Tutorial

Debugging Techniques

For the DA1468x Chips

Abstract

This tutorial should be used as a reference guide to gain a deeper understanding of the 'Debugging Techniques' for DA1468x family of devices. As such, it covers a broad range of topics including an introduction to debugging tools that are available for performing a debugging session as well as detailed description of the most common system faults. Furthermore, it covers a number of sections containing in depth analysis of real use cases of system failures and how to deal with them.

Contents

Figures

Tables

Terms and Definitions

API	Application Programming Interface
BOD	Brown-out Detection
DevKit	Development Kit
HW	Hardware
IDE	Integrated Development Environment
JTAG	Joint Test Action Group
LR	Link Register
MCU	Microcontroller Unit
PC	Program Counter
POR	Power-on Reset
SDK	Software Development Kit
SW	Software
SWD	Serial Wire Debug
WDOG	Watchdog

References

[1] UM-B-044, DA1468x Software Platform Reference, User Manual, Dialog Semiconductor.

1 Introduction

1.1 Before You Start

Before you start you need to:

• Install the latest SmartSnippets Studio

• Download the latest SDK (currently version 1.0.12.1078)

These can be downloaded from the Dialog Semiconductor support portal.

Additionally, for this tutorial either a Pro or Basic Development kit is required.

The key goals of this tutorial are to:

• Provide a basic understanding of debugging and debuggers

• Explain reasons that may cause the device to fail and the steps that need to be followed depending on the root cause of the failure

• Give tips and insights for dealing with failures of the system

2 Debugging

A debugging session lets you control the execution of a program by setting breakpoints, suspending executed programs, stepping through the code, and examining the contents of variables and structures. This section demonstrates the most useful tools and features of the Eclipse IDE that can be used during debugging.

2.1 Initiating a Debugging Session

To initiate a debugging session, a debugger must be attached to the SWD or JTAG port of the application MCU. The DA1468x Pro DevKit has a debugger mounted on it, thus eliminating the need for an external one which can be quite an expensive tool.

There are two ways to start a debugging session, the selected method depends on the issue being examined. One option is for the developer to start inspecting program execution from the beginning by triggering a hardware reset while attaching the debugger to the MCU. This means that the program is executed from its very first step and by default the first breakpoint is set to the main function of the application. The second option is for the debugger to be attached without triggering any reset to the system.

1. Click on the **Debug** button (1) and select either **ATTACH** or **QSPI** (2). Selecting **QSPI** triggers a hardware reset to the system while the debugger is being attached.
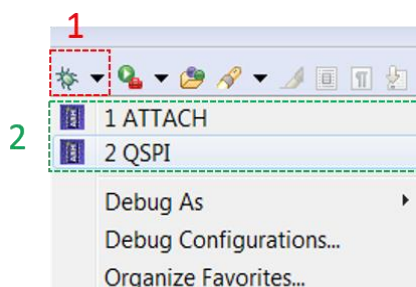
Figure 1: Initiating a Debugging Session

Debugging Techniques

2. Depending on the configuration, a pop-up window might be displayed asking for permission to switch to the **Debug** view.

Figure 2: Debug View

3. When switching to the **Debug** view, either the code execution is automatically halted on a breakpoint, or the developer should pause the execution of the application by clicking on the **Suspend** button.
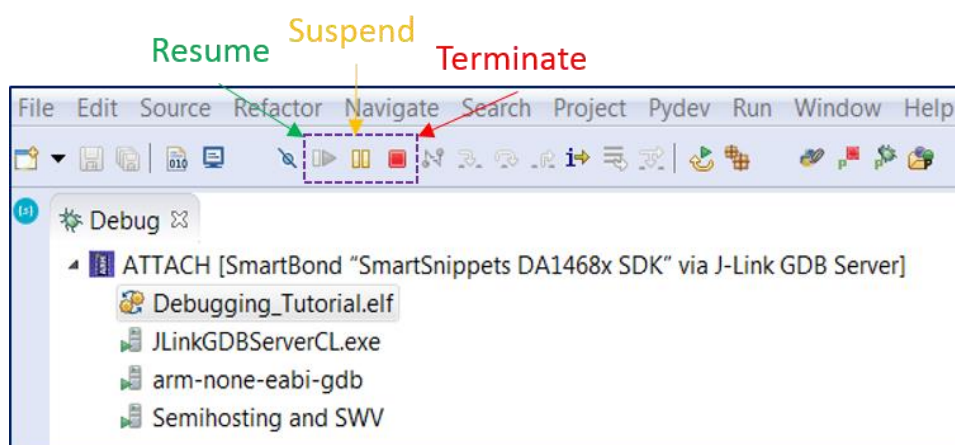
Figure 3: Resume, Suspend, and Terminate a Debug Session

2.2 Changing the Default Breakpoint

By default, a breakpoint is set on the main function of the application. However, things may go wrong before the code even gets to the main function. In this case, operation should be halted earlier, for instance in the reset handler. To do so, the following steps should be executed:

1. Click on the **Debug** button and select **Debug Configurations...**.

Debugging Techniques
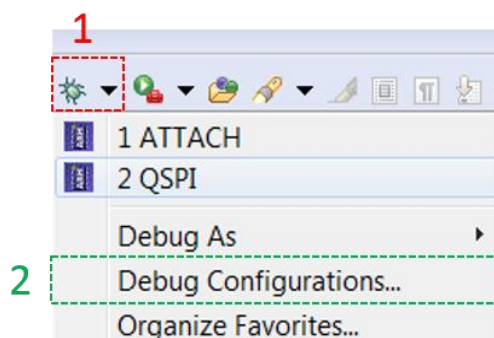
Figure 4: Changing the Default Breakpoint #1

The **Debug Configurations** window is displayed.

2. In the **Debug Configurations** window, select **SmartBond SmartSnippets DA1468x via J-Link GDB Server > QSPI**.

3. In the **Startup** tab, change the default breakpoint in the **Set breakpoint at** field and click on **Apply**. In this example, Reset_Handler is selected.
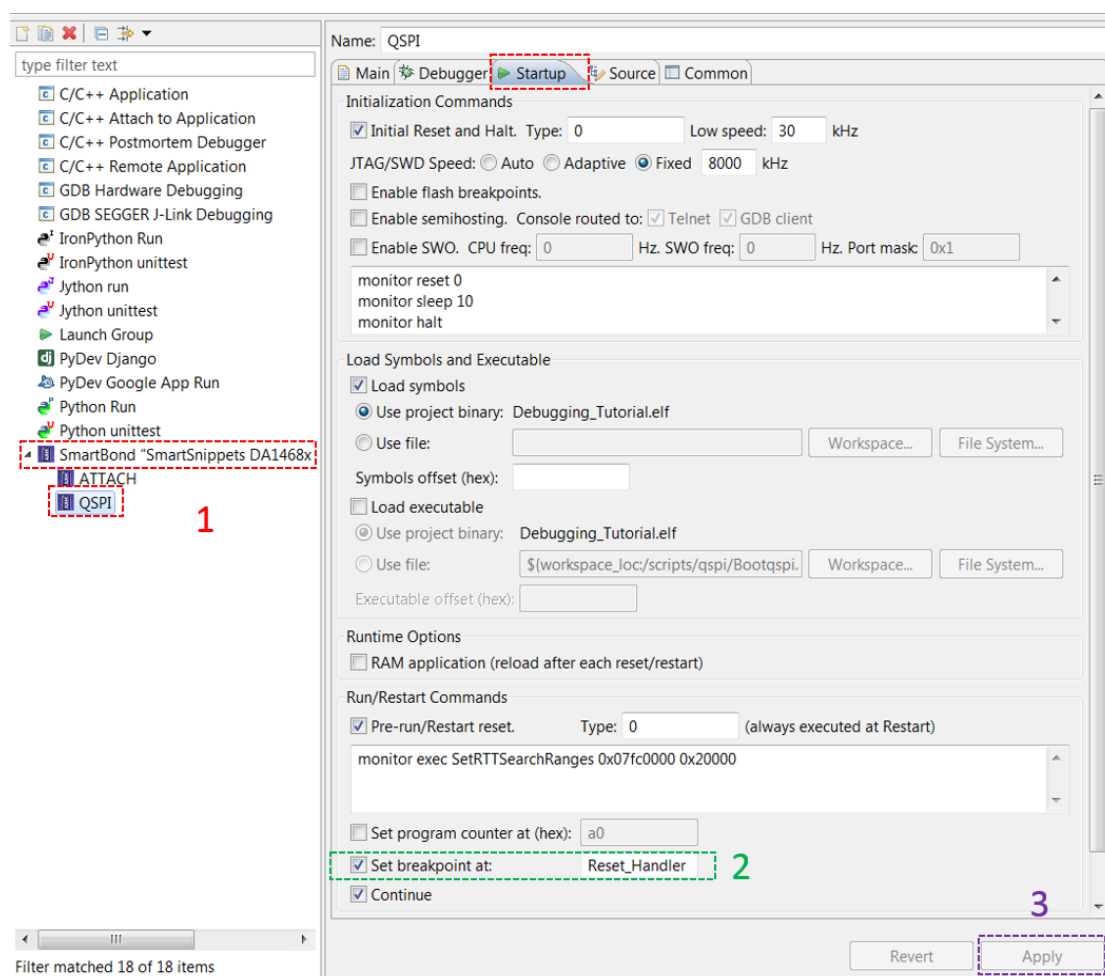
Figure 5: Changing the Default Breakpoint #2

© 2018 Dialog Semiconductor

Debugging Techniques

Note: The Cortex-M0 Breakpoint Unit (BPU) implementation provides between zero and four hardware breakpoint registers. A processor configured with zero breakpoints implements no breakpoint functionality. Typical hardware breakpoints watch an internal bus or the program counter and if it matches a certain condition it will either stop the processor or do whatever the hardware implements for that condition.

2.3   Device-Specific Registers

During a debugging session, the developer can read, as well as set, systems registers including GPIOs and other peripheral registers. This is done in the Eclipse tool named **EmbSys Registers**. To enable this tool, follow the steps below:

1.   From the **Window** menu, select **Show View > Other...**. The **Show View** window is displayed.

2.   In the **Show View** window, select **Debug > EmbSys Registers** (1) and then click **OK** (2).
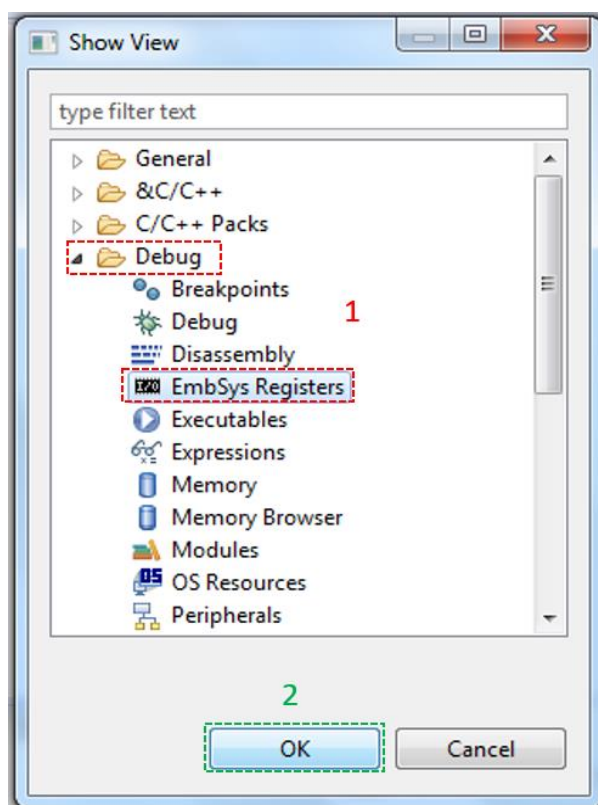
Figure 6: Show View Window

The **EmbSys Registers** window is displayed.

3.   To configure the tool, from the **Window** menu, select **Preferences**. The **Preferences** window is displayed.

4.   In the **Preferences** window, select **C/C++ > Debug > EmbSys Register View** (1) and then configure the tool as required (2).

      In the **Chip** field, select the correct family of devices (in our case DA14681-01). Finally click **OK** (3).

Debugging Techniques
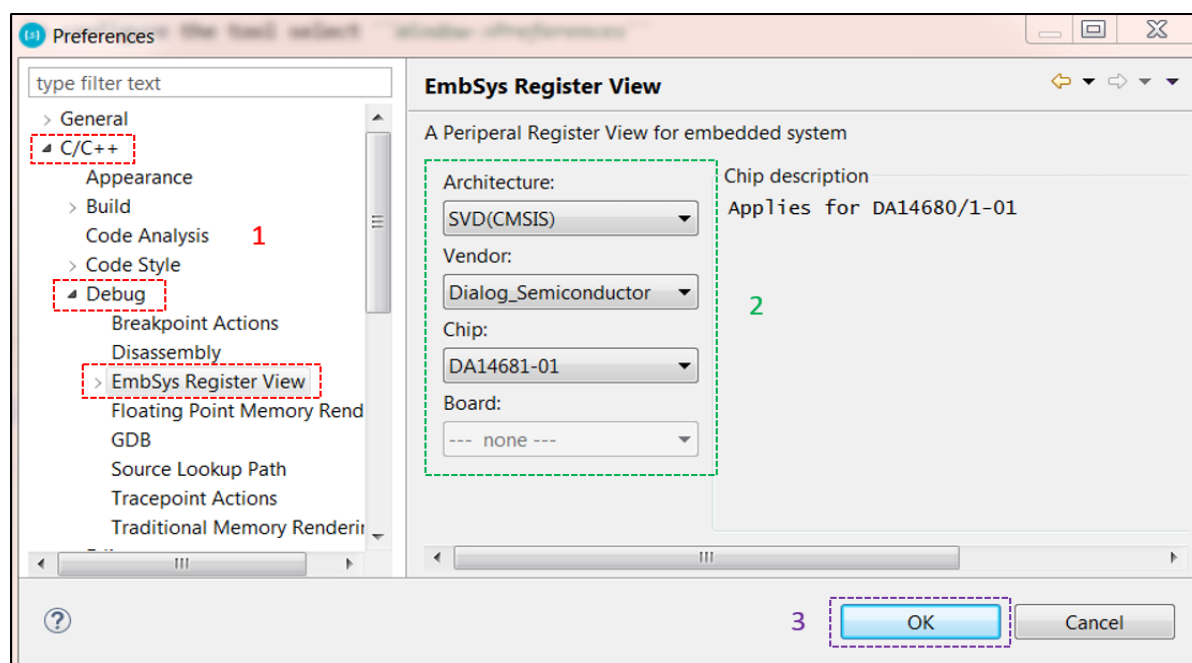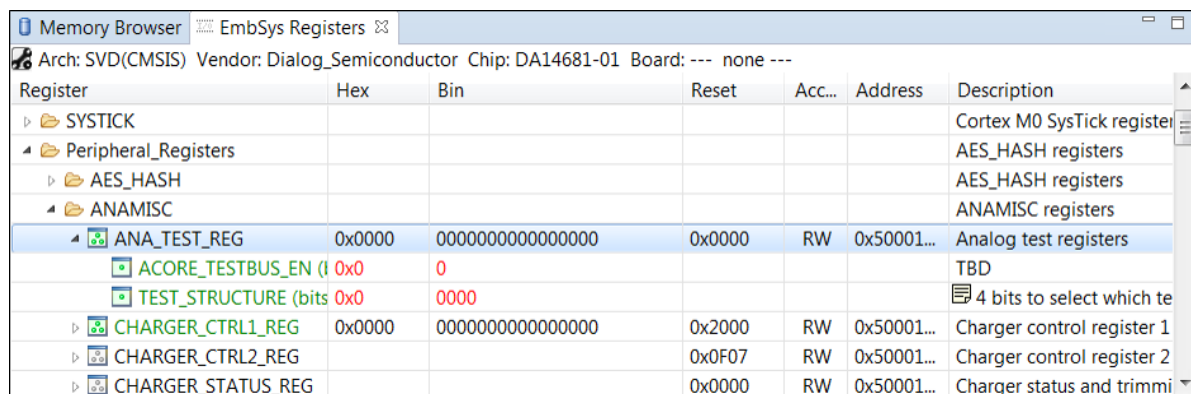
Figure 7: Configure the EmbSys Registers Tool

5. To read or modify register values, make sure the debugger is attached and suspended.

Figure 8: The EmbSys Registers View

2.4   Useful Debug Tools

The Eclipse IDE offers a variety of debugging tools which can be used during a debugging session. The tools are available through various windows which can be enabled/disabled by selecting them from the **Window > Show View** menu.

Debugging Techniques
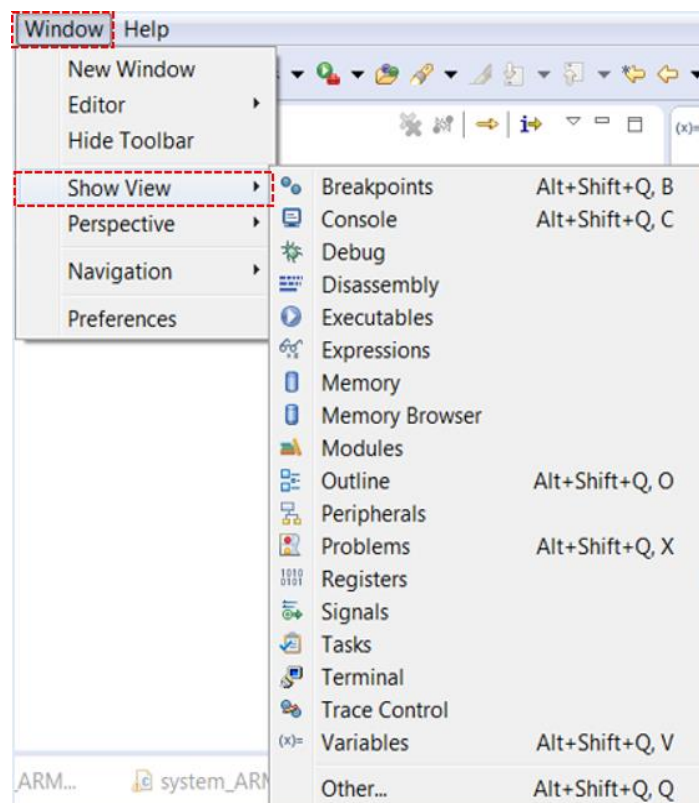
Figure 9: Selecting a Window

- The **Debug** window displays the currently running task's stack through the function call tree.
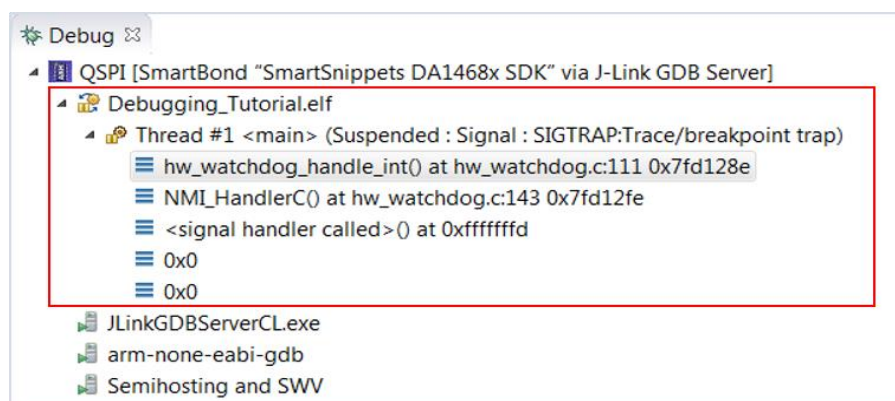
Figure 10: Stack Frame in the Debug Window

- The **Registers** window (1) displays the MCU registers.
- The **Variables** window (2) displays local variables.
- The **Expressions** window (3) displays:
    - variables residing in statically created variables
    - variables residing in heap using the variable location address
    - arrays of data and also complex structures of data
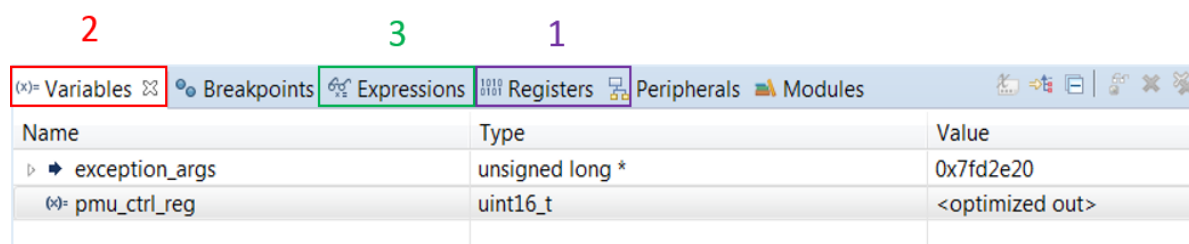
Figure 11: Useful Debug Windows

3   Hardfaults Session

This section provides a brief description of hardfaults on Cortex-M0 processors. It also describes the tools that can be used to deal with system faults.

It also explains how the SDK handles system faults and demonstrates a real use case of a hardfault, including all the steps that need to be followed to handle the fault.

3.1   Introduction

In ARM processors, when a program goes wrong and the processor detects the cause that made the device to fail, an exception is raised. On the Cortex-M0 processor integrated in the DA1468x family of devices, there is only one exception type that handles hardfaults. This is named "the hardfault handler". There are many reasons for a fault to occur, such as accessing invalid memory address during bus transaction or attempting to generate an unaligned memory access.

The way the SDK handles the various system faults depends on whether the application is built in **development** or **production** mode.

- In development mode (enabled by default), the SDK stores the system status in a predefined retained location in memory (SySRAM) and then adds an infinite loop. This allows the developer to attach a debugger, extract all the information stored in that memory area, and eventually identify the reason for the fault.

- In production mode, adding an infinite loop is not practical as it would require the user to get involved with debugging and recovery of the system. Instead, the system status is stored in a dedicated retained area (hard_fault_info) in SySRAM memory and a system reset is triggered so that the device starts its execution from start.

Debugging Techniques
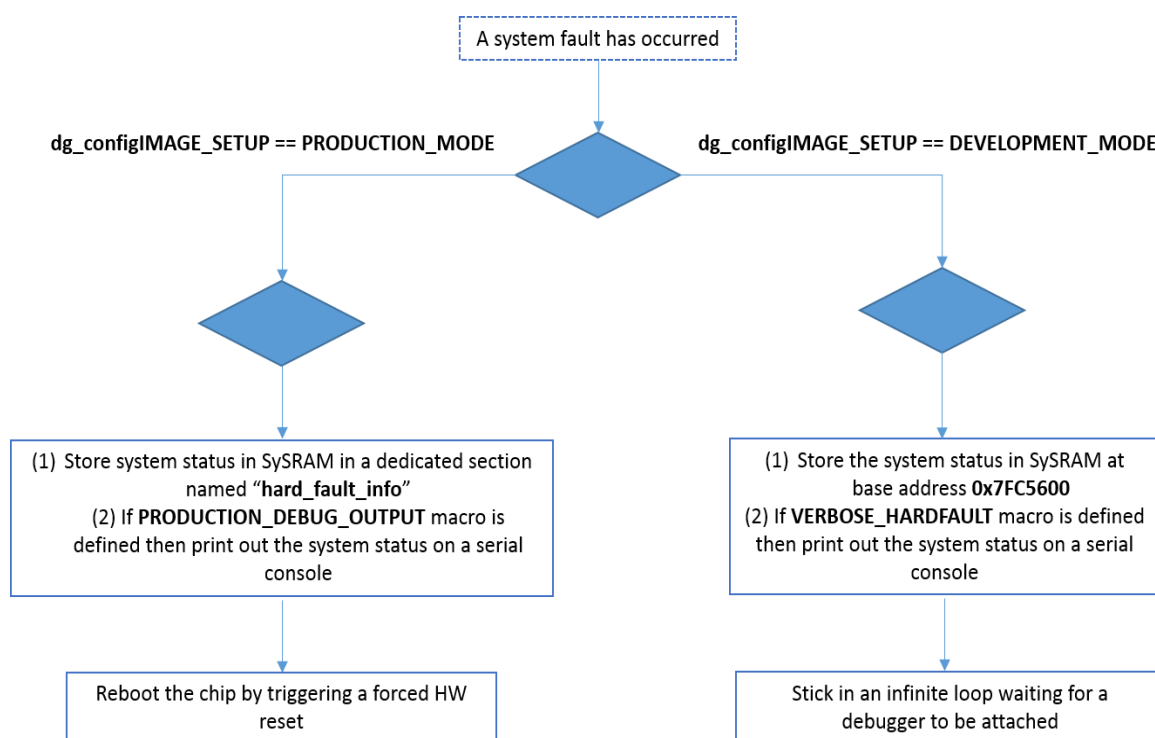
Figure 12: SW FSM of the Hardfault Exception Handler

Note: When in production mode and before executing the main application tasks, it can be determined whether or not a reset derives from a system fault by examining the information stored during a hardfault exception. For validity purposes, the first entry in the memory area, where the system status has been stored, should be 0xBADC0FFE. This entry can be considered a special flag to indicate that the data that follows is valid.

3.2   Manually Triggering a Hardfault

The following real use case demonstrates triggering a hardfault. The next section demonstrates how to identify the cause of the fault.

1.   Make a copy of the **freertos_retarget** sample code found in the SDK of the DA1468x family of devices. For information on how to create a new project, see *Create a New Project* in the Starting a Project tutorial.

2.   In the main.c source file of the newly created application, insert the following code which will trigger a hardfault. The function tries to access an invalid memory address, which triggers an exception to be issued when executed.

```
/* This is an invalid memory address - outside the recognized memory boundaries */
#define INVALID_ADDRESS 0x99999999UL;

void trigger_hardfault(void);
void trigger_hardfault(void)
{
    /* Declare a pointer that points to an invalid memory address */
    uint8_t *p = (uint8_t *) INVALID_ADDRESS;
```

Debugging Techniques

```
    /* Try to access that invalid memory address */
    *p = 0x50;
}
```

3. In the main task of the application, that is prvTemplateTask, call the trigger_hardfault() function somewhere within its main loop, for example:

```
/* Place this task in the blocked state until it is time to run again.
   The block time is specified in ticks, the constant used converts ticks
   to ms.  While in the blocked state, this task will not consume any CPU
   time. */
vTaskDelayUntil( &xNextWakeTime, mainCOUNTER_FREQUENCY_MS );


/* Trigger a hardfault deliberately! */
trigger_hardfault();

test_counter++;
```

4. Optionally, to enable debugging messages on the serial console, add the following macro definition in the config/custom_config_qspi.h header file.

```
/* Enable hardfault debugging messages */
#define VERBOSE_HARDFAULT          (1)
```

5. Build the project in **Debug mode** (for example, in the case of DA14681 SoC select the DA14681-01-Debug-QSPI build scheme) and burn the generated image to the chip.

Note: Debug mode is preferred over production mode when a debugging session is to be performed, as stepping the code is a more straightforward task. In production mode, the source code is built using optimizations, thus making tracing more complex.

6. Press the **K2** button on Pro DevKit. This starts the chip executing its firmware. After a while, the hardfault will be triggered.

3.3 Dealing with a Hardfault

This section provides the steps to identify the cause of a hardfault. Pointing to the exact location in the source code where a fault occurred, can be a tough task. However, a debugging session can reveal the point where things started to go wrong.

1. Initiate a debugging session by selecting the **ATTACH** mode. When switching to Debug view, select **Suspend** to pause the code execution.

Debugging Techniques
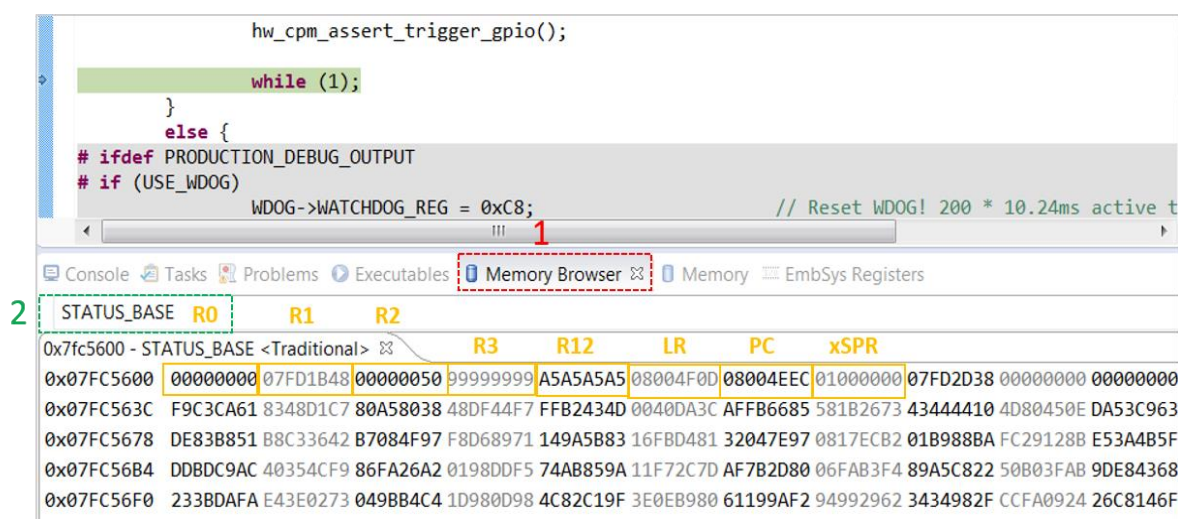
Program execution should be stuck in an infinite loop under the HardFault_HandlerC interrupt handler.

Figure 13: Hardfault Handler Function

The hardfault handler function provides the whole register set values when the hardfault was triggered. For example, the values of registers R0 to R3, R12, LR, PC, and xPSR are stored in memory position 0x07FC5600.

2. Use the **Memory Browser** tool (1) to view the contents stored in memory. In this tool, enter the base address where the stack frame is stored (2). Either enter the physical memory address value or the name of the corresponding macro, that is, STATUS_BASE.

Figure 14: Probing the Stack Frame Captured Following a Hardfault Event

If debugging output is enabled, the following information will be displayed on the console:

Debugging Techniques
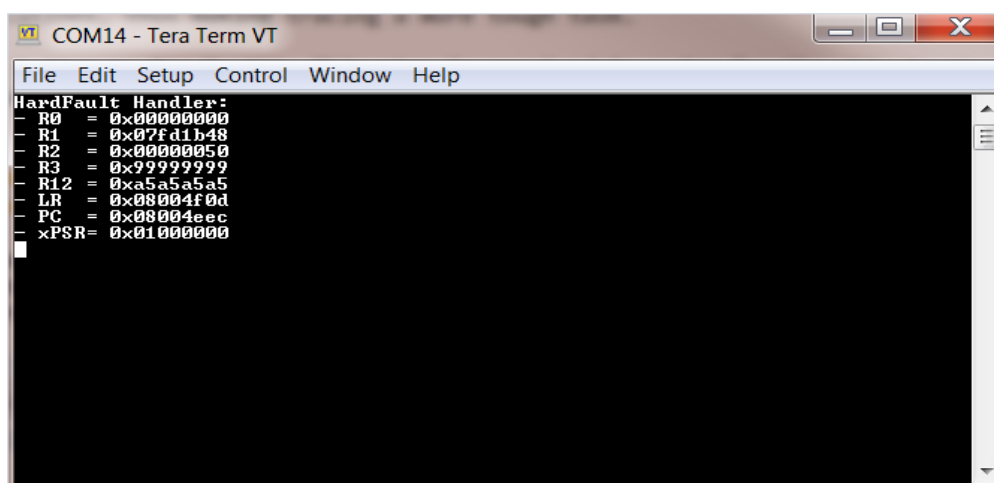
Figure 15: Debugging Messages Following a Hardfault Event

The most useful information is held in the Program Counter (PC) and Link Register (LR). The PC holds the current instruction address plus four bytes (this is caused by the pipeline nature of the Cortex-M0 processor). The LR is used for storing the return address of a subroutine or function call. At the end of the subroutine or function, the return address stored in LR is loaded into the PC so that the execution of the calling program is resumed.

This information, together with the **Disassembly** tool, can be used to identify the exact assembly command that caused the error.

3. At this stage, we can examine the command pointed to by the PC register value. To do this, select the **Disassembly** window (1), enter the value of the PC (2), and press **Enter**. Next locate the command pointed by the PC register (3) (displayed both in C and Assembly language).
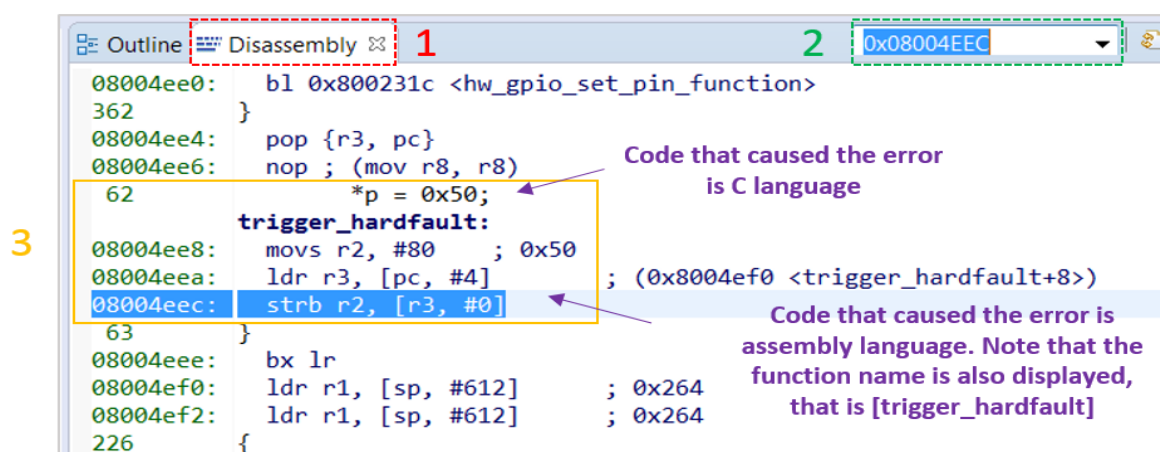
Figure 16: Probing the Contents of the Program Counter

4. Similarly, examine the instruction pointed by the Link register. Since the command that caused the fault is part of the trigger_hardfault() function, the LR should point to the instruction that will be executed upon the return of this function call.

Debugging Techniques

Figure 17: Probing the Contents of the Link Register

4 Reboot Analysis - BOD

This section provides a brief description of the Brown-out Detection (BOD) mechanism as implemented in the DA1468x family of devices. It describes the tools that can be used to deal with BOD events, and demonstrates a real use case of a BOD event, including all the steps that need to be followed for coping with the issue.

4.1 Introduction

Typically, an MCU includes a built-in Brown-out Detection (BOD) circuit, which monitors supply voltage levels during operation. Usually, BOD circuitry consists of comparators which constantly compare voltage levels against fixed trigger levels. As soon as a voltage level drops below a pre-defined threshold, a Power-on Reset (POR) may take place in order for the system to recover from power failure.
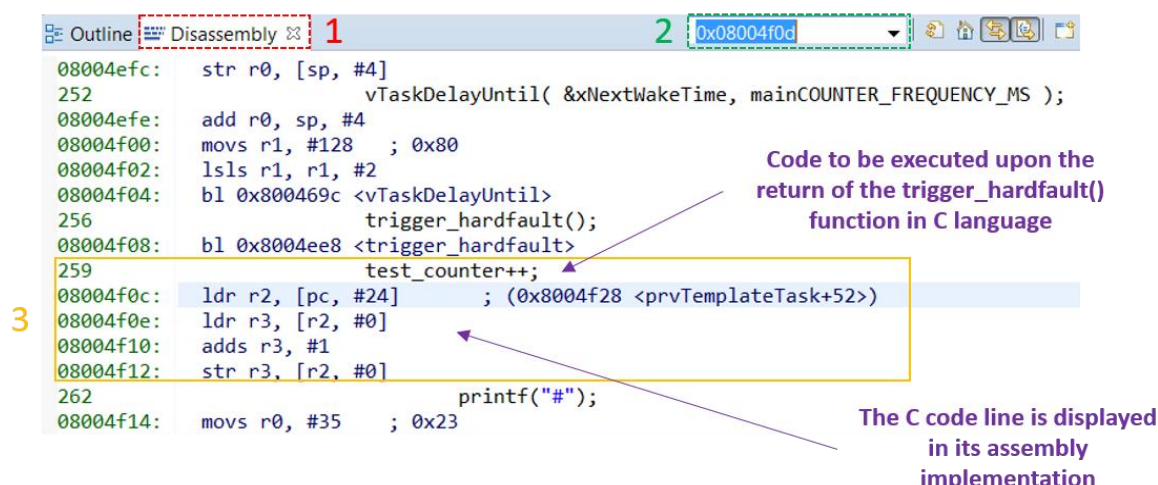
The DA1468x family of devices incorporates a BOD circuit which can be used for monitoring voltage levels derived from various power rails as illustrated in Table 1. Typically, from a software point of view, BOD functionality involves enabling the BOD mechanism and then monitoring a dedicated BOD status register.

Table 1: Power Rails of the SoC Monitored by the BOD Circuitry   Error! Bookmark not defined.

Power Rail	Description
1V8 power rail	This powers the externally connected Flash memory. The BOD mechanism for the rail is enabled by default. As soon as the voltage level drops below the predefined threshold, and given that BOD is enabled for the specific rail, a POR is issued.
1V8P power rail	This is intended for powering external devices even when the system is in sleep mode. This power rail is disabled by default. When enabled, the BOD protection for the rail is also enabled by the SDK. As soon as the voltage level drops below a predefined threshold, and given that BOD is enabled for the specific rail, a POR is issued.
VDD power rail	This powers the core itself. The BOD protection for the rail is disabled by default. The SDK does not have a macro to enable it. As soon as the voltage level drops below a predefined threshold, and given that BOD is enabled for the specific rail, a POR is issued.

Vsys power rail	This powers almost the whole system. The BOD protection for the rail is disabled by default. The SDK does not have a macro to enable it. As soon as the voltage level drops below a predefined threshold, and given that BOD is enabled for the specific rail, a POR is issued.
VBAT power rail	This is powered by the battery port. The BOD protection for the rail is enabled by default. As soon as the voltage level drops below a predefined threshold, a system interrupt is issued. This notifies the application that from this point and below, the DC-DC converter is not providing better efficiency than the LDOs. The application should switch the Power Management Unit (PMU) operation to the LDOs.

Note: The BOD status register, that is BOD_STATUS_REG, does not contain valid information for the power rails in which BOD protection is disabled. Also, the contents stored in this register are kept intact during a POR cycle.

4.2   Manually Triggering a BOD

The following real use case demonstrates triggering a BOD event on the 1V8P power rail and identifying that a reset was caused by that BOD event.

1. Make a copy of the **freertos_retarget** sample code found in the SDK of the DA1468x family of devices. For information on how to create a new project, see *Create a New Project* in the Starting a Project tutorial.

2. Enable the BOD mechanism by setting the following macro in the config/custom_config_qspi.h configuration file:

```
/* Enable WDOG */
#define dg_configUSE_WDOG          (1)
```

3. Enable the 1V8P power rail by setting the following macro in config/custom_config_qspi.h configuration file. After enabling it, its BOD protection is enabled by the SDK.

```
/* Power the 1V8P power rail */
#define dg_configPOWER_1V8P        (1)
```

4. The key goal of this exercise is for the developer to determine whether or not a reset was caused by a BOD event. This means that the BOD status should be monitored before the execution of the main application tasks. To do this, follow the steps below:

    a. In startup/system_ARMCM0.c, declare a variable to hold the contents of the BOD_STATUS_REG register.

```
__RETAINED_UNINIT uint32_t bod_status_reg_val;
```

Debugging Techniques

b. In the SystemInit function in startup/system_ARMCM0.c, get the contents of the
    BOD_STATUS_REG register just before enabling the BOD protection mechanism. After
    initializing the BOD mechanism, any previously stored information in the status register is
    invalidated.

```
/*
 * Initialize UNINIT variables.
 */
sys_tcs_init();

/*
 * Get the BOD status register before enabling the BOD protection
 */
bod_status_reg_val = CRG_TOP->BOD_STATUS_REG & 0x1F;

/*
 * BOD protection
 */
if (dg_configUSE_BOD == 1) {
    /* BOD has already been enabled at this point but it must be reconfigured */
    hw_cpm_configure_bod_protection();
} else {
    hw_cpm_deactivate_bod_protection();
}
```

c. In main.c, add the following variables:

```
/* This is an external variable declared in system_ARMCM0.c source file */
extern uint32_t bod_status_reg_val;

/*
 * This variable should be of type [volatile] since it will be reassigned
 * during the debugging session.
 */
static volatile bool bod_1v8p_flag = true;
```

d. In the system_init function in main.c, add the code that checks whether or not the system
    recovers from a BOD event on the 1V8P power rail:

```
/* Set the desired sleep mode. */
pm_set_sleep_mode(pm_mode_extended_sleep);

/*
 * Check whether or not the system recovers after a BOD event
 */
if (!(bod_status_reg_val & REG_MSK(CRG_TOP, BOD_STATUS_REG,
BOD_1V8_PA_LOW))) {
    /*
     * If yes, print a message on the serial console and then add an infinite
```

Debugging Techniques

```
        * loop so that the debugger can catch it! Then manually unblock!
        */
        printf("\n\n\rSystem has just recovered from a POR due to voltage drop on 1V8P
power rail\n\n\r");
        while(bod_1v8p_flag) {


            ;
        }
}


/* Start main task here (text menu available via UART1 to control application) */
OS_TASK_CREATE( "Template",        /* The text name assigned to the task, for
```

5. Increase the stack size reserved for the system_init initialization function by at least 100 bytes.

```
/* Start the two tasks as described in the comments at the top of this file. */
status = OS_TASK_CREATE("SysInit",         /* The text name assigned to the task, for
                                             debug only; not used by the kernel. */
        system_init,                       /* The System Initialization task. */
        ( void * ) 0,                      /* The parameter passed to the task. */
        (configMINIMAL_STACK_SIZE * OS_STACK_WORD_SIZE + 100),
                                           /* The number of bytes to allocate to the
                                             stack of the task. */
        OS_TASK_PRIORITY_HIGHEST,          /* The priority assigned to the task. */
        xHandle );                         /* The task handle */
OS_ASSERT(status == OS_TASK_CREATE_SUCCESS);
```

6. Build the project either in **Debug** or **Release** mode and burn the generated image in the chip.

7. Press the **K2** button on Pro DevKit. This step starts the chip executing its firmware.

8. Before triggering a voltage drop on 1V8P power rail, monitor the status of the BOD mechanism. To do this:

    a. Start a debugging session by selecting the **ATTACH** mode and then selecting **Suspend** to halt MCU operation.

    b. Select the **EmbSys Registers** window and navigate to **BOD_CTRL2_REG** and **BOD_STATUS_REG** registers under the **CRG_TOP** folder. Double-click on each register name to display the contents.

Debugging Techniques

This is the BOD entry of interest. A value set to [**1**] indicates that the voltage level on that rail is **above** the predefined threshold. A value set to [**0**] indicates that the voltage level on that rail is **below** the predefined threshold and a **POR** is issued if BOD protection for that power line is enabled.
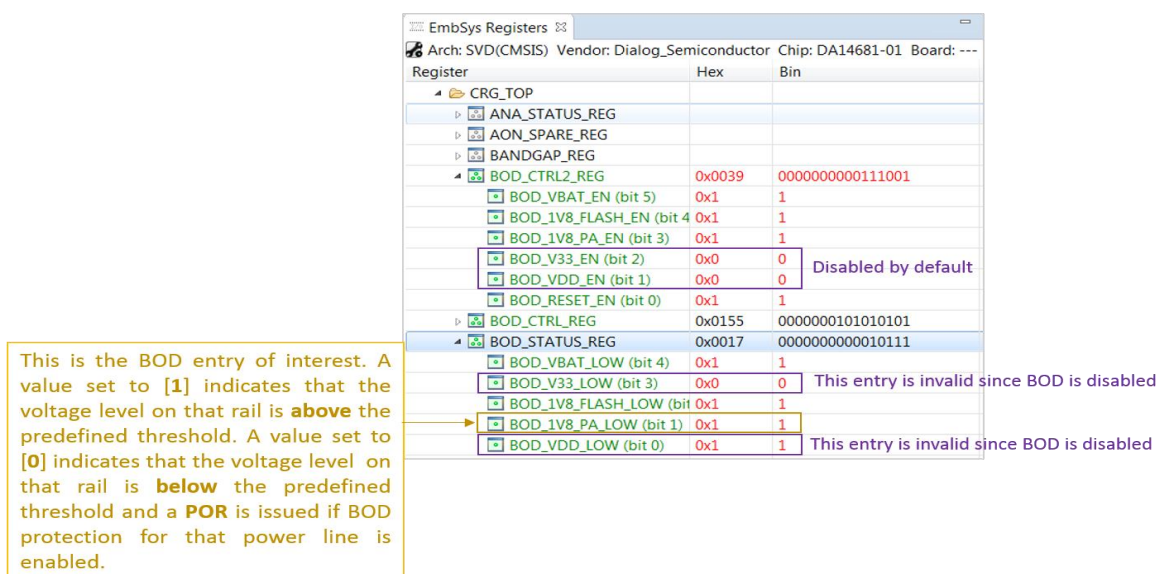
Figure 18: Inspecting BOD Related Registers

9. Select **Terminate** to terminate the current debugging session.

10. Open a serial terminal. For more information on configuring a serial terminal, see *Prepare the System* in the Starting a Project tutorial.

11. Shortcut the 1V8P power rail using a low value resistor (for example, 100 Ohm). The shortcut can also be done without using a resistor but it is not recommended as this may damage the chip.
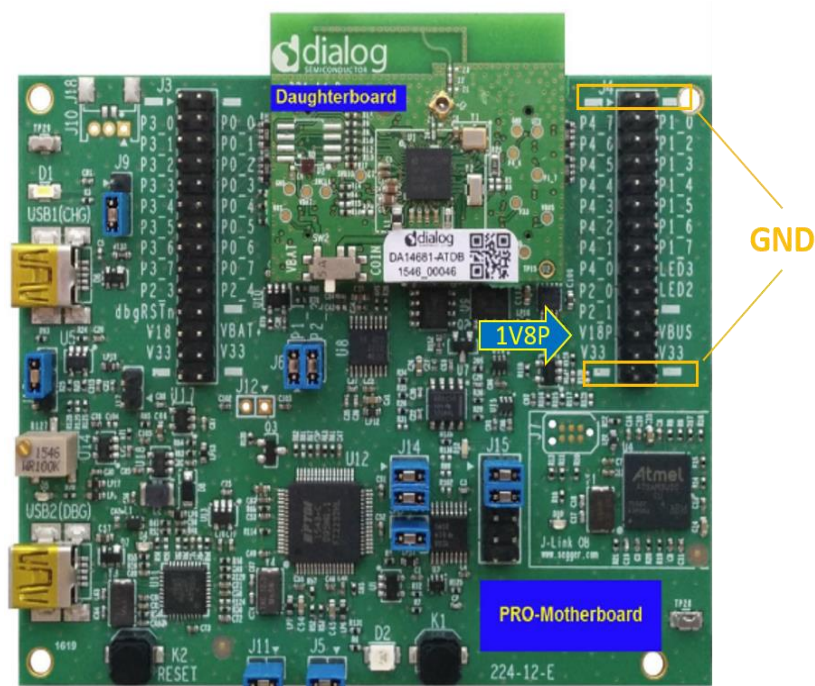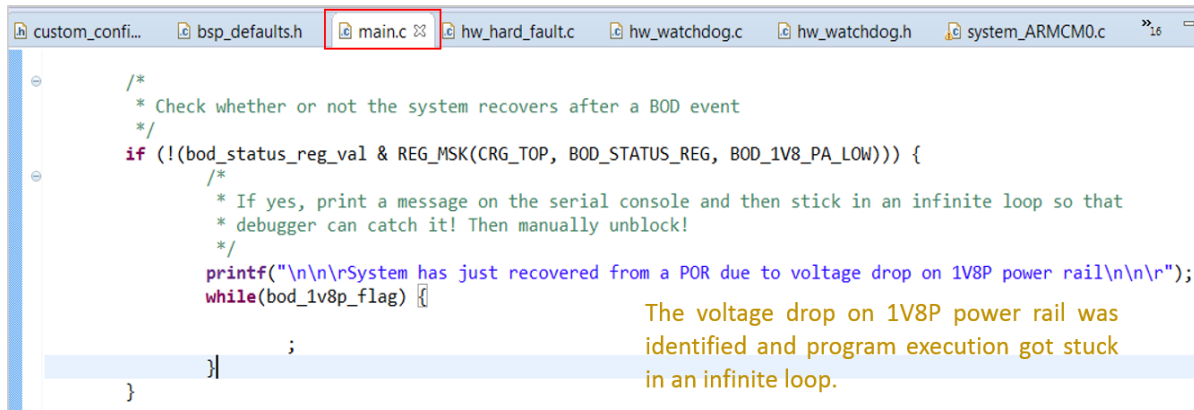

Figure 19: DA1468x Pro DevKit

4.3   Dealing with a BOD Event

1. Following the shortcut on the 1V8P power rail, the code should be in an infinite loop. To verify this, a new debugging session should be performed by selecting **ATTACH** mode and then **Suspend** to halt the MCU operation.

```
/*
 * Check whether or not the system recovers after a BOD event
 */
if (!(bod_status_reg_val & REG_MSK(CRG_TOP, BOD_STATUS_REG, BOD_1V8_PA_LOW))) {
    /*
     * If yes, print a message on the serial console and then stick in an infinite loop so that
     * debugger can catch it! Then manually unblock!
     */
    printf("\n\n\rSystem has just recovered from a POR due to voltage drop on 1V8P power rail\n\n\r");
    while(bod_1v8p_flag) {

        ;
    }
}
```

The voltage drop on 1V8P power rail was identified and program execution got stuck in an infinite loop.

Figure 20: Dealing with a BOD Event

2. Verify that the BOD_1V8_PA_LOW bit in the BOD_STATUS_REG register was set to zero by monitoring the bod_status_reg_val variable. To do this, in the **Expressions** window (1), click **Add new expression** (2) and write the variable name of interest (3).
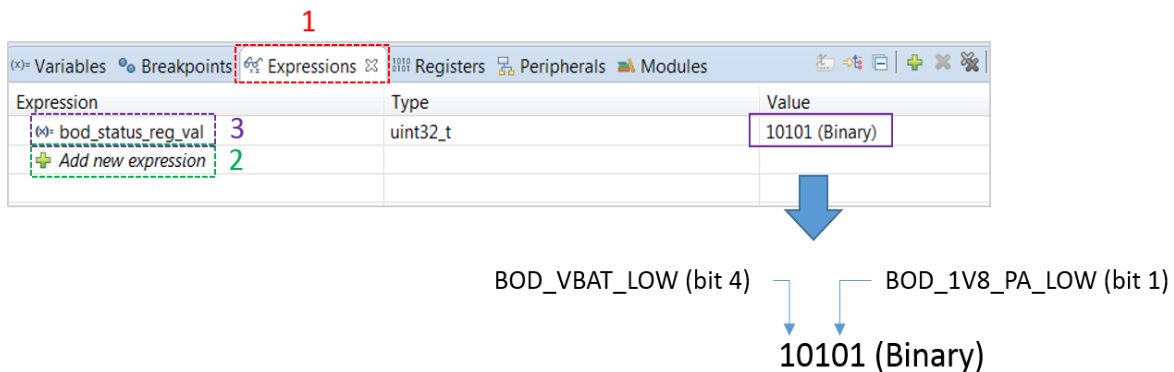
Expression	Type	Value
bod_status_reg_val 3	uint32_t	10101 (Binary)
Add new expression 2		

BOD_VBAT_LOW (bit 4)   BOD_1V8_PA_LOW (bit 1)

10101 (Binary)

Figure 21: Inspecting Variables and Expressions using the Expressions Window

Note: A voltage drop on a power line may also affect other power rails of the system.

3. The default presentation format for all variables is decimal. To change it, right-click on the variable name in the **Expressions** window (1), select **Number Format** (2), and then select the preferred format (3).
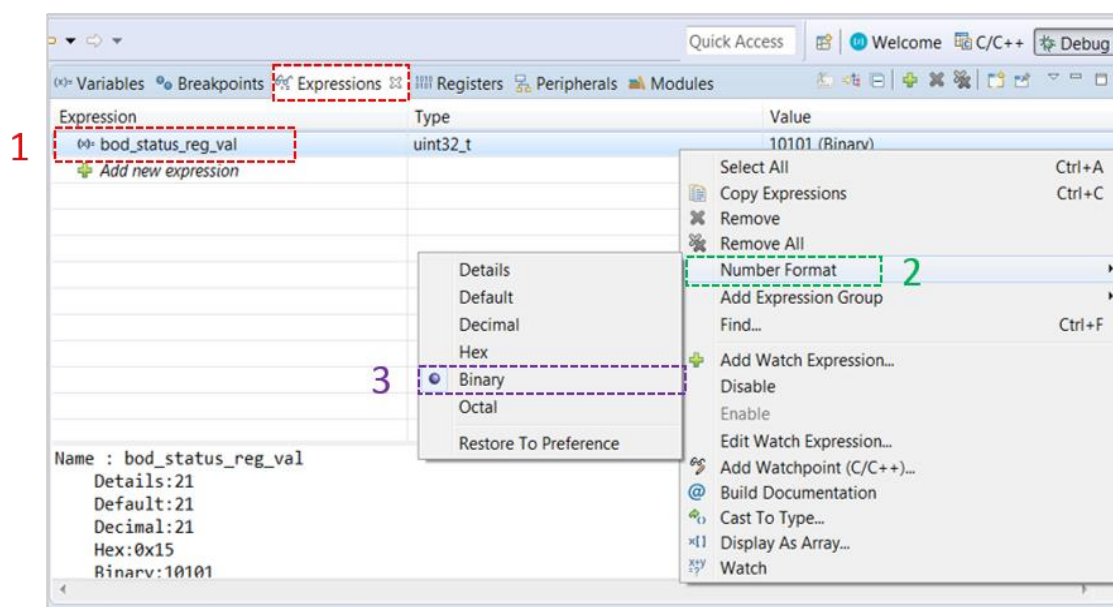
Debugging Techniques

Figure 22: Changing Number Presentation Formats

4. A debugging message, indicating that BOD mechanism trigged a POR cycle, should also be displayed on the serial console. This message could also be sent through BLE functionality using a custom BLE Profile.

This text is displayed by the main application task, that is the **prvTemplateTask**

This text is displayed by the Bootloader during a hardware reset using a 57600 baud rate and not 115200 which is the one set for the main application task
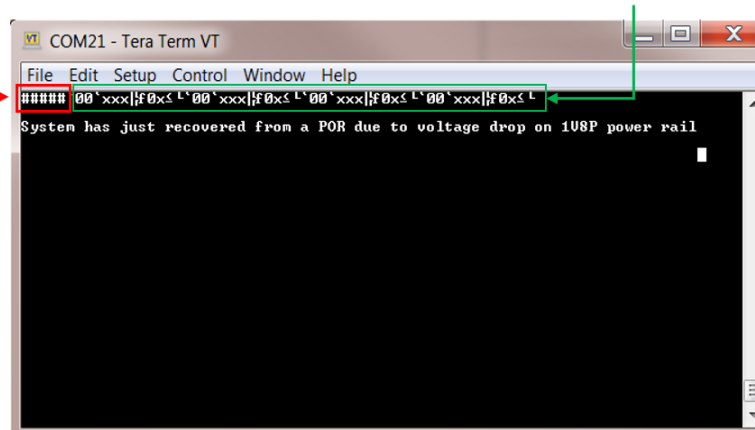

Figure 23: Printing Debugging Messages on the Serial Console upon a BOD Event

5. Normally, code execution will not stick in infinite loops, we only do this to verify various parameters related to a BOD event. To resume code execution the bod_1v8p_flag variable should be set to false. To do this, in the **Expressions** window (1), click on **Add new expression** (2) and enter the variable name of interest (3).
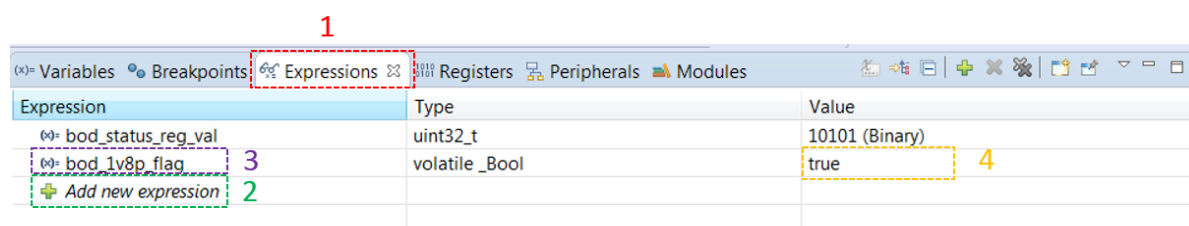
Figure 24: Inspecting Variables and Expressions using the Expressions window

6.   Click on the **Value** cell, in our case that is **true** (4) and change the value to **false**. The variable shoud now contain the newly declared value.
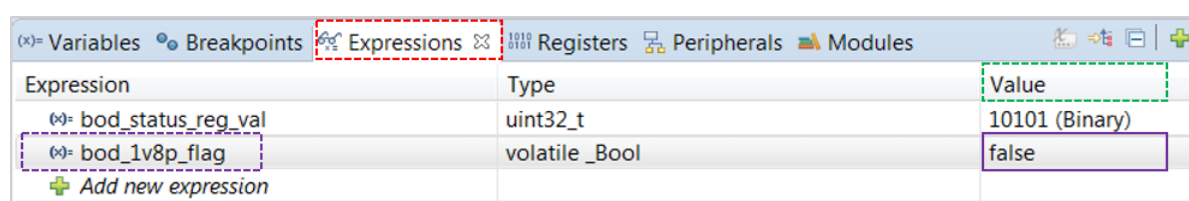

Figure 25: Modifying Variable Contents while in Debugging Session

Note: The volatile keyword indicates that a value may change between different accesses, even if it does not appear to be modified. In other words, it tells the compiler that the value of the variable may change at any time thus, not performing optimizations for that object. The system always reads the current value of a volatile object from the memory location rather than keeping its value in temporary core registers at the point it requested, even if a previous instruction asked for a value from the same object.

7.   Select **Terminate** to terminate the current debugging session. The code should resume its execution. To verify this, monitor the serial console. The # character should be printed out every 1 second.

5   Reboot Analysis - WDOG

This section provides a brief description of the watchdog (WDOG) mechanism as implemented in the DA1468x family of devices. It describes the tools that can be used to deal with watchdog exceptions, explains how the SDK handles watchdog exceptions, and demonstrates a real use case of a watchdog exception, including all the steps that need to be followed for copying with the exception.

5.1   Introduction

Most embedded systems need to be self-reliant. It's not usually possible to wait for someone to reboot them if the software hangs. A watchdog mechanism is a special hardware timer that can be used to automatically detect unexpected system behaviors during software execution. The DA1468x family of devices incorporates an 8-bit down counter driven by a 10.24 ms clock pulse, resulting in a maximum 2.6 seconds time-out. The embedded software selects the counter's initial value, by default this is set to 255, and periodically restarts it indicating that the application is up and running. If for any reason the firmware execution gets stuck, the watchdog timer is not updated and therefore expires after a time. Upon its expiration, and depending on the system configurations, either an NMI

Debugging Techniques

exception or a WDOG reset is issued to recover the system. By default, the system is configured so that an NMI interrupt is issued when the WDOG reaches a zero value.

Note: An NMI (Non Maskable Interrupt) is similar to an IRQ interrupt but it cannot be disabled by control registers and therefore its responsiveness is guaranteed.
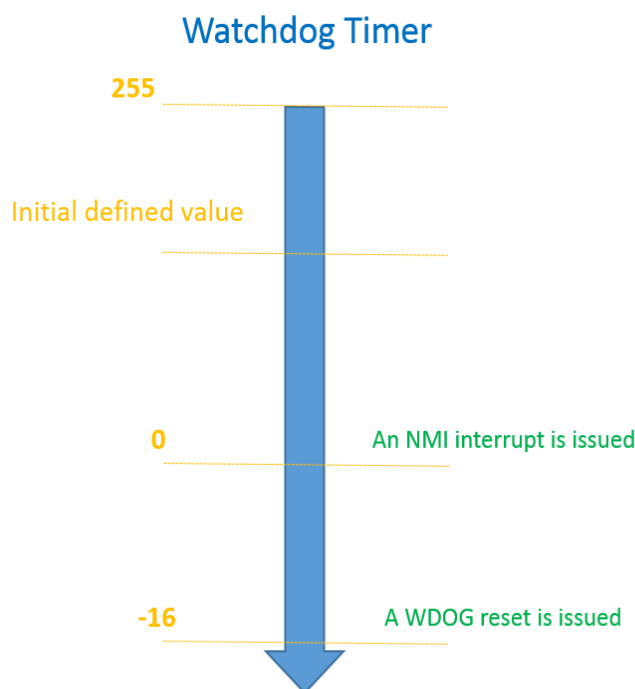
Watchdog Timer

255 — Initial defined value

0 — An NMI interrupt is issued

-16 — A WDOG reset is issued

Figure 26: Watchdog Functionality as Configured by Default

The way the SDK handles watchdog related events depends on whether the application is built in **development** or **production** mode.

- In development mode (enabled by default), the SDK freezes the watchdog operation, stores the system status in a predefined retained location in memory (SySRAM), and then halts CPU operation. This allows the developer to attach a debugger, extract all the information stored in that memory area, and eventually identify the reason for the fault.

- In production mode, halting system operation is not practical as it would require the user to get involved with debugging and recovery of the system. Instead, the system status is stored in a dedicated retained area (nmi_info) in SySRAM memory and after a while the watchdog mechanism triggers a hardware reset, which recovers the system.
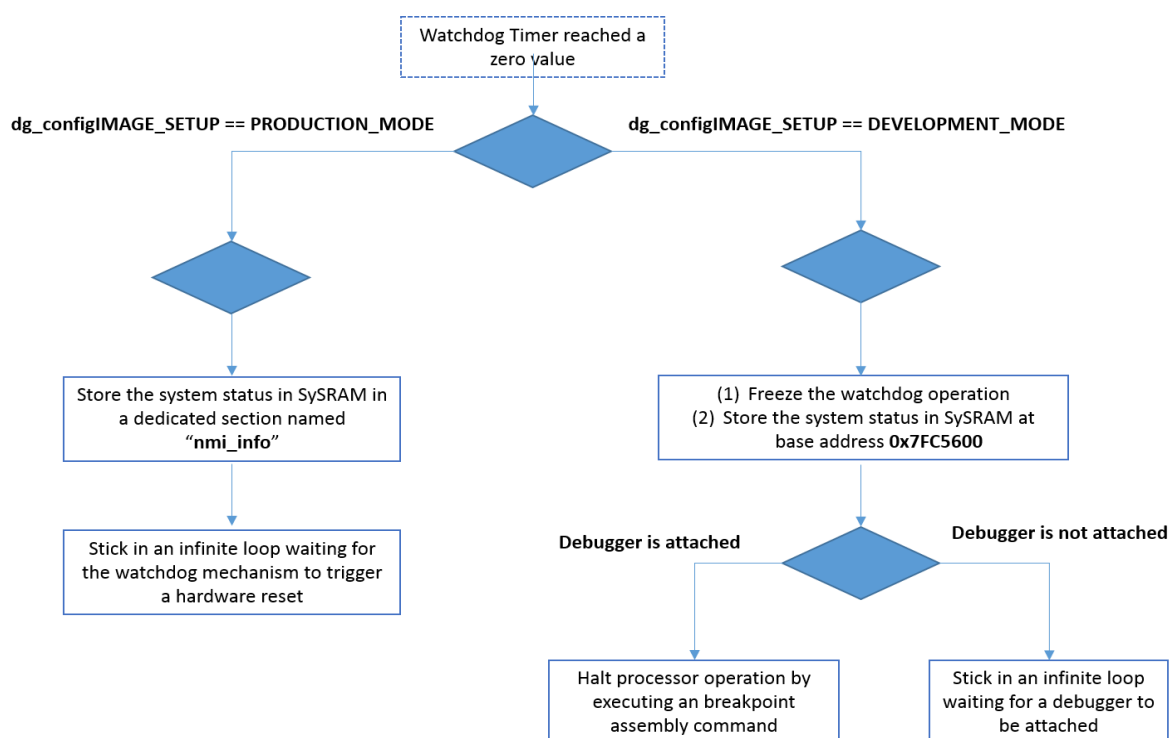
Debugging Techniques

Figure 27: SW FSM of the Watchdog Exception Handler

5.2 Manually Triggering a Watchdog Exception

The following real use case demonstrates expiring the watchdog timer and then identifying the cause of the fault.

1. Make a copy of the **freertos_retarget** sample code found in the SDK of the DA1468x family of devices. If this step has already been executed in previous sections do not repeat it. For information on how to create a new project, see *Create a New Project* in the Starting a Project tutorial.

2. In the main.c source file, insert the following code which triggers a watchdog expiration. This function traps the code execution in an infinite loop without updating the watchdog counter value. This results in the counter's expiration as soon as it reaches a value equal to zero.

```
void trigger_wdog(void)
{
    /*
     * Remain here until Watchdog Timer reaches a zero value and
     * an NMI interrupt is triggered.
     */
    for (;;) ;
}
```

3. In the main task of the application, prvTemplateTask, call the aforementioned function within its main loop, for instance:

Debugging Techniques

```
/* Place this task in the blocked state until it is time to run again.
   The block time is specified in ticks, the constant used converts ticks
   to ms.  While in the Blocked state this task will not consume any CPU
   time. */
vTaskDelayUntil( &xNextWakeTime, mainCOUNTER_FREQUENCY_MS );

/* Trigger a watchdog exception deliberately! */
trigger_wdog();

test_counter++;
```

4.   Enable the watchdog mechanism by declaring and setting the correct value to the dg_configUSE_WDOG macro. To do this, add the following macro definition in the config/custom_config_qspi.h header file.

Note: Before proceeding with this step check whether or not this macro has already been declared, so that to avoid duplicate declarations.

```
/* Enable WDOG */
#define dg_configUSE_WDOG            (1)
```

5.   If required, the developer can modify the initial WDOG counter value from 0xFF which is the default and the maximum allowable value. To do this, add the following macro definition in the config/custom_config_qspi.h configuration file.

```
#define dg_configWDOG_RESET_VALUE     XXXX
```

6.   Build the project in **Debug** mode (for the DA14681 SoC this is done by selecting the DA14681-01-Debug-QSPI build scheme) and burn the generated image to the chip.

Note: Debug mode is preferred over Release when a debugging session is to be performed, as stepping the code is a straightforward task. In Release mode, the source code is built using optimizations, thus making tracing a more complex task.

7.   Press the **K2** button on Pro DevKit. This step starts the chip executing its firmware. After a while, the watchdog will expire.

5.3   Dealing with a Watchdog Event

This section provides the steps required to identify the cause of the fault.

Debugging Techniques

1. Initiate a debugging session by selecting the **ATTACH** mode. Upon switching to Debug view pause the code execution by selecting **Suspend**.
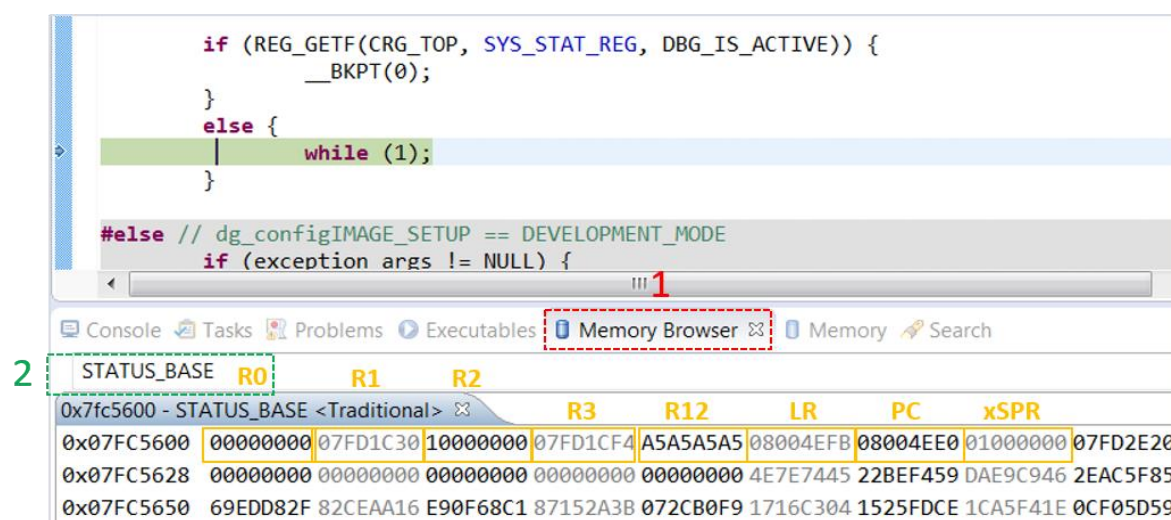
   Program execution should now be stuck in an infinite loop under the hw_watchdog_handle_int NMI handler.

```
hw_cpm_assert_trigger_gpio();

if (REG_GETF(CRG_TOP, SYS_STAT_REG, DBG_IS_ACTIVE)) {
        __BKPT(0);
}
else {
        while (1);
}
```

Figure 28: Watchdog Handler Function

2. The watchdog handler function provides the whole stack frame when the watchdog timer expired. The values of registers R0 to R3, R12, LR, PC, and xPSR are stored in memory position 0x07FC5600. To view the contents stored in memory, select the **Memory Browser** tool (1) and enter the base address where the stack frame is stored (2). Enter either the physical memory address value or the name of the corresponding macro, that is STATUS_BASE.

Figure 29: Probing the Stack Frame Captured upon a Watchdog Event

3. The most useful information is held in the Program Counter (PC) and Link Register (LR). Together with the **Disassembly View** tool, this can be used to identify the exact assembly command that caused the error.

   To examine the command pointed to by the PC register value, select the **Disassembly** window (1), enter the value of the PC (2), and press enter. Then locate the command pointed to by the PC register (3) (displayed both in C and Assembly language).

Debugging Techniques
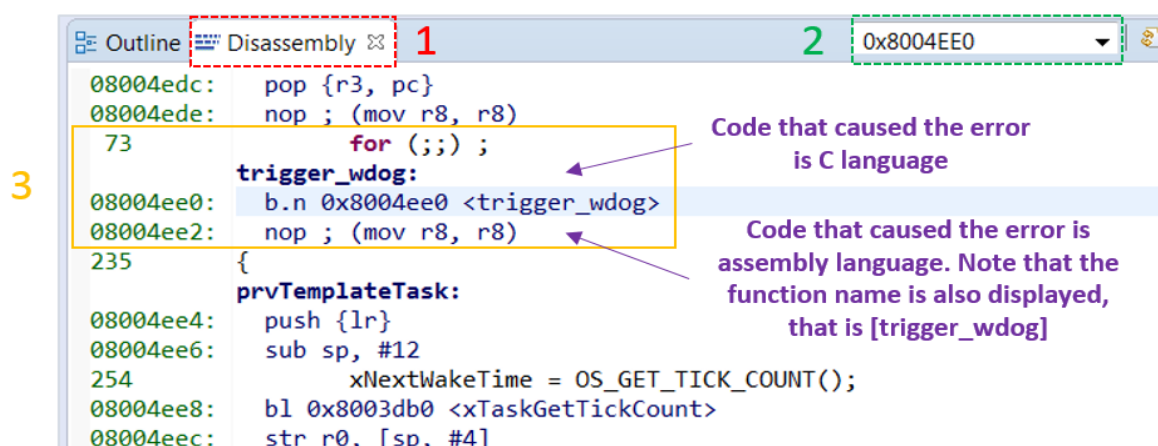
Figure 30: Probing the Contents of the Program Counter

4. Similarly, examine the instruction pointed by the Link Register. Since the code execution is trapped in an infinite loop within a function, the LR should point to that function.
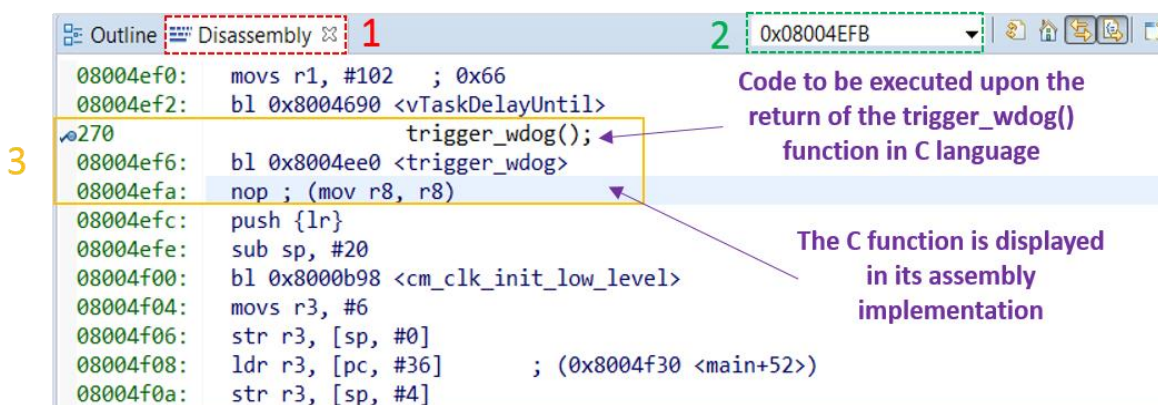

Figure 31: Probing the Contents of the Link Register

6   SW Cursor via Power Profiler

DA1468x SDK provides a debugging functionality which can be used to mark certain events and capture the corresponding pin levels with a logic analyzer. This feature can also be used for accurately measuring timing intervals. This feature can be used in combination with the Power Profiler tool for displaying the marked points of interest and measuring timing intervals.

1. If not already done so, make a copy of the **freertos_retarget** sample code found in the SDK of the DA1468x family of devices. For more information on how to create a new project, see *Create a New Project* in the Starting a Project tutorial.

2. To enable the SW cursor mechanism, set the appropriate macro in /config/custom_config_qspi.h:

```
#define dg_configUSE_SW_CURSOR (1)
```

3. This step is optional and is intended for those who are interested in using an external logic analyzer to capture the marked events. By default, P0.7 pin is the selected GPIO debugging

output. The developer can change the default pin by setting the appropriate macro definitions in config/custom_config_qspi.h. For example, to select pin P1.5, add the following macro definitions:

```
#define SW_CURSOR_PORT  (1)
#define SW_CURSOR_PIN    (5)
```

4. Call the following function at the point of interest. For example, modify the main task of the newly created project, that is prvTemplateTask, to mark an event every 200 ms.

```
for( ;; ) {
        /* Place this task in the blocked state until it is time to run again.
          The block time is specified in ticks, the constant used converts ticks
          to ms. While in the Blocked state this task will not consume any CPU
          time. */
        vTaskDelayUntil( &xNextWakeTime, mainCOUNTER_FREQUENCY_MS );

        test_counter++;

        /* Trigger the SW cursor */
        hw_cpm_trigger_sw_cursor();

        if (test_counter % (1000 / OS_TICKS_2_MS(mainCOUNTER_FREQUENCY_MS))
== 0) {
                printf("#");
                fflush(stdout);
        }
```

5. Build the project either in **Debug** or **Release** mode and burn the generated image to the chip.

6. Press the **K2** button on Pro DevKit. This step starts the chip executing its firmware.

7. Open SmartSnippets Toolbox and execute the following steps:

   a. Create a new project by selecting **New** (1). This step is optional if a project has already been created.

   b. Choose an available project (2).

   c. Choose a communication port (3).

   d. Select the family of devices that is being used (4).

   e. Open the selected project (5).

Debugging Techniques
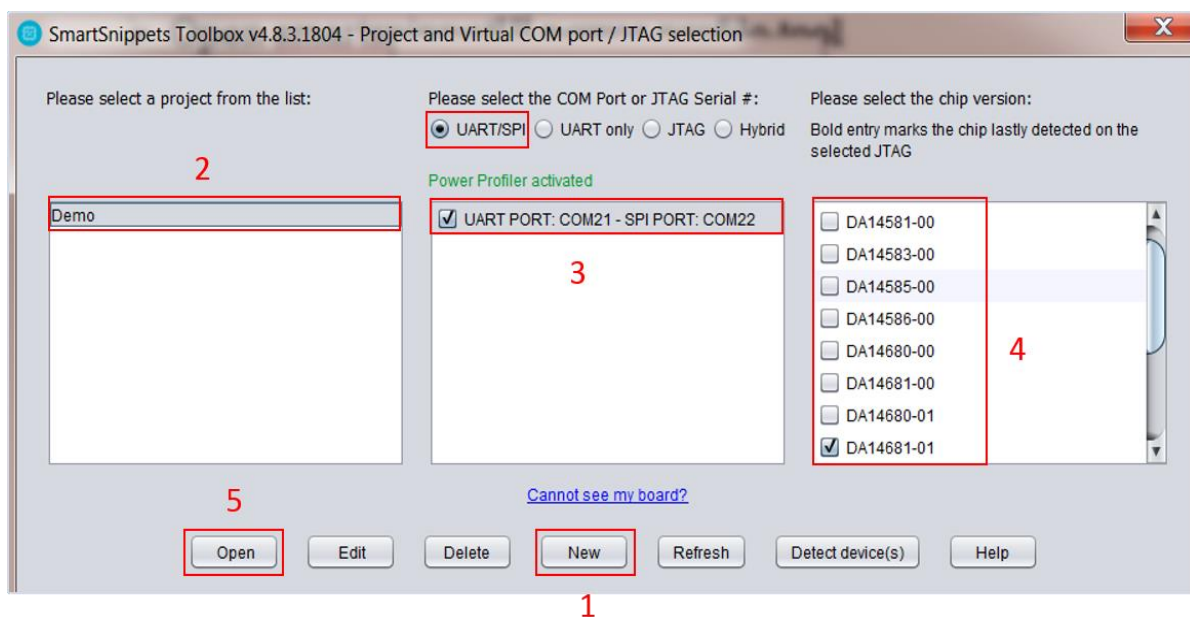
Figure 32: Opening SmartSnippets Toolbox

8. Start power profile monitoring:

   a. Switch to the **Power Profiler** window (1).

   b. Initialize Power Profiler (2).

   c. Start Power Profiler (3).
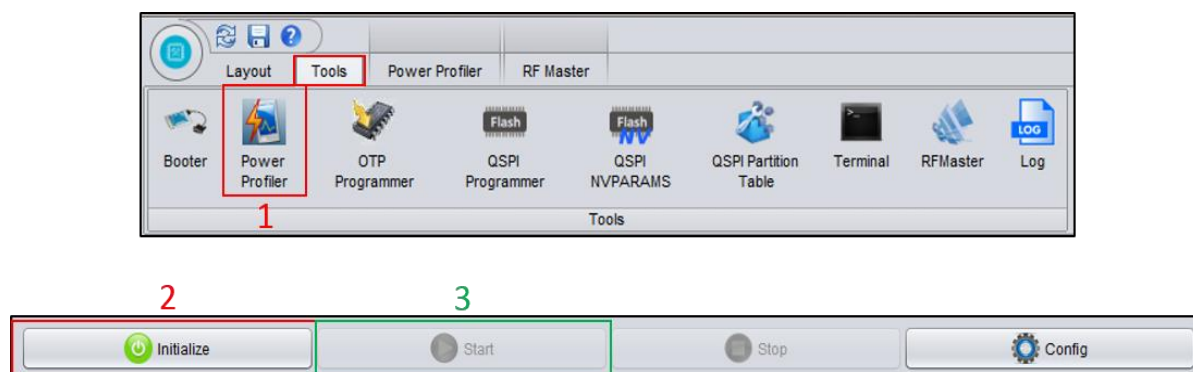
Figure 33: Initializing Power Profiler

9. Monitor the SW cursor indicated by a red line.

Debugging Techniques
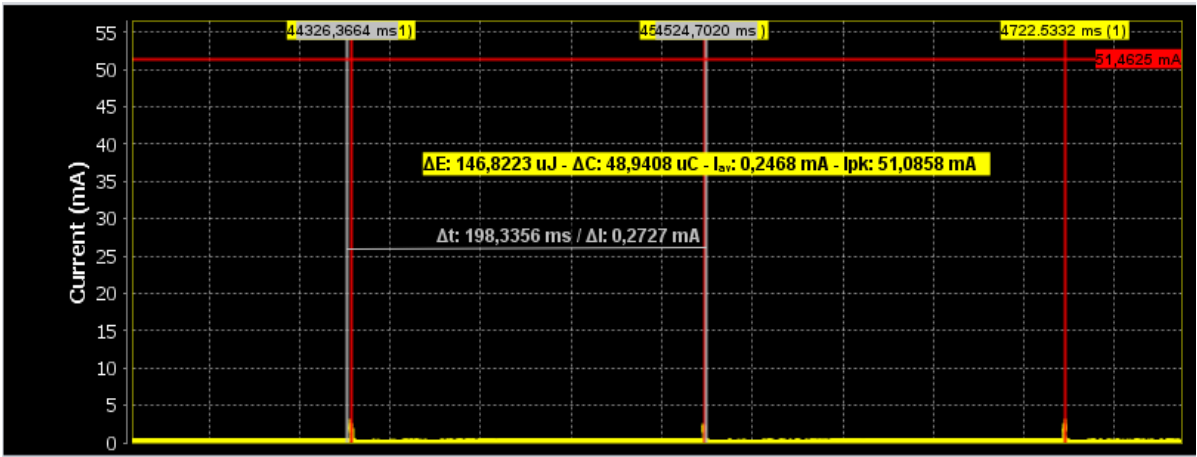
Figure 34: SW Cursor Indication

Revision	Date	Description
1.0	26-June-2018	First released version

Debugging Techniques

Status Definitions

Status	Definition
DRAFT	The content of this document is under review and subject to formal approval, which may result in modifications or additions.
APPROVED or unmarked	The content of this document has been approved for publication.

Disclaimer

Error! Bookmark not defined.

Contacting Dialog Semiconductor