequency

Pin Trace

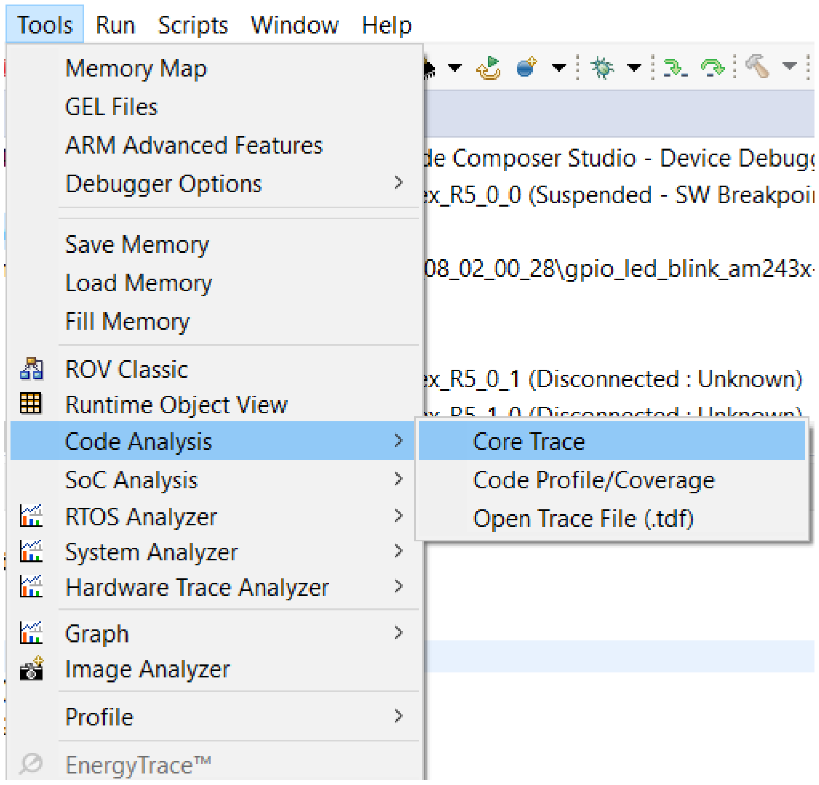
- A technology that features a trace buffer outside of the device without losing the ability to capture all instructions that are executed by the processing core.
- Buffer size is virtually unlimited (up to 2G)
- It features circular and one-shot modes
- Modifications to the hardware are needed
- Frequency depends on pin bandwidth

9.1.2 How to Use CCS to Capture Trace Data on an AM243x

When having a debug session started for launchpad, the target is connected to CCS and GPIO led blink example for the R5 core is loaded and halted at main.

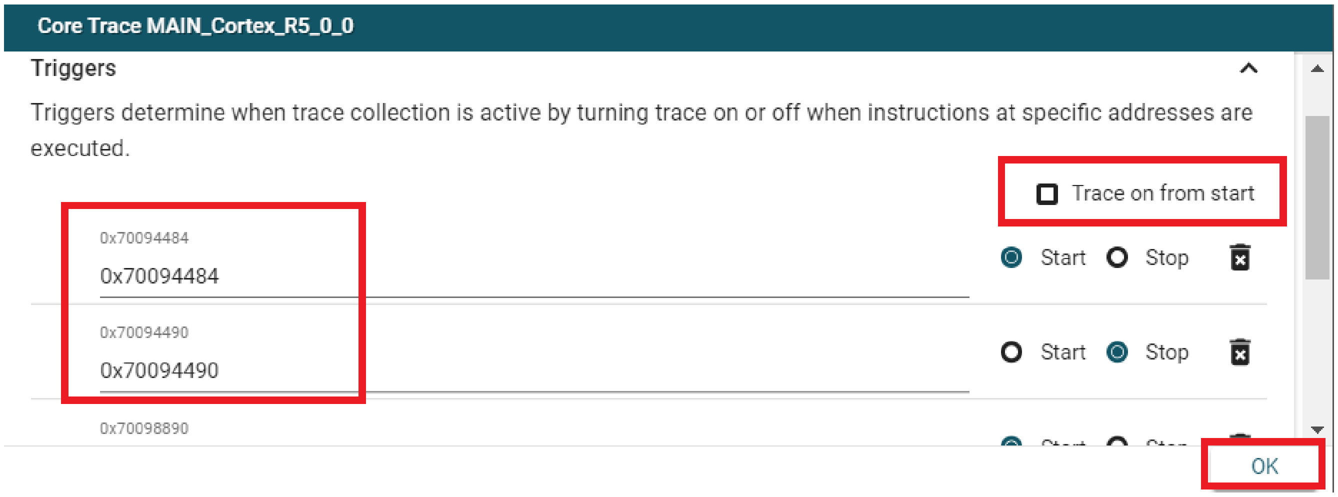
Under the **tools** → code analysis.

Select the first option (Core Trace) to capture core trace data, which is real-time gathering of instructions being executed by the core.



In the core trace tab, there are several options available. **Filters** are used to include or exclude trace data from specific address ranges. **Triggers** determine when trace collection is active by turning trace on or off instructions that specific addresses are executed.

Receiver settings allow you to specify the trace receiver to be used to specify the bugger type and specify if trace collection should be synchronized with the target execution. ETB is the only receiver option available for this launchpad (leave the rest of the options as default). Due to the small size of the ETB, it's not possible to capture core trace data for the entire application. Hence, you need to use triggers to enable and disable trace collection at specific addresses. This allows us to only capture the trace data that you are interested without risk of losing any data if the buffer wraps around. Our goal is to enable trace collection only around the part of the toggles the GPIO pin that toggles the LED. You can found those addresses by setting breakpoints and look at the Dissasembly of the breakpoints. Set start and stop addresess, and uncheck the "Trace on from start" option:


Core Trace MAIN_Cortex_R5_0_0

Triggers

Triggers determine when trace collection is active by turning trace on or off when instructions at specific addresses are executed.

☐ Trace on from start

0x70094484 — [● Start] Stop

0x70094490 — Start [● Stop]

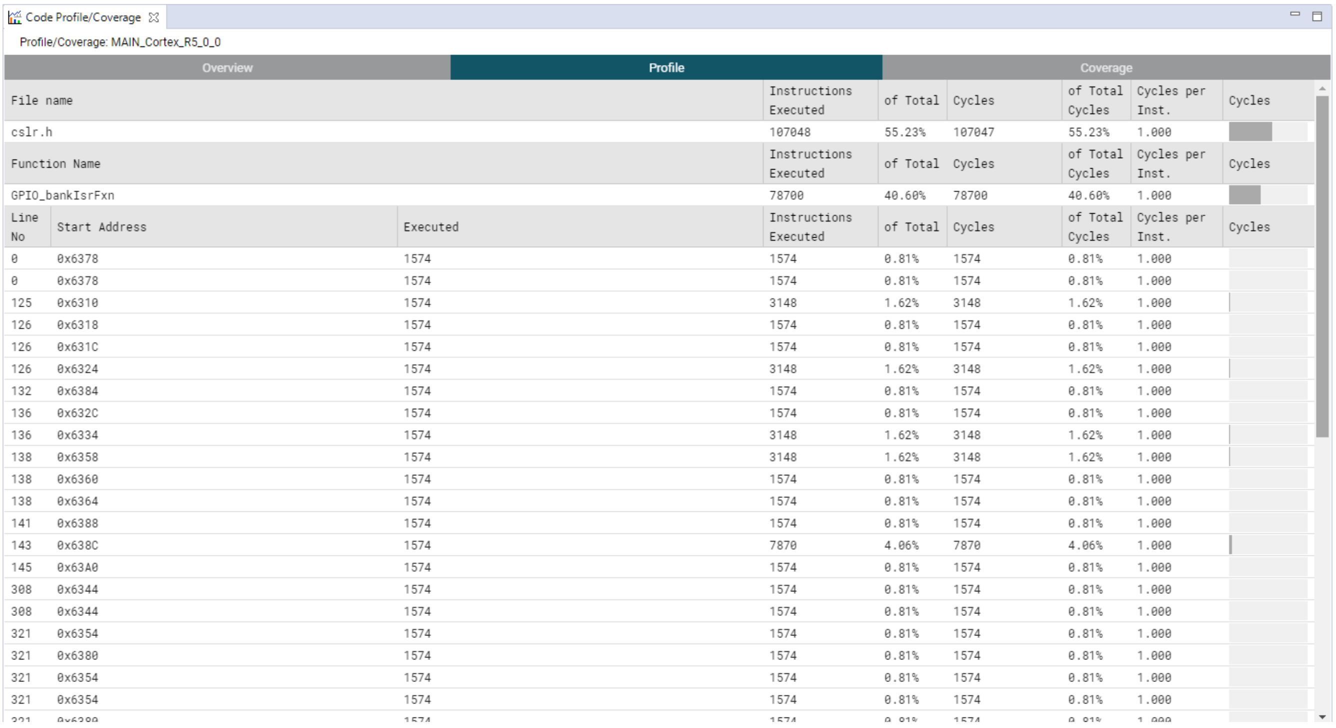
0x70098890

Hit on **"Ok"** and run the program (you can see that the program is running successfully since the led is blinking). Once the program has finished running and the target is halted at the exit point, the captured data in the ETB is sent to CCS and displayed in the core trace view. There you can see the data for each dace entry, data such as program address, the associated opcode and so on.

9.2 Code Profile / Coverage

Sitara MCU devices support code coverage that is particularly suited for embedded applications. In addition to being generally useful for thorough application development, code coverage is required by internal and external developers in the Industrial and Automotive markets for Functional Safety.

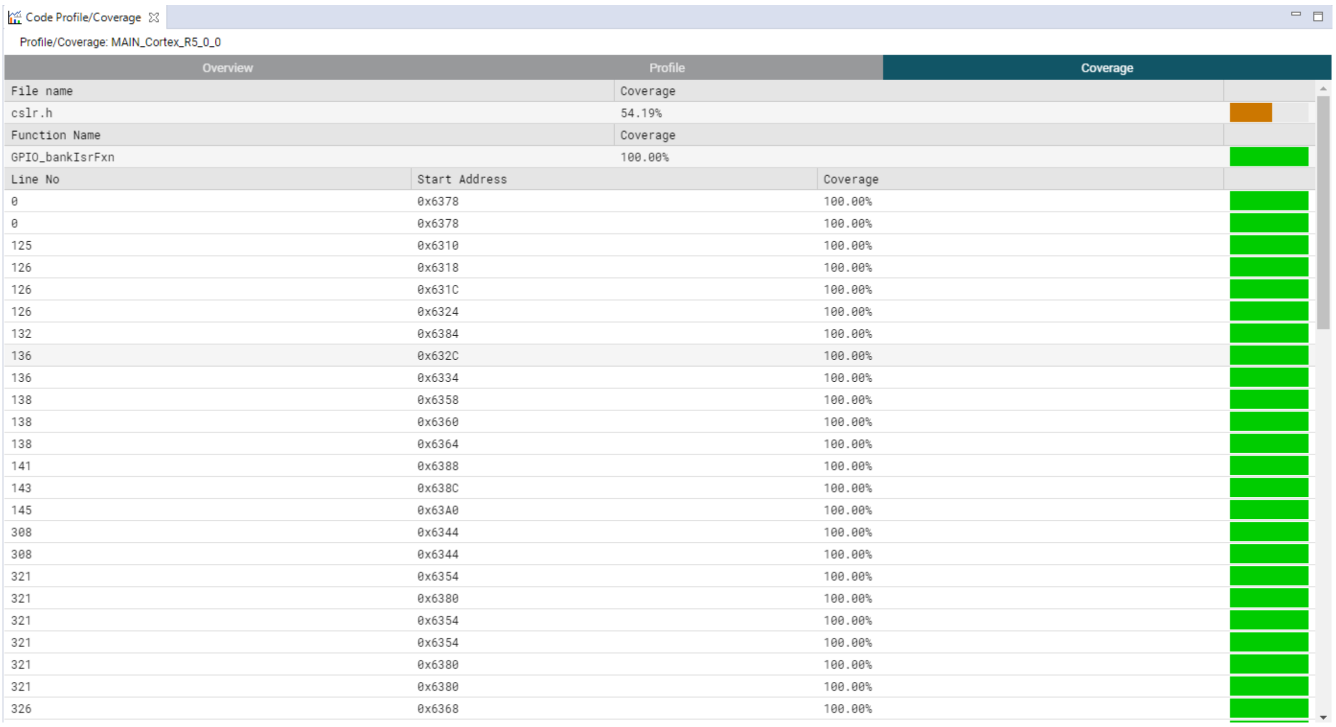
In order to open the Code Coverage tool, click on **Tools->Code Analysis->Code Profile / Coverage**.



File name	Instructions Executed	of Total	Cycles	of Total Cycles	Cycles per Inst.	Cycles
cslr.h	107048	55.23%	107047	55.23%	1.000	

Function Name	Instructions Executed	of Total	Cycles	of Total Cycles	Cycles per Inst.	Cycles
GPIO_bankIsrFxn	78700	40.60%	78700	40.60%	1.000	

Line No	Start Address	Executed	Instructions Executed	of Total	Cycles	of Total Cycles	Cycles per Inst.	Cycles
0	0x6378	1574	1574	0.81%	1574	0.81%	1.000	
0	0x6378	1574	1574	0.81%	1574	0.81%	1.000	
125	0x6310	1574	3148	1.62%	3148	1.62%	1.000	
126	0x6318	1574	1574	0.81%	1574	0.81%	1.000	
126	0x631C	1574	1574	0.81%	1574	0.81%	1.000	
126	0x6324	1574	3148	1.62%	3148	1.62%	1.000	
132	0x6384	1574	1574	0.81%	1574	0.81%	1.000	
136	0x632C	1574	1574	0.81%	1574	0.81%	1.000	
136	0x6334	1574	3148	1.62%	3148	1.62%	1.000	
138	0x6358	1574	3148	1.62%	3148	1.62%	1.000	
138	0x6360	1574	1574	0.81%	1574	0.81%	1.000	
138	0x6364	1574	1574	0.81%	1574	0.81%	1.000	
141	0x6388	1574	1574	0.81%	1574	0.81%	1.000	
143	0x638C	1574	7870	4.06%	7870	4.06%	1.000	
145	0x63A0	1574	1574	0.81%	1574	0.81%	1.000	
308	0x6344	1574	1574	0.81%	1574	0.81%	1.000	
308	0x6344	1574	1574	0.81%	1574	0.81%	1.000	
321	0x6354	1574	1574	0.81%	1574	0.81%	1.000	
321	0x6380	1574	1574	0.81%	1574	0.81%	1.000	
321	0x6354	1574	1574	0.81%	1574	0.81%	1.000	
321	0x6354	1574	1574	0.81%	1574	0.81%	1.000	



File name	Coverage	
cslr.h	54.19%	🟧

Function Name	Coverage	
GPIO_bankIsrFxn	100.00%	🟩

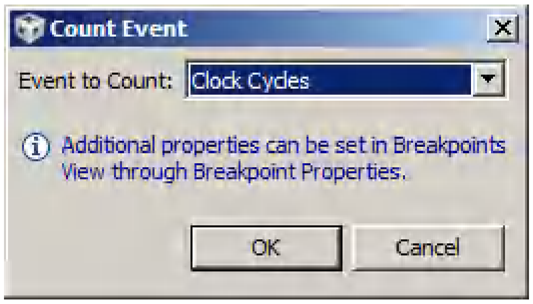
Line No	Start Address	Coverage	
0	0x6378	100.00%	🟩
0	0x6378	100.00%	🟩
125	0x6310	100.00%	🟩
126	0x6318	100.00%	🟩
126	0x631C	100.00%	🟩
126	0x6324	100.00%	🟩
132	0x6384	100.00%	🟩
136	0x632C	100.00%	🟩
136	0x6334	100.00%	🟩
138	0x6358	100.00%	🟩
138	0x6360	100.00%	🟩
138	0x6364	100.00%	🟩
141	0x6388	100.00%	🟩
143	0x638C	100.00%	🟩
145	0x63A0	100.00%	🟩
308	0x6344	100.00%	🟩
308	0x6344	100.00%	🟩
321	0x6354	100.00%	🟩
321	0x6380	100.00%	🟩
321	0x6354	100.00%	🟩
321	0x6354	100.00%	🟩
321	0x6380	100.00%	🟩
321	0x6380	100.00%	🟩
326	0x6368	100.00%	🟩

The code coverage and profiling shows you how often a function / file is called, how much code of the file was executed, how many instructions and cycles it took and more.

9.2.1 CCS Count Event

Count Event can be used to count different events, one of them being clock cycles. Using Count Event for measuring clock cycles could be one method of profiling code.

1. Use the pulldown to select **Count Event**.



Event to Count: Clock Cycles

Additional properties can be set in Breakpoints View through Breakpoint Properties.

2. Enable a breakpoint in your code by double-clicking on the line in the source file.
3. Select the **Resume** icon. When the breakpoint is reached the Count Event will display the number of Clock Cycles.

Note that the number of CPU cycles can vary greatly depending on the type of memory the code is running from (Flash, RAM, external).

9.3 Real-Time UART Monitor

Real time debug is enabled by UART connection between CCS and Am243x. With real time debug, global variables can be added to expression window and ready for read/write during continuous run of the program. The connection is built by debug program in the listed files.

- Serial_Cmd_Monitor.c
- Serial_Cmd_Monitor.h
- Serial_Cmd_HAL.c
- Serial_Cmd_HAL.h

Even though there are four files listed here, there is only two functions required in application program. One is "SerialCmd_init()" called in initialization and the other is "SerialCmd_read()" called in background loop of BareMetal or low priority task of RTOS. This section focus on how to create the UART connection and how to launch real time debug in CCS.

9.3.1 Confirm CCS Features

It is recommended to check the following CCS driver file if the CCS version is older than 11.1. The configuration of Cortex_R5 should be similar to the below. If any line is missing, it is necessary to add it. As for the content of the lines, COM Port and Baud Rate need to be updated in target configuration file, which is included in the next step.

ccs\ccs_base\common\targetdb\drivers\gti_uart_driver.xml

```
<isaType="Cortex_R5" ProcID="0x75803400">
   <driverfile="../../../DebugServer/drivers/XPCOMToGTIAdapter.dvr">
      <propertyType="stringfield" Value="COM14" id="COM Port"/>
      <propertyType="stringfield" Value="9600" id="Baud Rate"/>
      <propertyType="hiddenfield" Value="Little Endian" id="Endianness"/>
      <propertyType="hiddenfield" Value="32" id="Word Size Page 0"/>
      <propertyType="hiddenfield" Value="8" id="Minimum Addressable Size Page 0"/>
      <propertyType="hiddenfield" Value="@ti.com/UARTMonitor;1" id="XPCOM Class ID"/>
      <propertyType="hiddenfield" Value="Flash DLL Delegate" id="TargetAccess"/>
      <connectionTypeType="UARTConnection"/>
   </driver>
</isa>
```

9.3.2 Create Target Configuration File

A step-to-step guide is given in the screen shots shown below:

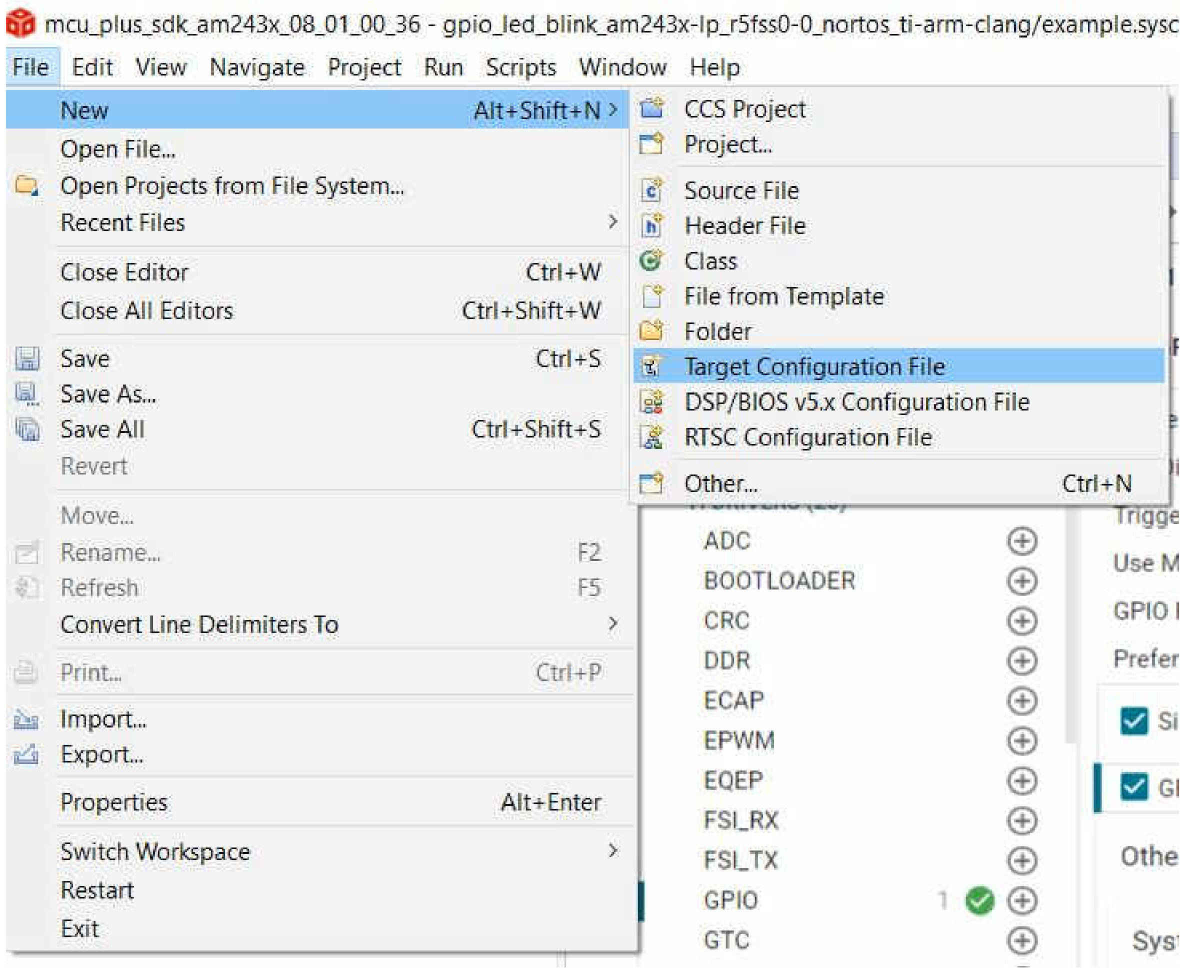
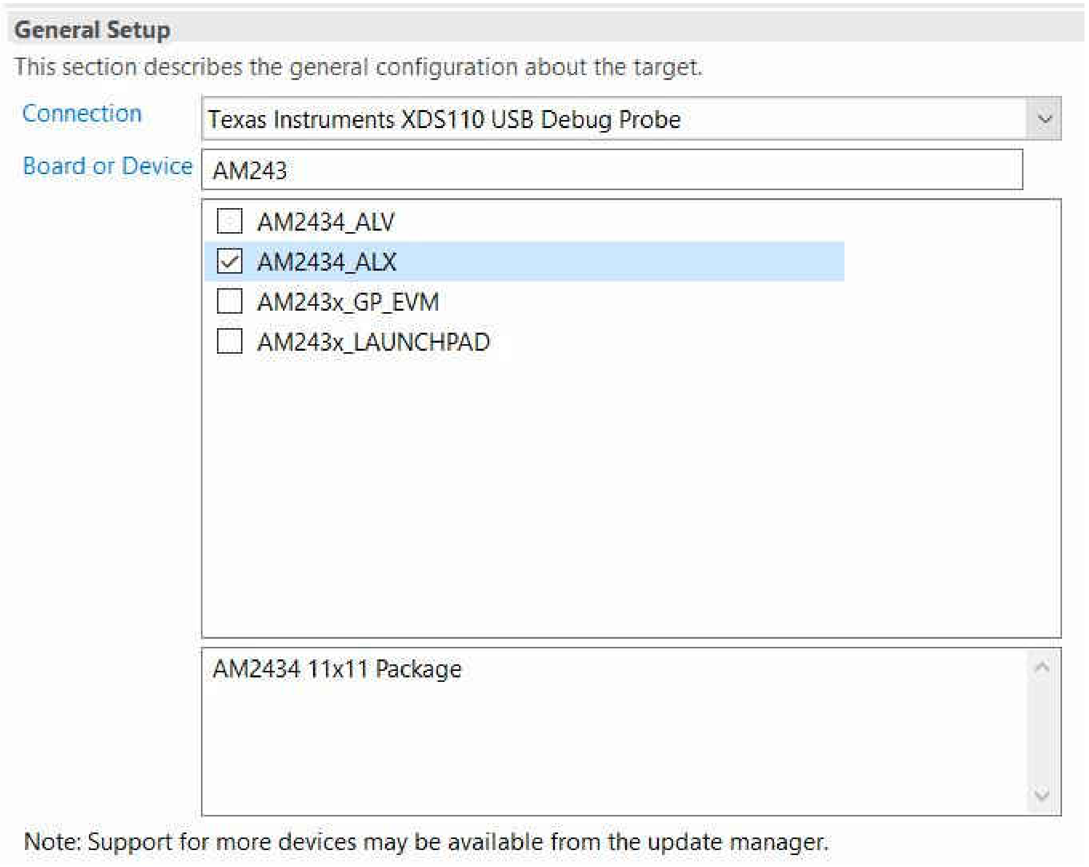


Figure 9-1. Create New Target Configuration File


General Setup

This section describes the general configuration about the target.

Connection: Texas Instruments XDS110 USB Debug Probe

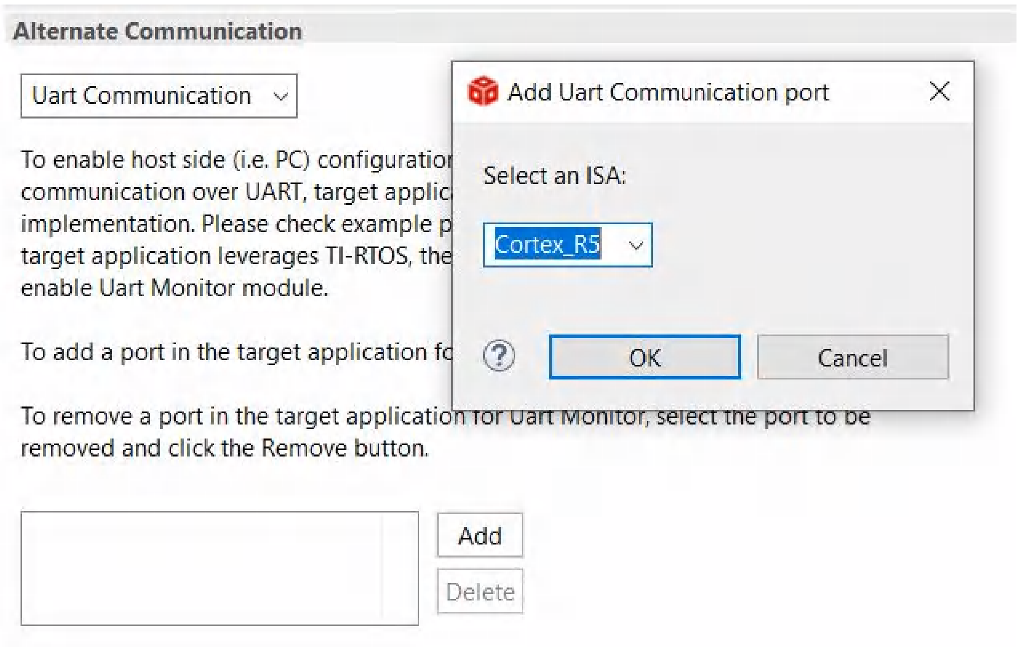
Board or Device: AM243

- [ ] AM2434_ALV
- [x] AM2434_ALX
- [ ] AM243x_GP_EVM
- [ ] AM243x_LAUNCHPAD

AM2434 11x11 Package

Note: Support for more devices may be available from the update manager.

Figure 9-2. Select JTAG Connection and Device


Alternate Communication

Uart Communication

To enable host side (i.e. PC) configuratio communication over UART, target applic implementation. Please check example p target application leverages TI-RTOS, the enable Uart Monitor module.

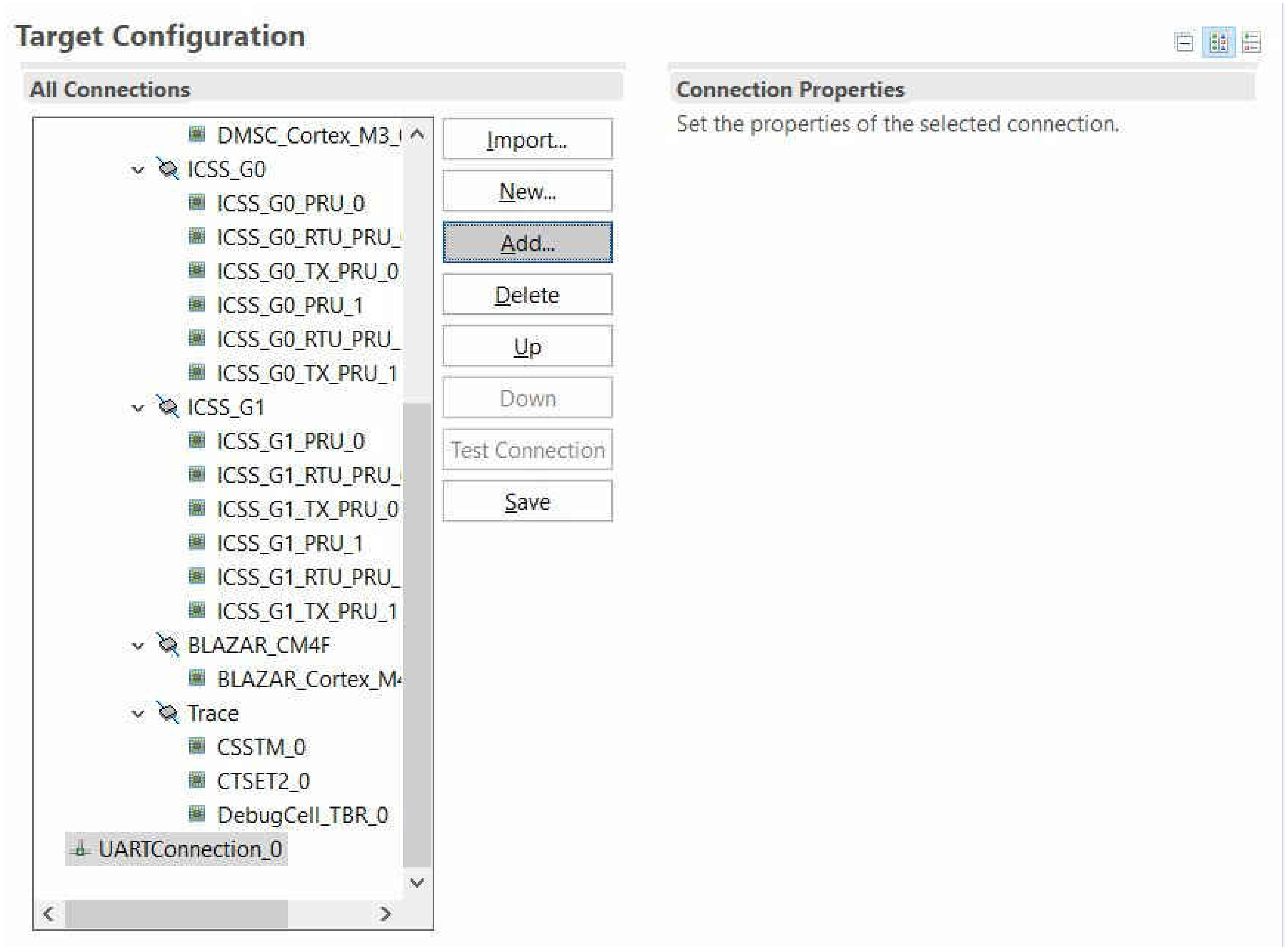
To add a port in the target application fo

To remove a port in the target application for Uart Monitor, select the port to be removed and click the Remove button.

Add Uart Communication port

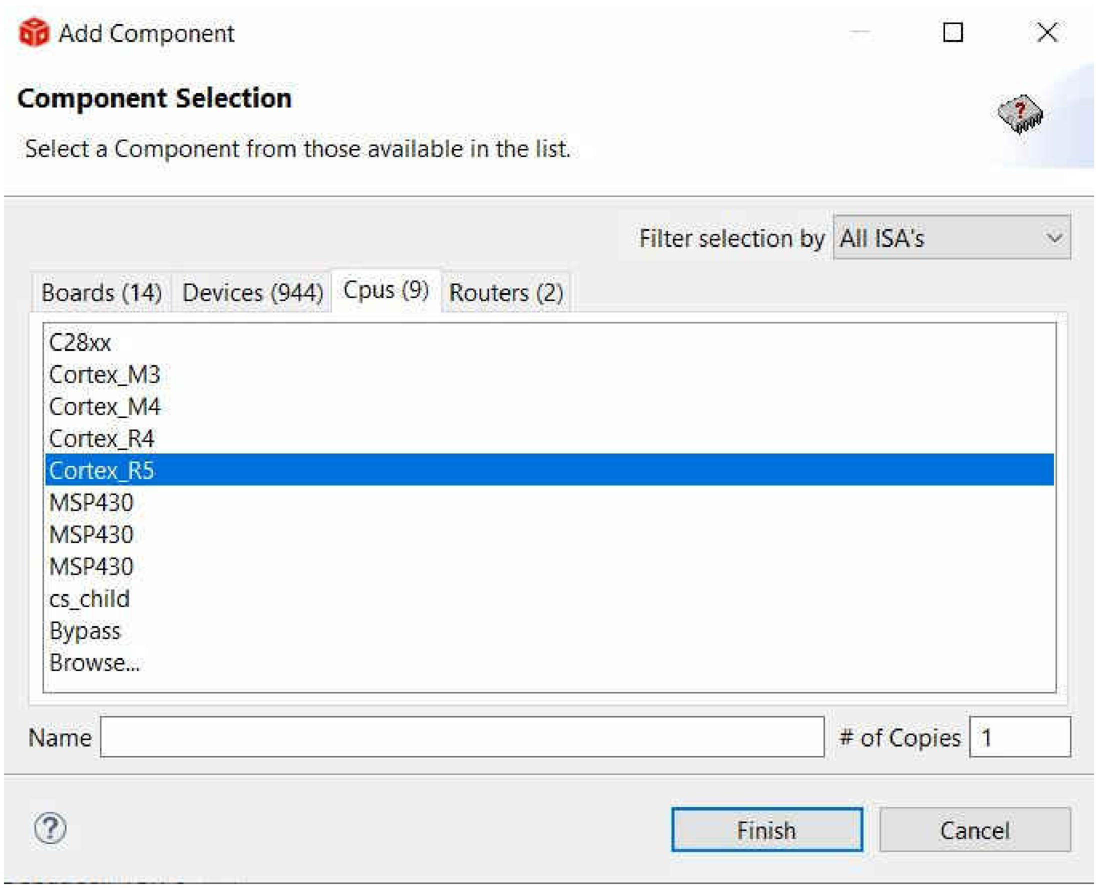
Select an ISA: Cortex_R5

Figure 9-3. Add UART Communication Port



Set the properties of the selected connection.

Figure 9-4. Open Advanced Target Configuration


Add Component

Component Selection

Select a Component from those available in the list.

Filter selection by All ISA's

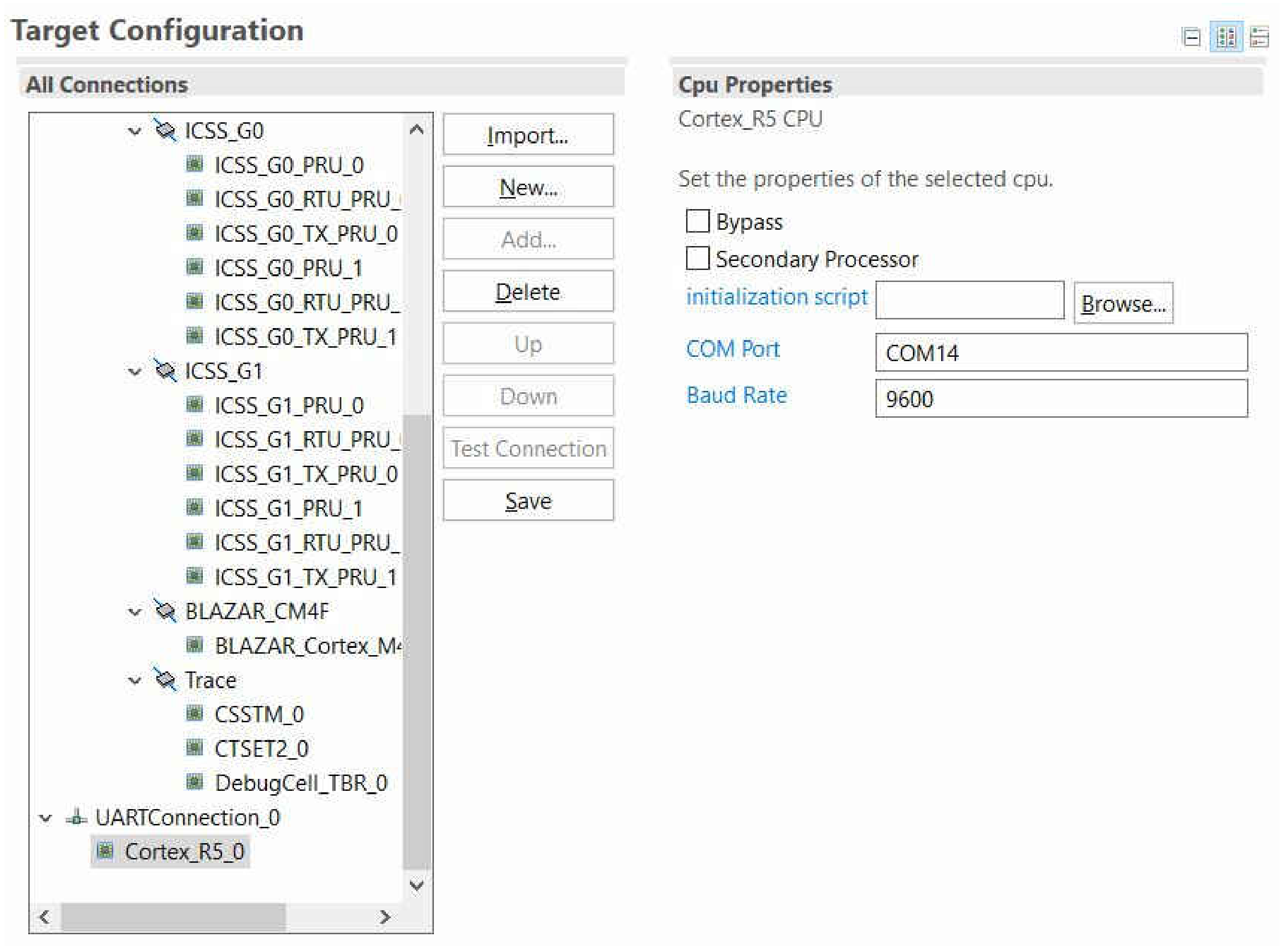
Boards (14) Devices (944) Cpus (9) Routers (2)

C28xx
Cortex_M3
Cortex_M4
Cortex_R4
Cortex_R5
MSP430
MSP430
MSP430
cs_child
Bypass
Browse...

Name

# of Copies 1

Figure 9-5. Add Component



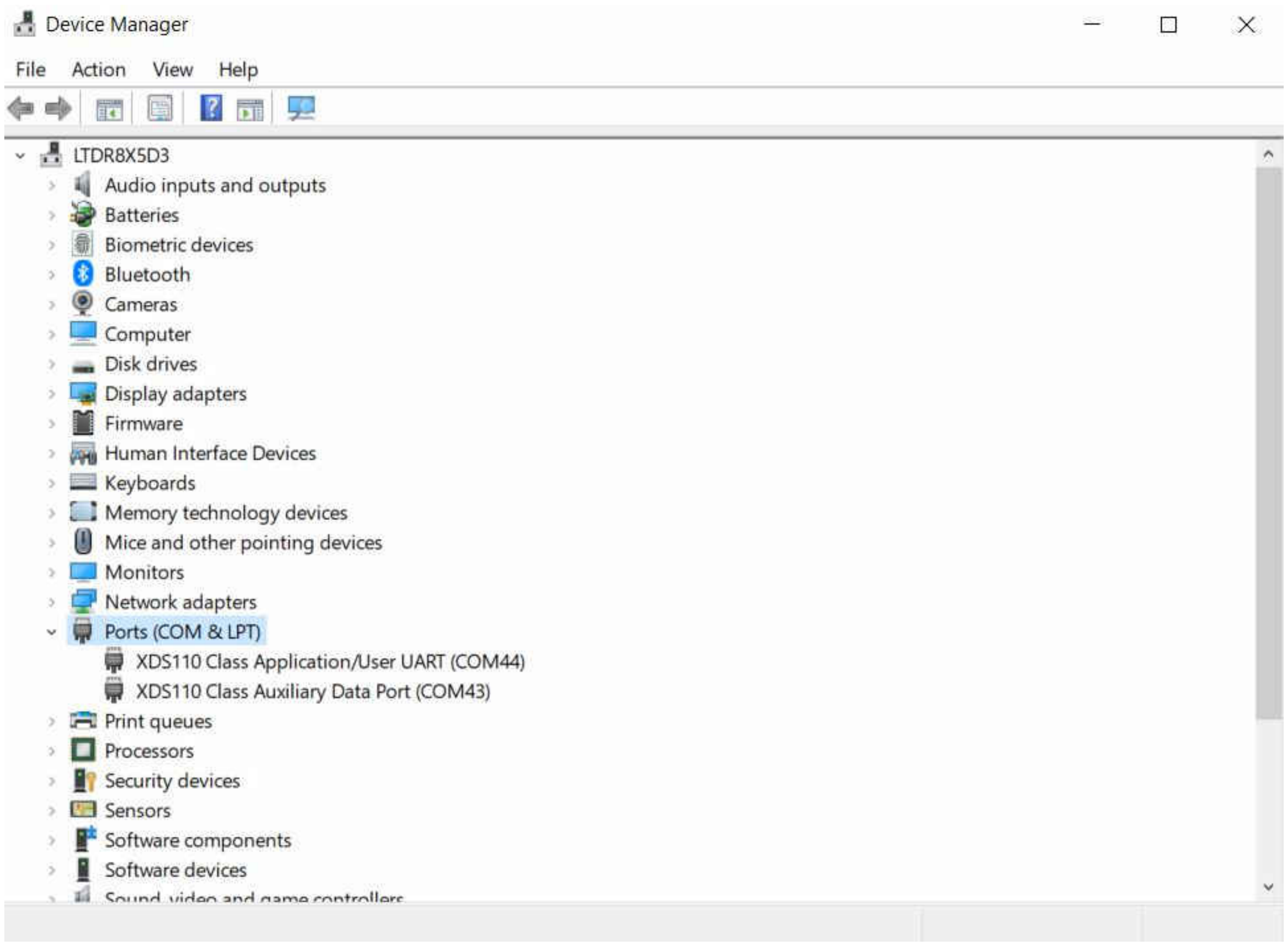
Cortex_R5 CPU

Set the properties of the selected cpu.

- [ ] Bypass
- [ ] Secondary Processor

initialization script	
COM Port	COM14
Baud Rate	9600

Figure 9-6. Select CPU Properties

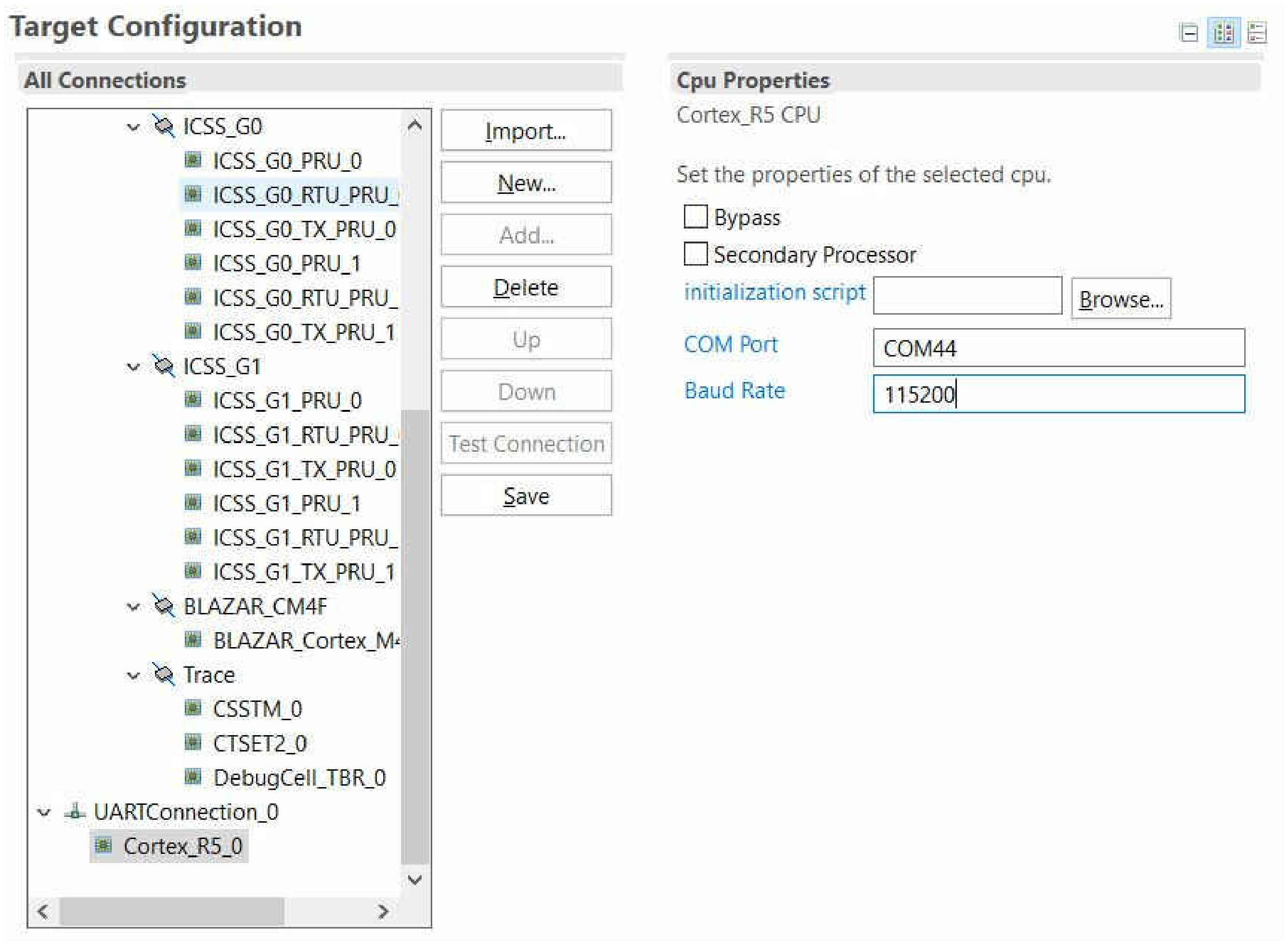


Device Manager

File Action View Help

LTDR8X5D3
- Audio inputs and outputs
- Batteries
- Biometric devices
- Bluetooth
- Cameras
- Computer
- Disk drives
- Display adapters
- Firmware
- Human Interface Devices
- Keyboards
- Memory technology devices
- Mice and other pointing devices
- Monitors
- Network adapters
- Ports (COM & LPT)
  - XDS110 Class Application/User UART (COM44)
  - XDS110 Class Auxiliary Data Port (COM43)
- Print queues
- Processors
- Security devices
- Sensors
- Software components
- Software devices
- Sound, video and game controllers

Figure 9-7. Find XDS110 UART COM Port


Target Configuration

All Connections

Cpu Properties

Cortex_R5 CPU

Set the properties of the selected cpu.

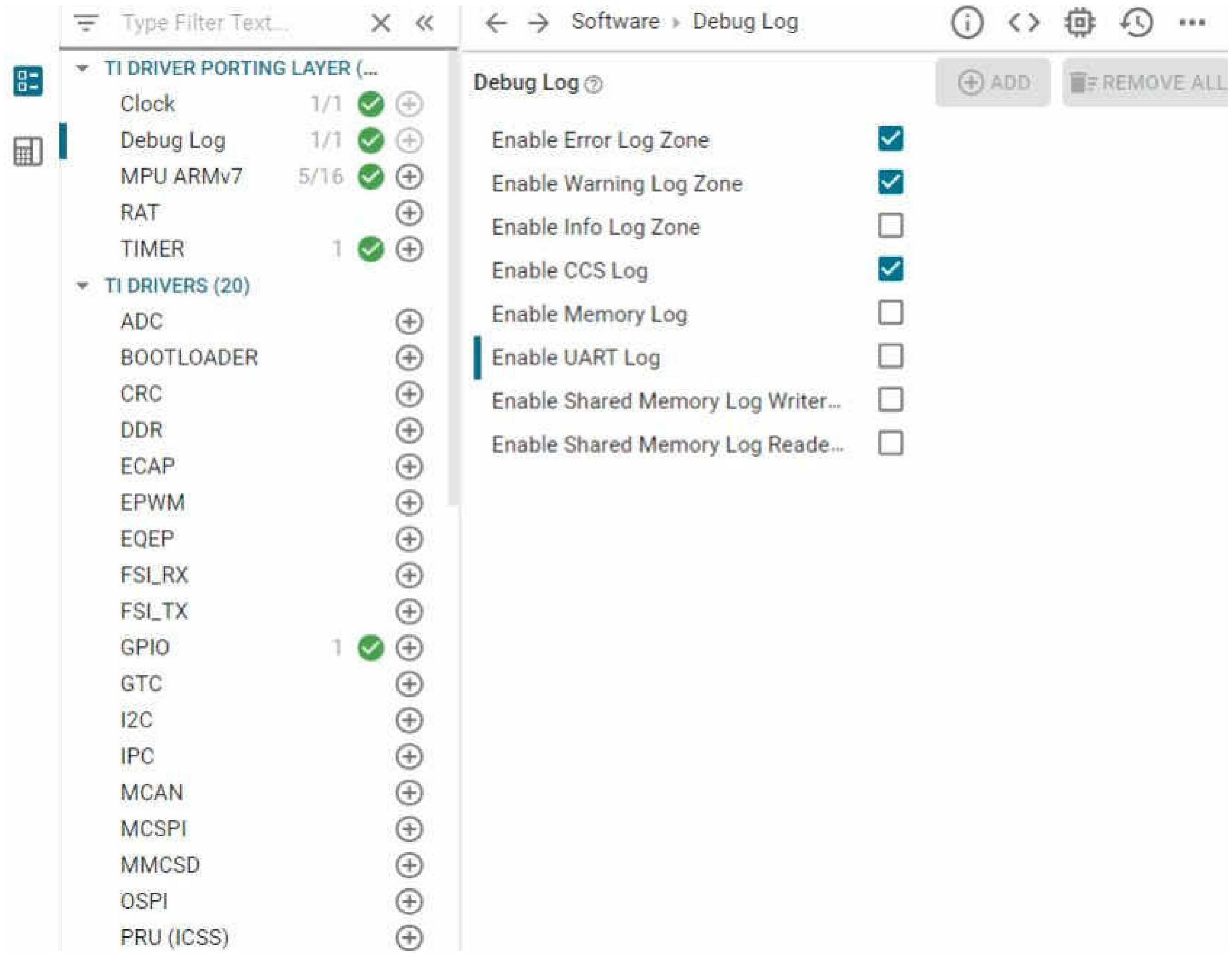
- [ ] Bypass
- [ ] Secondary Processor

initialization script	
COM Port	COM44
Baud Rate	115200

Figure 9-8. Update CPU Properties in Advanced Target Configuration

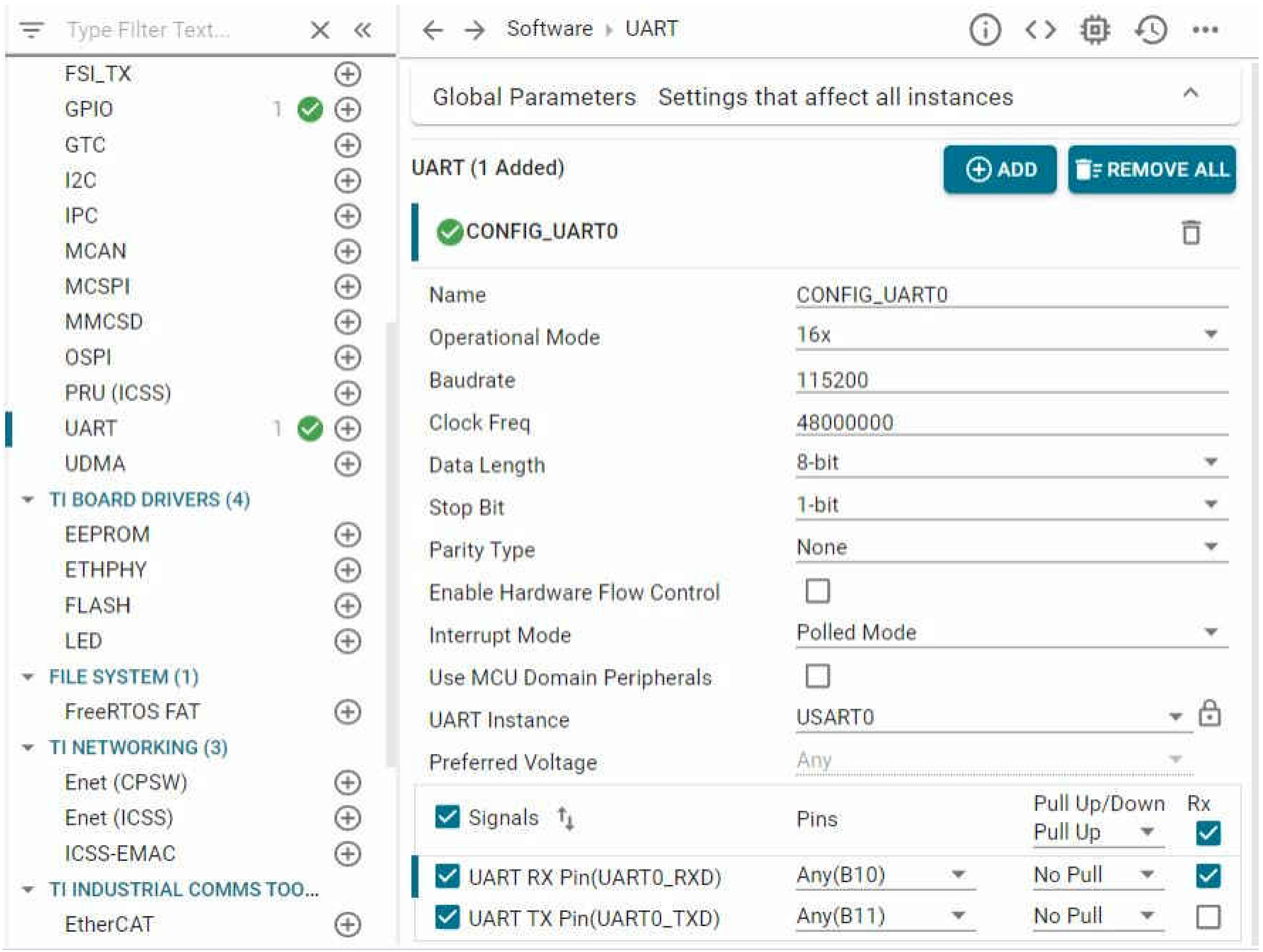
9.3.3 Add Serial Command Monitor Software

There are multiple ways to use UART0 as a debug interface. They are Debug Log and Serial Command Monitor. Debug Log is a built-in tool located at Driver Porting Layer of SDK. Like Serial Cmd Monitor, its function must be located out of interrupt callback. It is a handy tool enabling string input and output. But, input and output go through UART console only. There is no built-in GUI like Expression Window and Graph in CCS. It is recommended to disable UART0 in Debug Log at SysCfg and configure UART0 instance for Serial Command Monitor. As the name of UART in Sysconfig, "CONFIG_UART_CONSOLE", matches the handle name in "Serial_Cmd_HAL.c", it is not necessary to modify the two functions required by initialization and background loop.



Enable Error Log Zone	☑
Enable Warning Log Zone	☑
Enable Info Log Zone	☐
Enable CCS Log	☑
Enable Memory Log	☐
Enable UART Log	☐
Enable Shared Memory Log Writer...	☐
Enable Shared Memory Log Reade...	☐

Figure 9-9. Disable UART Log in Debug Log



Name	CONFIG_UART0
Operational Mode	16x
Baudrate	115200
Clock Freq	48000000
Data Length	8-bit
Stop Bit	1-bit
Parity Type	None
Enable Hardware Flow Control	☐
Interrupt Mode	Polled Mode
Use MCU Domain Peripherals	☐
UART Instance	USART0
Preferred Voltage	Any

Signals	Pins	Pull Up/Down	Rx
✓ Signals		Pull Up	✓
✓ UART RX Pin(UART0_RXD)	Any(B10)	No Pull	✓
✓ UART TX Pin(UART0_TXD)	Any(B11)	No Pull	☐

Figure 9-10. Configure UART0 Instance

Then, add the serial monitor functions that were used in the benchmark demo example:

```
void benchmarkdemo_foc_main(void)
{
    Drivers_open();
    DebugP_log("\r\n START FOC benchmark\r\n");
    App_statsInit(APP_ID_FOC);
    SerialCmd_init();
    while (1)
    {
        SerialCmd_read();
        if (App_timerHasExpired())
        {
            focLoop(1);
        }
        App_statsUpdateUI();
    }
    Drivers_close();
}
```

9.3.4 Launch Real Time Debug

After building the program, the debug window should be opened with the target configuration file you created. If the created target configuration file is not already opened, it can be located into "User Defined" folder of the "Target Configuration" window. Right click on the file, a menu shows up and there is a option "Launch Selected Configuration" as shown in Figure 9-11. Then, debug window shows up. The steps to connect target, load image and run via JTAG can be found in many CCS tutorials. The processor must be running continuously before connecting to UART. As the UART connection is based on continuous operation of the program, the UART connection will be broken and CCS will be frozen by Break-point, Suspend, Terminate or any other events stopping the Serial Command Monitor program from running. Sometimes, it is just a habit to use those features when they are available. It is recommended to disconnect target via JTAG while using UART connection. When the processor is running, UART connection can be established by simply select the UART connection → Run → Load → Load Symbols

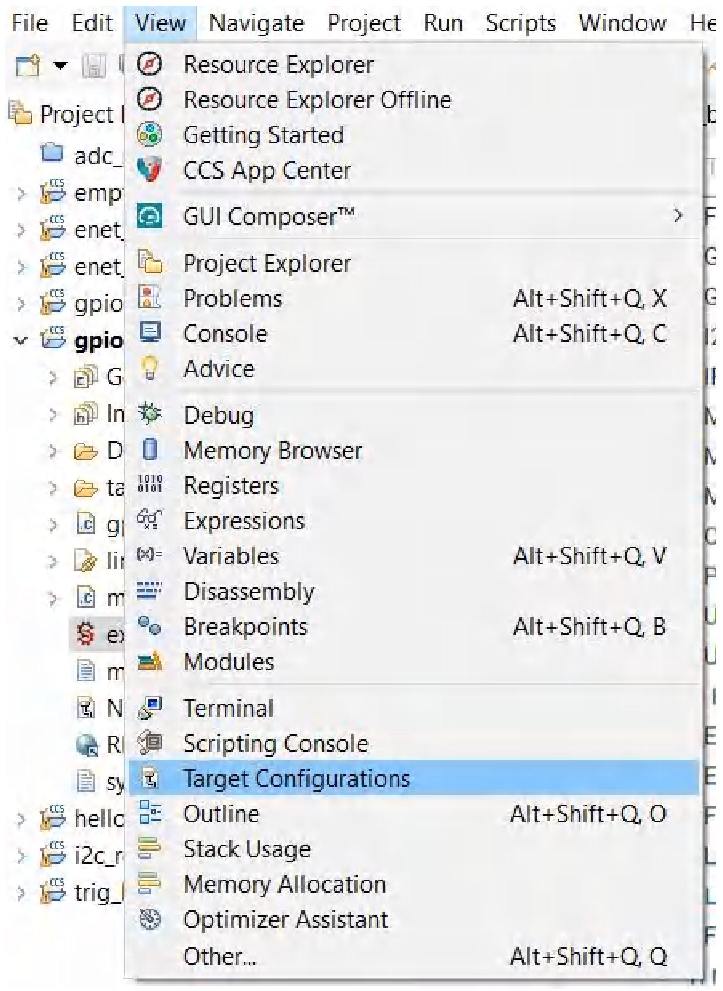


Figure 9-11. Locate Target Configuration File

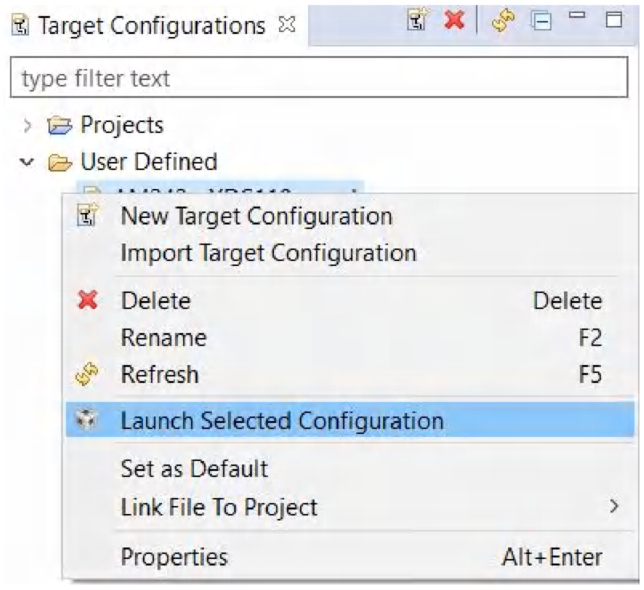


Figure 9-12. Launch Selected Configuration

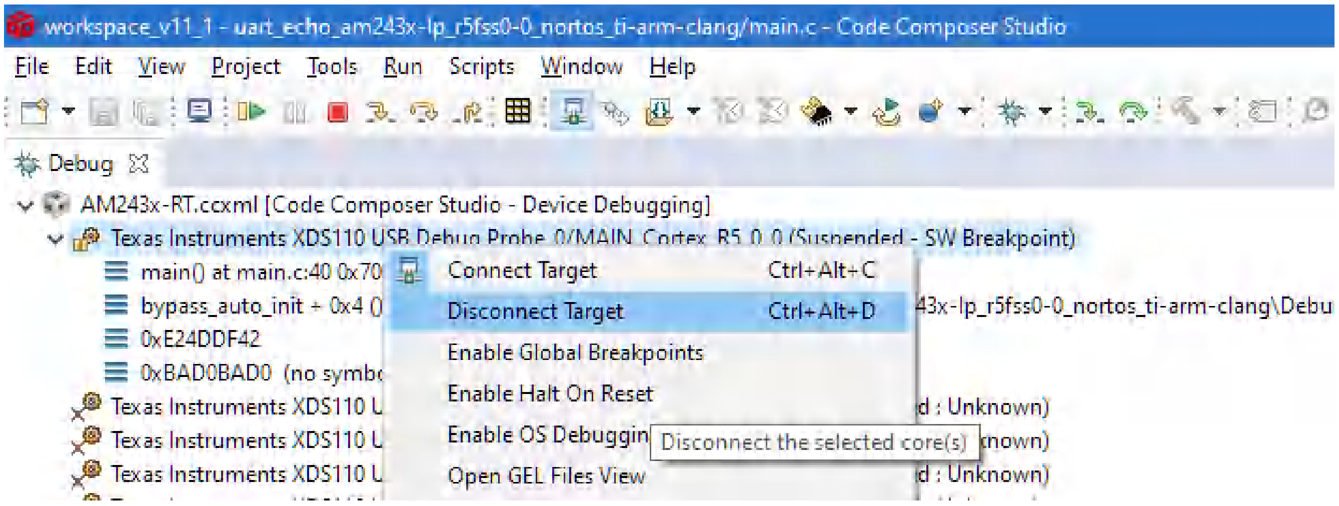

workspace_v11_1 - uart_echo_am243x-lp_r5fss0-0_nortos_ti-arm-clang/main.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
AM243x-RT.ccxml [Code Composer Studio - Device Debugging]
Texas Instruments XDS110 USB Debug Probe_0/MAIN_Cortex_R5_0_0 (Suspended - SW Breakpoint)
main() at main.c:40 0x70
bypass_auto_init + 0x4 ()
0xE24DDF42
0xBAD0BAD0 (no symbo
Connect Target Ctrl+Alt+C
Disconnect Target Ctrl+Alt+D
Enable Global Breakpoints
Enable Halt On Reset
Enable OS Debuggin
Open GEL Files View
Disconnect the selected core(s)

Figure 9-13. Disconnect JATG Connection

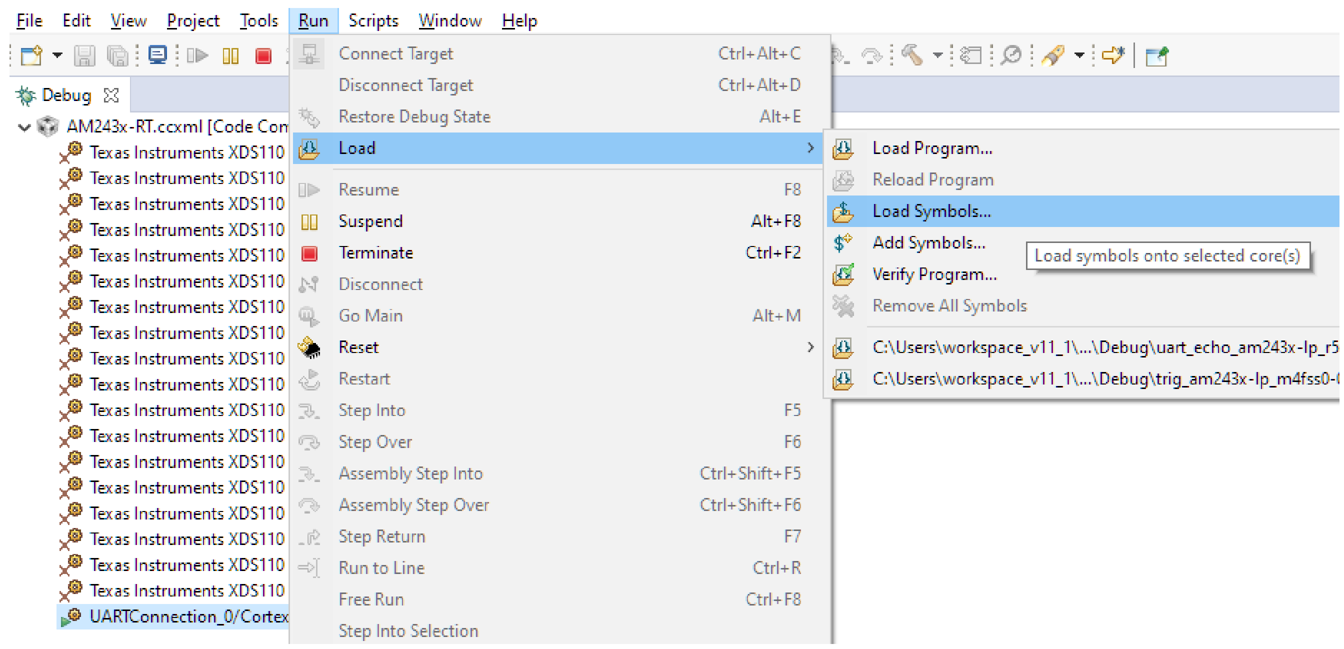

File Edit View Project Tools Run Scripts Window Help
Debug
AM243x-RT.ccxml [Code Con
Texas Instruments XDS110
UARTConnection_0/Cortex
Connect Target Ctrl+Alt+C
Disconnect Target Ctrl+Alt+D
Restore Debug State Alt+E
Load
Resume F8
Suspend Alt+F8
Terminate Ctrl+F2
Disconnect
Go Main Alt+M
Reset
Restart
Step Into F5
Step Over F6
Assembly Step Into Ctrl+Shift+F5
Assembly Step Over Ctrl+Shift+F6
Step Return F7
Run to Line Ctrl+R
Free Run Ctrl+F8
Step Into Selection
Load Program...
Reload Program
Load Symbols...
Add Symbols...
Load symbols onto selected core(s)
Verify Program...
Remove All Symbols
C:\Users\workspace_v11_1\...\Debug\uart_echo_am243x-lp_r5
C:\Users\workspace_v11_1\...\Debug\trig_am243x-lp_m4fss0-

Figure 9-14. Establish UART Connection

10 E2E Support Forums

If you have any questions or issues, create a thread on TI's E2E support forums.