Debugging with System Analyzer

Todd Mullanix

TI-RTOS Apps Manager

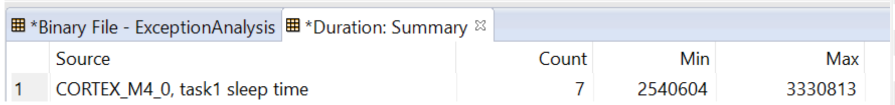
Oct. 15, 2017

TEXAS INSTRUMENTS

Abstract

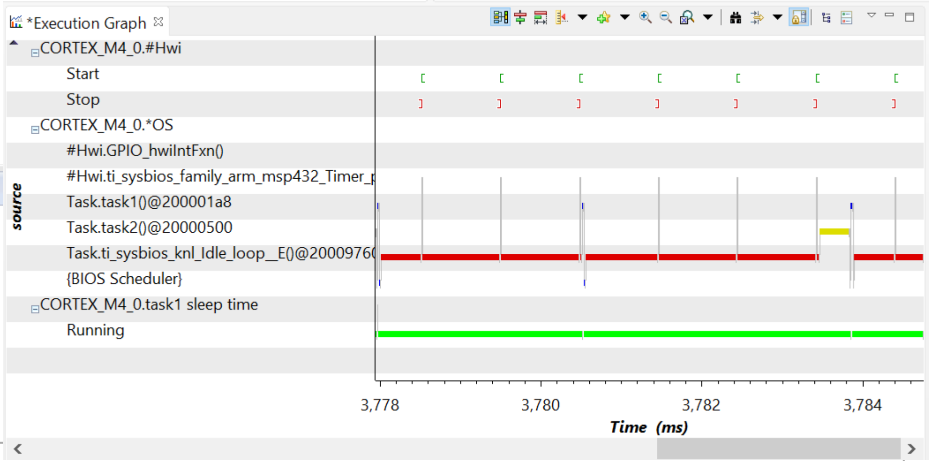
“In software engineering, tracing involves a specialized use of logging to record information about a program's execution.”[1]

This presentation will look at the software log capabilities in TI-RTOS and see how they can be used in conjunction with System Analyzer in CCS to debug an application. Specifically we’ll look at the Execution Graph and Duration Analysis to help to understand an application’s behavior.



*Binary File - ExceptionAnalysis | *Duration: Summary

	Source	Count	Min	Max
1	CORTEX_M4_0, task1 sleep time	7	2540604	3330813



[1]Wikipedia

Agenda

Here's the high-level view of what we will cover in this presentation:

1. Stopmode Example
2. Debugging with System Analyzer
   - Live View
   - Execution Graph
   - Duration Analysis
3. Logging Concepts
   - What is the Log module?
   - Logger Comparison
   - Diags Mask
4. Debugging with System Analyzer Post Mortem
   - Runtime Enabling/Disabling Logging
   - Generating a binary file and importing in System Analyzer
5. Appendix
   - Advanced Topics

Stopmode Example: Introduction

Let’s jump right into using System Analyzer to debug an application. We’ll call this application “stopmode” since we’ll use Logger_StopMode as the logging mechanism. We will not focus on the mechanics of the logging too much now. We’ll cover it more later.

The stopmode example has two tasks of different priorities. It also has a button callback that generates an exception. We’ll use System Analyzer to explain some delays and find the location of the exception.

The next slides will focus on the following

- Logging Configuration
- Exception Handling Overview
- Application Source Code Description

Stopmode Example: Logging Configuration

Here's the logging configuration in the app.cfg file. We'll use the LoggingSetup module. This module makes enabling logging easier and gets users up and running faster. We'll cover this module and more advanced capabilities later in this presentation.

- Include LoggingSetup and have two different log buffers to manage the different areas in the applicaton. Also enable Hwi logging.

```
var LoggingSetup = xdc.useModule('ti.uia.sysbios.LoggingSetup');
LoggingSetup.sysbiosHwiLogging = true;
LoggingSetup.loadLogging = false;
LoggingSetup.mainLoggerSize = 512;
LoggingSetup.sysbiosLoggerSize = 2048;
```

- Make sure the kernel has logging enabled.

```
BIOS.logsEnabled = true;
```

- Include these modules so we can make Log calls in the application.

```
var Log = xdc.useModule('xdc.runtime.Log');
var Diags = xdc.useModule('xdc.runtime.Diags');
var UIABenchmark = xdc.useModule('ti.uia.events.UIABenchmark');
```

Also, make sure the following is NOT a predefined compiler setting (SimpleLink CC2640R2 SDK does this in some examples).

```
xdc_runtime_Log_DISABLE_ALL
```

Stopmode Example: Exception Handling [cont.]

When Button0 is pushed an exception will occur. TI-RTOS has several types of exception handling options for CortexM devices*:

- TI-RTOS Enhanced Exception Decoding Handler
- TI-RTOS "Minimal" Exception Decoding Handler
- TI-RTOS Spin loop Handler
- User supplied Handler

For a detailed overview, please refer to https://training.ti.com/debugging-common-application-issues-ti-rtos.

The next slide will show the stopmode application's exception configuration and source code.

*Refer to the TI-RTOS Kernel documentation for details about the exception handling for your specific device.

Stopmode Example: Exception Handling

Here at the two pieces needed in the example for the exception handling…

- The source code which is responsible for plugging in the exception handler function in the `app.cfg` file.

```
m3Hwi.excHandlerFunc = "&myExceptionHandler";
```

- And here's the function in `main_tirtos.c`. For now, we're just going to turn on an LED to know that an exception has occurred and spin. We'll add content to this function later in the postmortem example.

```
Void myExceptionHandler(UInt *excStack, UInt lr)
{
    GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_ON);
    while(1);
}
```

Stopmode Example: Example Description

Here is the main gist of this very simple application…

- Task1 is a priority 1 (lowest) task which sleeps for 3 ticks. It uses the `UIABenchmark_start/stop` events. We look at these later in System Analyzer.

```
Void task1(UArg arg0, UArg arg1)
{
    while(1) {
        Log_write1(UIABenchmark_start, (IArg)"task1 sleep time");
        Task_sleep(3);
        Log_write1(UIABenchmark_stop, (IArg)"task1 sleep time");
    }
}
```

- Task2 is a priority 2 which burns some CPU to simulate some real action and then sleeps for 6 ticks.

```
Void task2(UArg arg0, UArg arg1)
{
    int i;
    volatile int dummy;
    int counter = 0;

    while(1) {
        Log_print2(Diags_USER1, "counter1 = %d (0x%x)", counter, counter);
        counter++;

        /* Burn some CPU to simulate some real action */
        for(i = 0; i < 1000; i++) {
            dummy *= i;
        }
        Task_sleep(6);
    }
}
```

- When pushed, Button0 causes an exception!

```
void gpioButtonFxn0(uint_least8_t index)
{
    asm("        .word 0x4567f123  "); // undefine instruction!
}
```

Stopmode Example: Running the Example

The example uses the MSP-EXP432P401R LaunchPad* and following software:

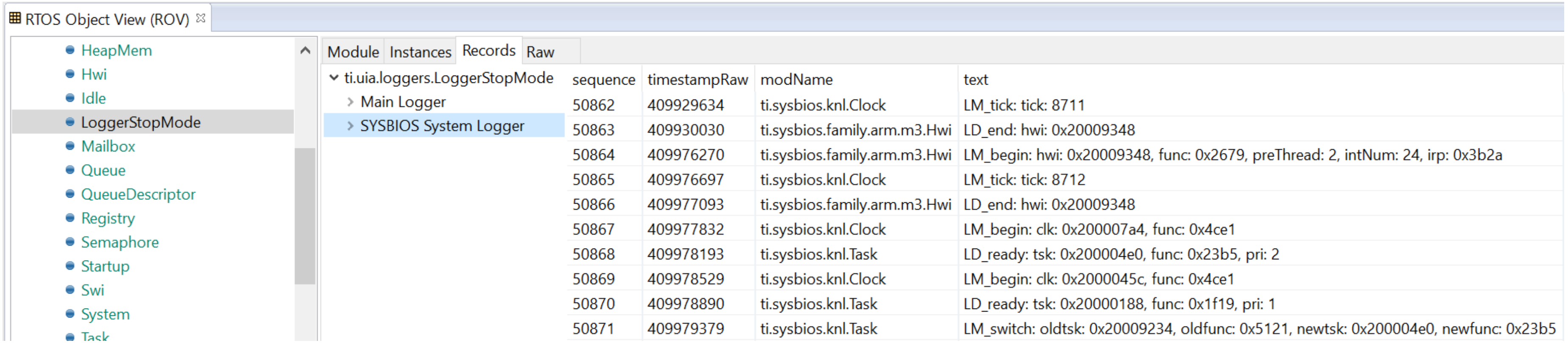
- CCS 7.3.0
- SimpleLink MSP432 v1.40.01.00
- XDCtools 3.50.02.20
- Compiler ARM version 16.9.4.LTS

- Please build, load, and run the example.
- After 5 or 10 seconds, hit Button0 to cause the exception. You should see `Board_GPIO_LED0` come on.
- Now let's go use System Analyzer to help determine what happened…

* We'll use a MSP-EXP432P401R LaunchPad as a concrete example, but the concepts and the code is exactly the same for all devices (minus pontentially the asm call to generate the exception).

Stopmode Example: But first let's confirm we have Log records

A quick way to make sure there are Log records is to open Tools->ROV->LoggerStopMode->Records. You should see records in both areas.



sequence	timestampRaw	modName	text
50862	409929634	ti.sysbios.knl.Clock	LM_tick: tick: 8711
50863	409930030	ti.sysbios.family.arm.m3.Hwi	LD_end: hwi: 0x20009348
50864	409976270	ti.sysbios.family.arm.m3.Hwi	LM_begin: hwi: 0x20009348, func: 0x2679, preThread: 2, intNum: 24, irp: 0x3b2a
50865	409976697	ti.sysbios.knl.Clock	LM_tick: tick: 8712
50866	409977093	ti.sysbios.family.arm.m3.Hwi	LD_end: hwi: 0x20009348
50867	409977832	ti.sysbios.knl.Clock	LM_begin: clk: 0x200007a4, func: 0x4ce1
50868	409978193	ti.sysbios.knl.Task	LD_ready: tsk: 0x200004e0, func: 0x23b5, pri: 2
50869	409978529	ti.sysbios.knl.Clock	LM_begin: clk: 0x2000045c, func: 0x4ce1
50870	409978890	ti.sysbios.knl.Task	LD_ready: tsk: 0x20000188, func: 0x1f19, pri: 1
50871	409979379	ti.sysbios.knl.Task	LM_switch: oldtsk: 0x20009234, oldfunc: 0x5121, newtsk: 0x200004e0, newfunc: 0x23b5

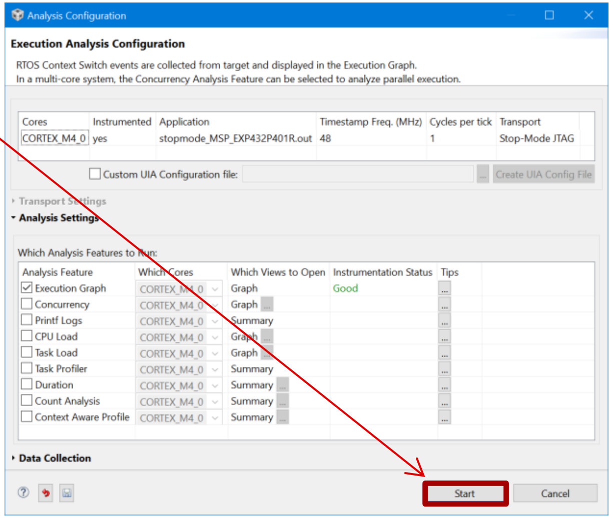
We use the term "Main" for non-kernel code. For example the `Log_write1()` in task1 will go into the "Main" logger. To be more exact, xdc.runtime.Main is used for all non-XDC modules.

Stopmode Example: Start System Analyzer

Let's open the Execution Graph now….

1. Tools->RTOS Analyzer->Execution Analysis. Note: RTOS Analyzer may have to initialize the first time.

2. Select "Start"



Execution Analysis Configuration

RTOS Context Switch events are collected from target and displayed in the Execution Graph.
In a multi-core system, the Concurrency Analysis Feature can be selected to analyze parallel execution.

Cores	Instrumented	Application	Timestamp Freq. (MHz)	Cycles per tick	Transport
CORTEX_M4_0	yes	stopmode_MSP_EXP432P401R.out	48	1	Stop-Mode JTAG

☐ Custom UIA Configuration file: Create UIA Config File

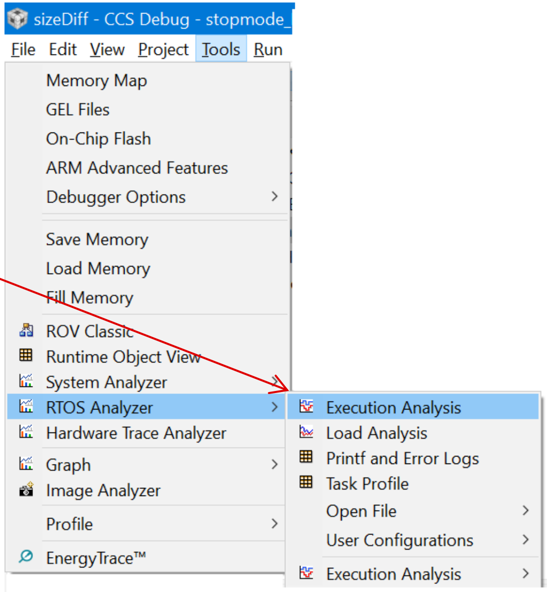
Transport Settings

Analysis Settings

Which Analysis Features to Run:

Analysis Feature	Which Cores	Which Views to Open	Instrumentation Status	Tips
✓ Execution Graph	CORTEX_M4_0	Graph	Good	...
☐ Concurrency	CORTEX_M4_0	Graph		...
☐ Printf Logs	CORTEX_M4_0	Summary		...
☐ CPU Load	CORTEX_M4_0	Graph		...
☐ Task Load	CORTEX_M4_0	Graph		...
☐ Task Profiler	CORTEX_M4_0	Summary		...
☐ Duration	CORTEX_M4_0	Summary		...
☐ Count Analysis	CORTEX_M4_0	Summary		...
☐ Context Aware Profile	CORTEX_M4_0	Summary		...

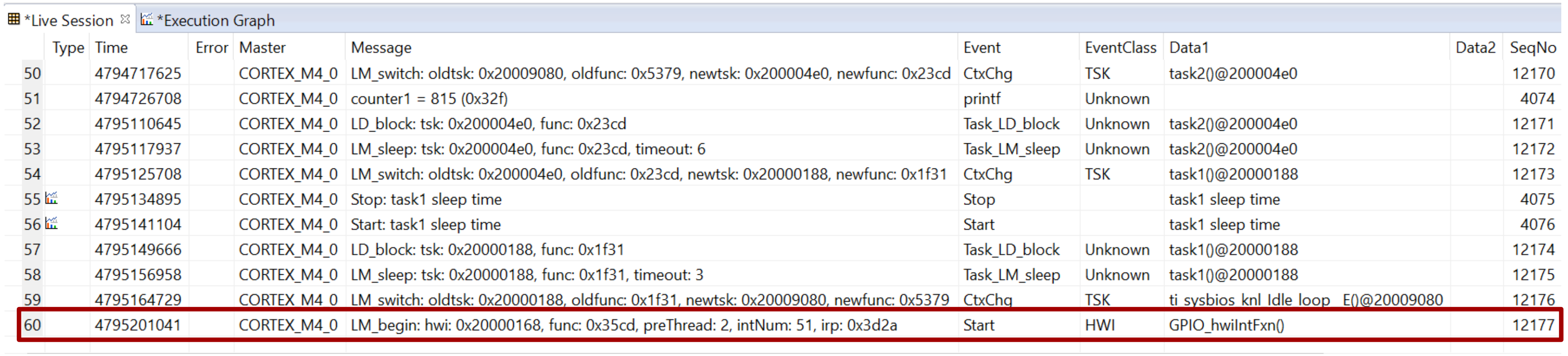
Data Collection



Please note: RTOS Analyzer is closely related to System Analyzer. For this presentation, we might use the term System Analyzer in a generic manner to mean either System Analyzer or RTOS Analyzer.

Stopmode Example: Live Session View

System Analyzer should display the Log records in the “Live Session” window.



*Live Session | *Execution Graph

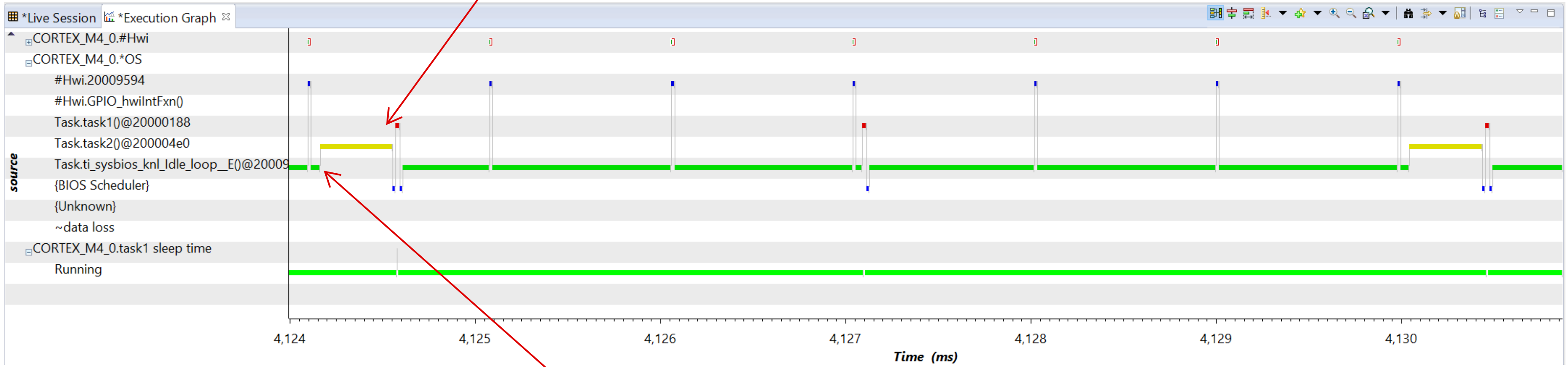
	Type	Time	Error	Master	Message	Event	EventClass	Data1	Data2	SeqNo
50		4794717625		CORTEX_M4_0	LM_switch: oldtsk: 0x20009080, oldfunc: 0x5379, newtsk: 0x200004e0, newfunc: 0x23cd	CtxChg	TSK	task2()@200004e0		12170
51		4794726708		CORTEX_M4_0	counter1 = 815 (0x32f)	printf	Unknown			4074
52		4795110645		CORTEX_M4_0	LD_block: tsk: 0x200004e0, func: 0x23cd	Task_LD_block	Unknown	task2()@200004e0		12171
53		4795117937		CORTEX_M4_0	LM_sleep: tsk: 0x200004e0, func: 0x23cd, timeout: 6	Task_LM_sleep	Unknown	task2()@200004e0		12172
54		4795125708		CORTEX_M4_0	LM_switch: oldtsk: 0x200004e0, oldfunc: 0x23cd, newtsk: 0x20000188, newfunc: 0x1f31	CtxChg	TSK	task1()@20000188		12173
55	📈	4795134895		CORTEX_M4_0	Stop: task1 sleep time	Stop		task1 sleep time		4075
56	📈	4795141104		CORTEX_M4_0	Start: task1 sleep time	Start		task1 sleep time		4076
57		4795149666		CORTEX_M4_0	LD_block: tsk: 0x20000188, func: 0x1f31	Task_LD_block	Unknown	task1()@20000188		12174
58		4795156958		CORTEX_M4_0	LM_sleep: tsk: 0x20000188, func: 0x1f31, timeout: 3	Task_LM_sleep	Unknown	task1()@20000188		12175
59		4795164729		CORTEX_M4_0	LM_switch: oldtsk: 0x20000188, oldfunc: 0x1f31, newtsk: 0x20009080, newfunc: 0x5379	CtxChg	TSK	ti_sysbios_knl_Idle_loop__E()@20009080		12176
60		4795201041		CORTEX_M4_0	LM_begin: hwi: 0x20000168, func: 0x35cd, preThread: 2, intNum: 51, irp: 0x3d2a	Start	HWI	GPIO_hwiIntFxn()		12177

Look at the last record and we see that the `GPIO_hwiIntFxn()` was the last thing running. When we look at the GPIO callback function, it makes sense we crashed.

```
void gpioButtonFxn0(uint_least8_t index)
{
    asm("         .word 0x4567f123  "); // undefine instruction!
}
```

Stopmode Example: Execution Graph

You can see that task2 runs before task1 when they are lined up (every 6 ticks) since it is higher priority.

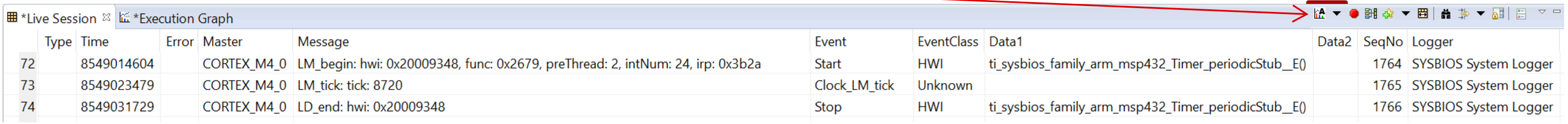


Note: Swi logging is not included to reduce the number of records. If it was enabled, you see the Clock tick Swi run after the Timer Hwi (instead of Idle task).

Please refer to the appendix to see how to get an execution graph over a longer period.

Stopmode Example: Duration Summary

You can select Duration View the drop-down.



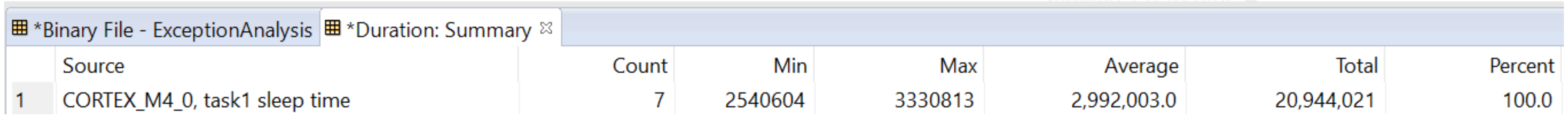
*Live Session | *Execution Graph

	Type	Time	Error	Master	Message	Event	EventClass	Data1	Data2	SeqNo	Logger
72		8549014604		CORTEX_M4_0	LM_begin: hwi: 0x20009348, func: 0x2679, preThread: 2, intNum: 24, irp: 0x3b2a	Start	HWI	ti_sysbios_family_arm_msp432_Timer_periodicStub__E()		1764	SYSBIOS System Logger
73		8549023479		CORTEX_M4_0	LM_tick: tick: 8720	Clock_LM_tick	Unknown			1765	SYSBIOS System Logger
74		8549031729		CORTEX_M4_0	LD_end: hwi: 0x20009348	Stop	HWI	ti_sysbios_family_arm_msp432_Timer_periodicStub__E()		1766	SYSBIOS System Logger

Remember the UIABenchmark events in Task1…

```
Void task1(UArg arg0, UArg arg1)
{
    while(1) {
        Log_write1(UIABenchmark_start, (IArg)"task1 sleep time");
        Task_sleep(3);
        Log_write1(UIABenchmark_stop, (IArg)"task1 sleep time");
    }
}
```

This is measuring the Task_sleep duration. You can see the max/min/avg in the Duration Summary.



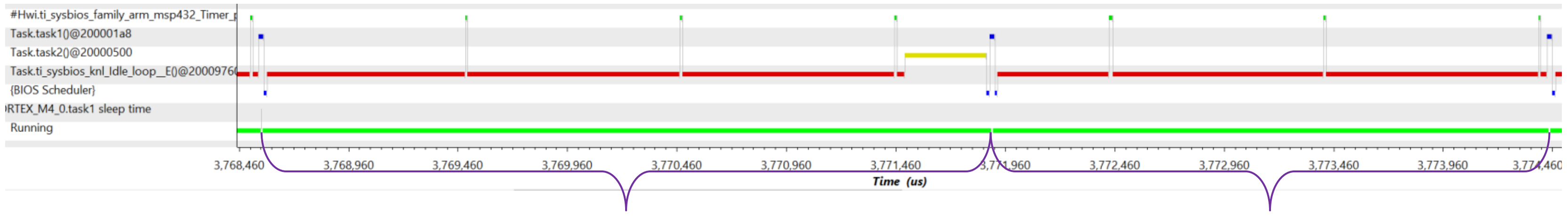
*Binary File - ExceptionAnalysis | *Duration: Summary

	Source	Count	Min	Max	Average	Total	Percent
1	CORTEX_M4_0, task1 sleep time	7	2540604	3330813	2,992,003.0	20,944,021	100.0

Why are the Min and Max so different?

Stopmode Example: Duration Summary [cont.]

Remember Task1 is lower priority than Task2. Task1 sleeps for 3 ticks and Task2 sleeps for 6 ticks. So every 6 ticks, both tasks will wake up. Since Task2 is higher priority, it will run first. Therefore Task1's execution is delayed. This can easily be seen in the Execution Graph.



Task1 is delayed so the duration is longer

Task2 was delayed so the duration is shorter

Let's look under the covers to see how we got all of this…

Logging Concepts: Introduction

TI-RTOS includes a Log module to aid in the generation of software log records. The goals of the Log module are

- Minimal runtime intrusion on the application (e.g. no ASCII management on the target).
- Usable by middleware software
- Able to compile out the Log API calls
- Can easily disable/enable logging during runtime
- Can be used in runtime and post mortem analysis
- Portable across all devices

When a Log API (e.g. Log_printN or Log_writeN) is called, a Log record is generated. Please note the Log record is relatively small (e.g. 20-64 bytes). It is up to a host tool to decode the binary data. The next slide will show this in more detail.

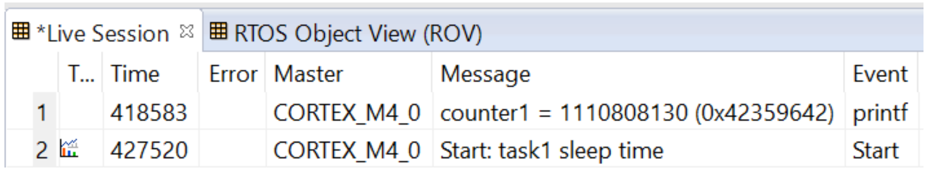
Logging Concepts: Log Records

Let’s look at the result of a couple Log API calls.

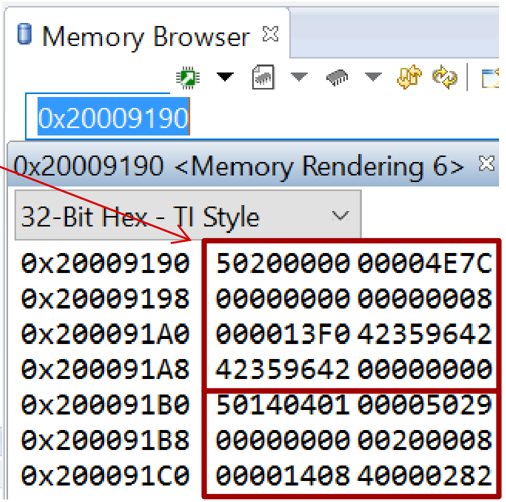
```
Log_print2(Diags_USER1, "counter1 = %d (0x%x)", counter, counter);
Log_write1(UIABenchmark_start, (IArg)"task1 sleep time");
```

These calls will generate two Log records in the internal LoggerStopMode buffer.

Then here are the decoded records in CCS

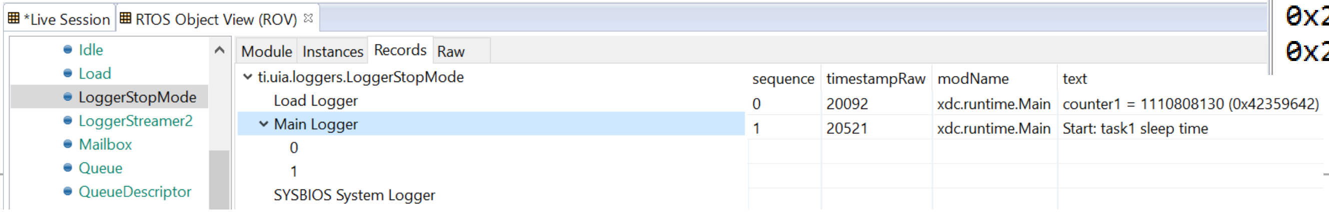


	T...	Time	Error	Master	Message	Event
1		418583		CORTEX_M4_0	counter1 = 1110808130 (0x42359642)	printf
2		427520		CORTEX_M4_0	Start: task1 sleep time	Start



```
0x20009190  50200000 00004E7C
0x20009198  00000000 00000008
0x200091A0  000013F0 42359642
0x200091A8  42359642 00000000
0x200091B0  50140401 00005029
0x200091B8  00000000 00200008
0x200091C0  00001408 40000282
```

And ROV

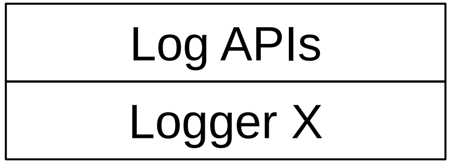


sequence	timestampRaw	modName	text
0	20092	xdc.runtime.Main	counter1 = 1110808130 (0x42359642)
1	20521	xdc.runtime.Main	Start: task1 sleep time

TEXAS INSTRUMENTS

Logging Concepts: Loggers

The Log module allows the application to select the underlying logger. There are several different types of loggers in TI-RTOS. This allows a software engineer to pick the one most optimal for their environment (comparison on the next slide).



A logger is plugged a module via the common$.logger field in the .cfg file. For example

```
Defaults.common$.logger = LoggerStopMode.create(loggerStopModeParams);
Task.common$.logger     = LoggerMin.create(loggerMinParams);
```

These lines plug in a LoggerMin instance for the Task module and all other modules will use the LoggerStopMode logger. Note: unless explicitly set, modules inherit from Defaults.

Note: LoggingSetup creates and assigns loggers automatically for you.

Logging Concepts: Logger Comparison

Here's a comparison of the most commonly used loggers in the UIA product (or in the kernel/ti/uia directories in the SimpleLink SDK).

Loggers	Description	Typical Use Cases
LoggerStopmode	Log records are stored in internal RAM buffer(s)	JTAG is connected and can halt the target.
LoggerMin	Log records are stored in an internal RAM buffer. Similar to LoggerStopmode but less features.	JTAG is connected and can halt the target. Post mortem analysis.
LoggerIdle	Log records are sent over UART or USBin the SYS/BIOS Idle Task.	No JTAG and/or prefer not to stop the target.
LoggerStreamer2	Log records are stored in a buffer supplied by the application.	Application wants to "own" the management of the Log record buffers.
LoggerRunMode (JTAG)	Log Records are uploaded over real-time JTAG. Available on C64+ and C66x targets only.	JTAG connected and cannot stop the target.
LoggerRunMode (Ethernet)	Events uploaded through ServiceMgr user-pluggable transport function.	Multi-core devices

For more details, please refer to the spruh43X.pdf document in the UIA or SimpleLink products.

Logging Concepts: Diags Mask

Too many log records is a common problem when logging. All Log calls have a “Diags” mask associated with it. For example:

```
Log_print2(Diags_USER1, "counter1 = %d (0x%x)", counter, counter);
```

The use of the Diags mask allows the Log call to be

- Compiled in or out
- Enabled/disabled during runtime

For example in the .cfg file, you can tell the TI-RTOS Task module to include the USER1 and not include USER2 Log calls.

```
Task.common$.diags_USER1 = xdc.module("xdc.runtime.Diags").ALWAYS_ON;
Task.common$.diags_USER2 = xdc.module("xdc.runtime.Diags").ALWAYS_OFF;
```

For the Task module, USER1 is used for high-level records (e.g. context switch) while USER2 is used for more detailed events (e.g. when a task calls Task_exit).

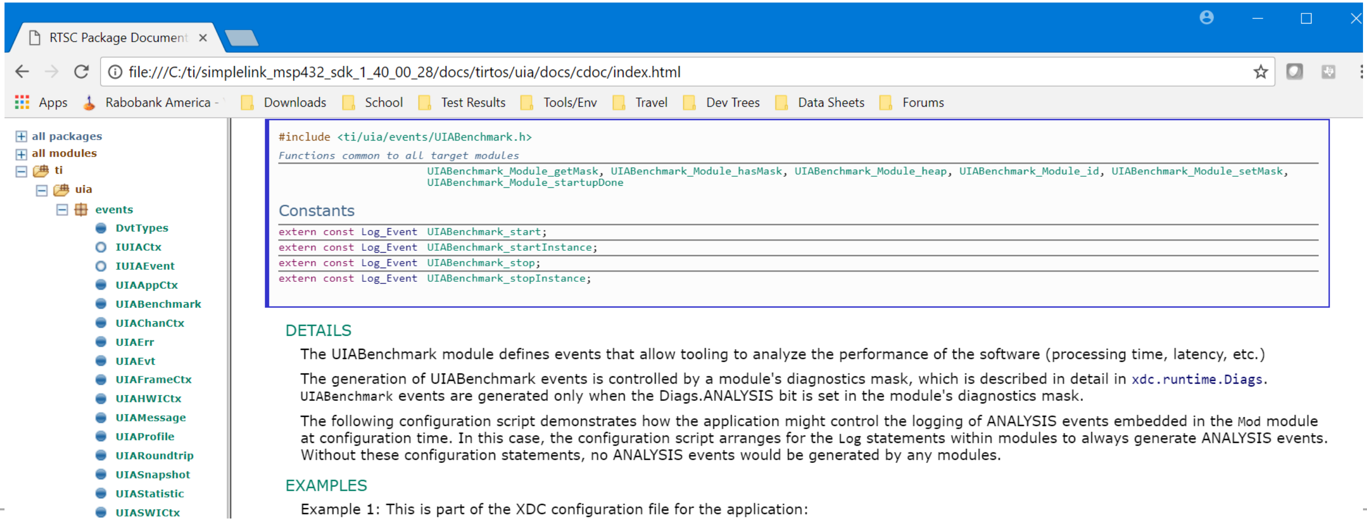
You can refer to the TI-RTOS cdoc for each modules usage of the Diags mask. The Diags module is also discussed here: http://rtsc.eclipse.org/cdoc-tip/xdc/runtime/Diags.html

Logging Concepts: Pre-Defined Events

The application can use pre-defined events also. For example the application uses UIABenchmark_start/UIABenchmark_stop events.

As we saw earlier, Duration Analysis will use these events to form all the stats.

How to use these pre-defined events is detailed in spruh43X.pdf in the UIA or SimpleLink products. More information is in the cdoc for each of the events also.



```
#include <ti/uia/events/UIABenchmark.h>
Functions common to all target modules
                         UIABenchmark_Module_getMask, UIABenchmark_Module_hasMask, UIABenchmark_Module_heap, UIABenchmark_Module_id, UIABenchmark_Module_setMask,
                         UIABenchmark_Module_startupDone
```

Constants

```
extern const Log_Event  UIABenchmark_start;
extern const Log_Event  UIABenchmark_startInstance;
extern const Log_Event  UIABenchmark_stop;
extern const Log_Event  UIABenchmark_stopInstance;
```

DETAILS

The UIABenchmark module defines events that allow tooling to analyze the performance of the software (processing time, latency, etc.)

The generation of UIABenchmark events is controlled by a module's diagnostics mask, which is described in detail in `xdc.runtime.Diags`. `UIABenchmark` events are generated only when the Diags.ANALYSIS bit is set in the module's diagnostics mask.

The following configuration script demonstrates how the application might control the logging of ANALYSIS events embedded in the `Mod` module at configuration time. In this case, the configuration script arranges for the `Log` statements within modules to always generate ANALYSIS events. Without these configuration statements, no ANALYSIS events would be generated by any modules.

EXAMPLES

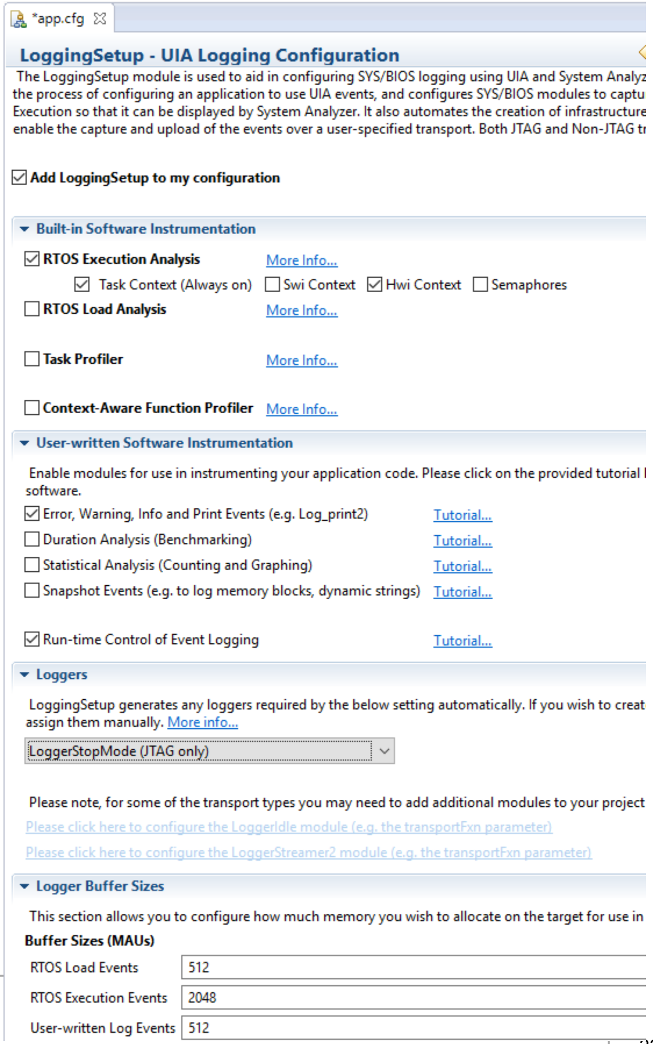
Example 1: This is part of the XDC configuration file for the application:

Logging Concepts: LoggingSetup

As stated before, the most common logging features can be set via LoggingSetup. LoggingSetup does many of the actions under the covers. For example

- Creates the requested logger instances
- Sets up module’s `common$.logger`.
- Sets up module’s `common$.diags_XYZ`

To the right is the graphical view of LoggingSetup configurations.


*app.cfg

LoggingSetup - UIA Logging Configuration

The LoggingSetup module is used to aid in configuring SYS/BIOS logging using UIA and System Analyz
the process of configuring an application to use UIA events, and configures SYS/BIOS modules to captu
Execution so that it can be displayed by System Analyzer. It also automates the creation of infrastructure
enable the capture and upload of the events over a user-specified transport. Both JTAG and Non-JTAG tr

☑ **Add LoggingSetup to my configuration**

Built-in Software Instrumentation

☑ **RTOS Execution Analysis** More Info...
☑ Task Context (Always on) ☐ Swi Context ☑ Hwi Context ☐ Semaphores
☐ **RTOS Load Analysis** More Info...
☐ **Task Profiler** More Info...
☐ **Context-Aware Function Profiler** More Info...

User-written Software Instrumentation

Enable modules for use in instrumenting your application code. Please click on the provided tutorial l
software.

☑ Error, Warning, Info and Print Events (e.g. Log_print2) Tutorial...
☐ Duration Analysis (Benchmarking) Tutorial...
☐ Statistical Analysis (Counting and Graphing) Tutorial...
☐ Snapshot Events (e.g. to log memory blocks, dynamic strings) Tutorial...
☑ Run-time Control of Event Logging Tutorial...

Loggers

LoggingSetup generates any loggers required by the below setting automatically. If you wish to creat
assign them manually. More info...

LoggerStopMode (JTAG only)

Please note, for some of the transport types you may need to add additional modules to your project
Please click here to configure the LoggerIdle module (e.g. the transportFxn parameter)
Please click here to configure the LoggerStreamer2 module (e.g. the transportFxn parameter)

Logger Buffer Sizes

This section allows you to configure how much memory you wish to allocate on the target for use in
Buffer Sizes (MAUs)

RTOS Load Events	512
RTOS Execution Events	2048
User-written Log Events	512

Post Mortem Example: Overview

We are going to have the same basic application, but show how the Log records can be written to non-volatile memory that can be retrieved and analyzed later. In the postmortem example, we’ll write the Log records to flash in the exception handler via the NVS module.

We are use LoggerMin instead of LoggerStopMode. LoggerMin has an API that can be used to retrieve all the Log records. Also it only manages a single buffer. Having one buffer simplifies the application code when writing the records to flash memory.

In this example, we’ll set the Main logging to be disabled at boot, but can be enabled later by pushing Button1.

Post Mortem Example: LoggerMin

Instead of using LoggingSetup, we're going to manual configure the loggers. This gives us more control on exact configuration we want.

- First remove (or comment out) all LoggingSetup lines in the .cfg

```
~~var LoggingSetup = xdc.useModule('ti.uia.sysbios.LoggingSetup');~~
~~LoggingSetup.sysbiosHwiLogging = true;~~
~~LoggingSetup.loadLogging = false;~~
~~LoggingSetup.mainLoggerSize = 512;~~
~~LoggingSetup.sysbiosLoggerSize = 2048;~~
```

- Make sure the kernel has logging enabled*.

```
BIOS.logsEnabled = true;
```

- Include these modules so we can make Log calls in the application*.

```
var Log = xdc.useModule('xdc.runtime.Log');
var Diags = xdc.useModule('xdc.runtime.Diags');
var UIABenchmark = xdc.useModule('ti.uia.events.UIABenchmark');
```

- Create a LoggerMin logger and plug into all the modules. The minus 16 is described later.

```
var LoggerMin = xdc.useModule('ti.uia.loggers.LoggerMin');
LoggerMin.bufSize = 2048 - 16;
var loggerMin0Params = new LoggerMin.Params();
loggerMin0Params.instance.name = "loggerMin0";
Defaults.common$.logger = LoggerMin.create(loggerMin0Params);
```

- Enable Main. Task and Hwi modules to log records. For Main, we're allowing runtime control.

```
var Main = xdc.useModule('xdc.runtime.Main');
halHwi.common$.diags_USER1   = Diags.ALWAYS_ON;
halHwi.common$.diags_USER2   = Diags.ALWAYS_ON;
Task.common$.diags_USER1     = Diags.ALWAYS_ON;
Main.common$.diags_USER1     = Diags.RUNTIME_OFF;
Main.common$.diags_ANALYSIS  = Diags.RUNTIME_OFF;
```

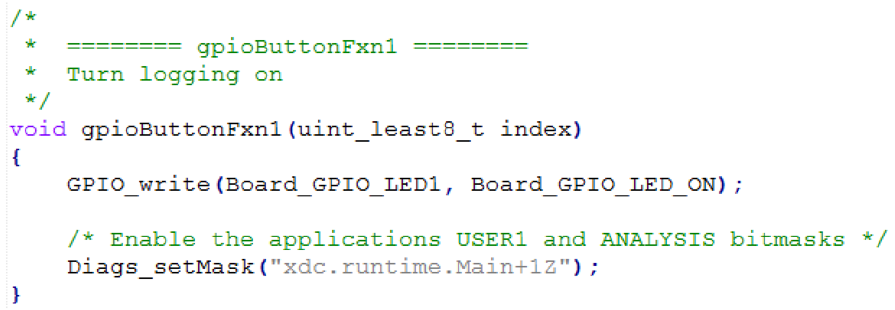
- These are needed to support runtime control of enabling/disabling logging.

```
Text.isLoaded = true;
//Text.isLoaded = false;
Defaults.common$.namedModule = true;
//Defaults.common$.namedModule = false;
```

* Same as the stopmode example

Post Mortem Example: Enabling Logging

Below is the code that will enable the Diags_USER1 and Diags_ANALYSIS diags mask for the application (e.g. the Log_print2 in Task2 and Log_write1 in Task1).



```
/*
 *  ======== gpioButtonFxn1 ========
 *  Turn logging on
 */
void gpioButtonFxn1(uint_least8_t index)
{
    GPIO_write(Board_GPIO_LED1, Board_GPIO_LED_ON);

    /* Enable the applications USER1 and ANALYSIS bitmasks */
    Diags_setMask("xdc.runtime.Main+1Z");
}
```

This is possible because we used `Diags.RUNTIME_OFF` in the previous slide.

Using this approach allows you to programmatically enable/disable logging. This can reduce the amount of Log records generated.

The downside is that each Log API will have an additional `if` check. When you use `Diags.ALWAYS_ON`, the generated Log code is more optimized.

Post Mortem Example: NVS module

The SimpleLink SDKs include a NVS (Non-Volatile Storage) driver. We'll use this module to store the Log records into flash on the MSP432.

The APIs used are

- `NVS_erase()`: to clear out data from a previous run
- `NVS_write()`: to store the Log records into flash

Depending on your device, you may need to store the Log records via a different mechanism. Basically, you need to place the Log records somewhere that can be easily retrieved. Note: the writing of the Log records to flash in our example is occurring in the exception handler, so interrupts are disabled.

The NVS module manages a 4096 block of flash by default in the MSP432 examples in the SimpleLink SDK. We'll use the default.

The NVS module is detailed in
`<SimpleLink_install_dir>/docs/tidrivers/doxygen/html/index.html`

Post Mortem Example: Exception Handler

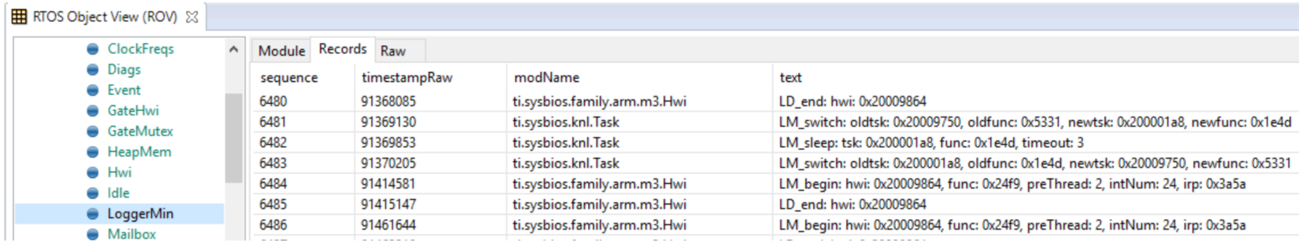
The exception handler in the postMortem example is expanded to read the Log records and write them to non-volatile memory.

- Buffer to hold the read Log records
- Open NVS module and erase the contents of the flash.
- Get the contents of the LoggerMin and place them in a buffer.
- Add the UIAPackect_Hdr onto the front of the buffer. It’s 16 bytes long. This is why we made the Log buffer size 2048 – 16 in the app.cfg file.
- Write the Log Records to Flash

```
#define LOGBUFFERSIZE    2048
#define UIAPACKETOFFSET    16
char buffer[LOGBUFFERSIZE];

Void myExceptionHandler(UInt *excStack, UInt lr)
{
    size_t copiedLen;
    NVS_Handle nvsHandle;
    NVS_Attrs regionAttrs;
    Bool rc;

    GPIO_write(Board_GPIO_LED0, Board_GPIO_LED_ON);

    nvsHandle = NVS_open(Board_NVS0, NULL);
    NVS_getAttrs(nvsHandle, &regionAttrs);
    NVS_erase(nvsHandle, 0, regionAttrs.regionSize);

    /* LoggerMin is a singleton, so we can use NULL here */
    rc = ti_uia_loggers_LoggerMin_getContents(NULL, &(buffer[UIAPACKETOFFSET]),
                                              LOGBUFFERSIZE - UIAPACKETOFFSET,
                                              &copiedLen);
    if (rc == TRUE) {

        UIAPacket_setEventLengthFast((UIAPacket_Hdr *)buffer,
                                     copiedLen + UIAPACKETOFFSET);
        UIAPacket_setSequenceCounts((UIAPacket_Hdr *)buffer, 1, 0);
        UIAPacket_setLoggerInstanceId((UIAPacket_Hdr *)buffer, 1);
        UIAPacket_setLoggerModuleId((UIAPacket_Hdr *)buffer,
                                    LoggerMin_Module_id());
        UIAPacket_setSenderAdrs((UIAPacket_Hdr *)buffer, 0);
        UIAPacket_setDestAdrs((UIAPacket_Hdr *)buffer, UIAPacket_HOST);


        if (copiedLen != 0) {
            NVS_write(nvsHandle, 0, buffer, copiedLen + UIAPACKETOFFSET, 0);
        }
    }

    while(1);
}
```

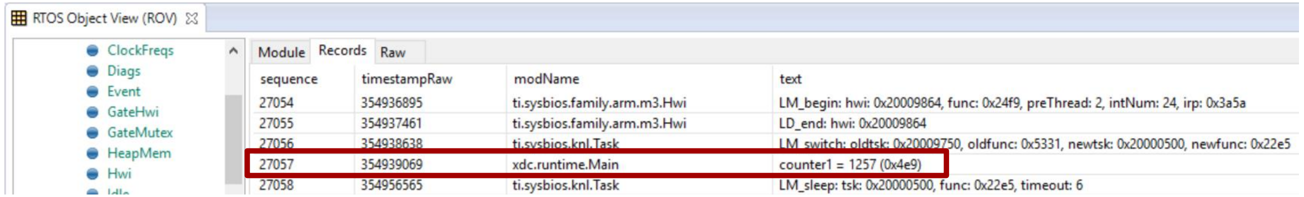
Post Mortem Example: Running the Example

- Please build, load, and run the postmortem example.
- You can halt the target now and look in ROV to see that the Main logs are not being recorded.



sequence	timestampRaw	modName	text
6480	91368085	ti.sysbios.family.arm.m3.Hwi	LD_end: hwi: 0x20009864
6481	91369130	ti.sysbios.knl.Task	LM_switch: oldtsk: 0x20009750, oldfunc: 0x5331, newtsk: 0x200001a8, newfunc: 0x1e4d
6482	91369853	ti.sysbios.knl.Task	LM_sleep: tsk: 0x200001a8, func: 0x1e4d, timeout: 3
6483	91370205	ti.sysbios.knl.Task	LM_switch: oldtsk: 0x200001a8, oldfunc: 0x1e4d, newtsk: 0x20009750, newfunc: 0x5331
6484	91414581	ti.sysbios.family.arm.m3.Hwi	LM_begin: hwi: 0x20009864, func: 0x24f9, preThread: 2, intNum: 24, irp: 0x3a5a
6485	91415147	ti.sysbios.family.arm.m3.Hwi	LD_end: hwi: 0x20009864
6486	91461644	ti.sysbios.family.arm.m3.Hwi	LM_begin: hwi: 0x20009864, func: 0x24f9, preThread: 2, intNum: 24, irp: 0x3a5a

- Resume the target and press Button1 (and confirm `Board_GPIO_LED1` comes on) and then halt the target. Now you'll see the Main Log records.



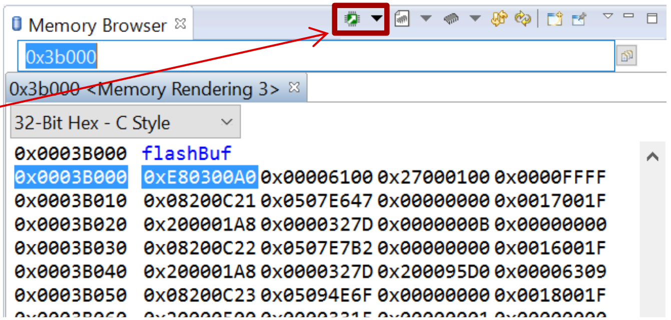
sequence	timestampRaw	modName	text
27054	354936895	ti.sysbios.family.arm.m3.Hwi	LM_begin: hwi: 0x20009864, func: 0x24f9, preThread: 2, intNum: 24, irp: 0x3a5a
27055	354937461	ti.sysbios.family.arm.m3.Hwi	LD_end: hwi: 0x20009864
27056	354938638	ti.sysbios.knl.Task	LM_switch: oldtsk: 0x20009750, oldfunc: 0x5331, newtsk: 0x20000500, newfunc: 0x22e5
27057	354939069	xdc.runtime.Main	counter1 = 1257 (0x4e9)
27058	354956565	ti.sysbios.knl.Task	LM_sleep: tsk: 0x20000500, func: 0x22e5, timeout: 6

- After 5 or 10 seconds, hit Button0 to cause the exception. You should see `Board_GPIO_LED0` come on.

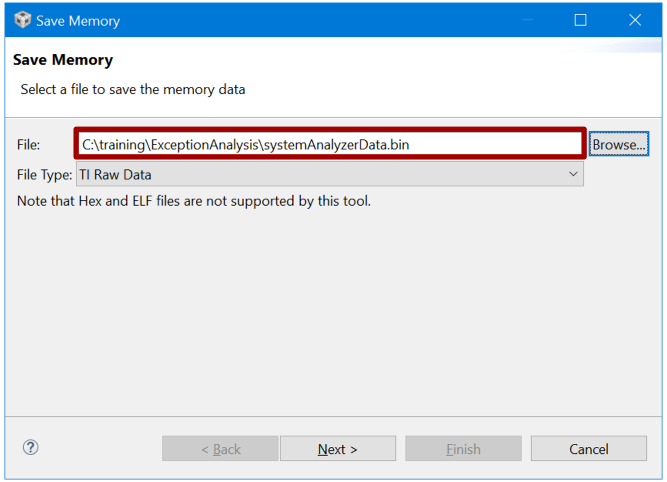
Post Mortem Example: Saving Log Records to a File in CCS

Now the Log records are in flash, you can retrieve them and save them to a binary file.

For this case, simply attach CCS and look at the flash memory that NVS manages. Select the “Save Memory” and specify the destination file name (must be systemAnalyzerData.bin!). Then hit “Next>” and specify the address and size (below is the size of the NVS flash region).



```
0x0003B000  flashBuf
0x0003B000  0xE80300A0 0x00006100 0x27000100 0x0000FFFF
0x0003B010  0x08200C21 0x0507E647 0x00000000 0x0017001F
0x0003B020  0x200001A8 0x0000327D 0x0000000B 0x00000000
0x0003B030  0x08200C22 0x0507E7B2 0x00000000 0x0016001F
0x0003B040  0x200001A8 0x0000327D 0x200095D0 0x00006309
0x0003B050  0x08200C23 0x05094E6F 0x00000000 0x0018001F
```



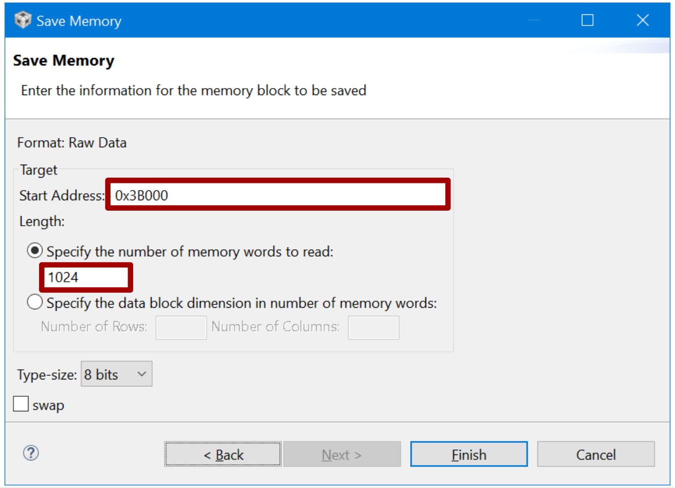
Save Memory

Select a file to save the memory data

File: C:\training\ExceptionAnalysis\systemAnalyzerData.bin

File Type: TI Raw Data

Note that Hex and ELF files are not supported by this tool.



Save Memory

Enter the information for the memory block to be saved

Format: Raw Data

Target

Start Address: 0x3B000

Length:

● Specify the number of memory words to read: 1024

○ Specify the data block dimension in number of memory words:

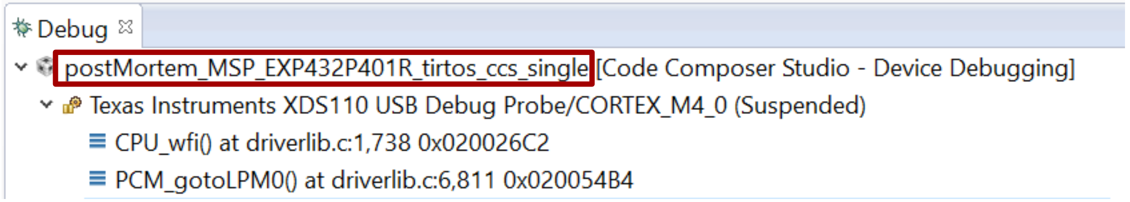
Number of Rows: Number of Columns:

Type-size: 8 bits

☐ swap

Post Mortem Example: Creating a .usmxml file…do only once!

System Analyzer needs to know about the application since the debugger may not be attached when looking at the binary file. This step needs to be done once. To make it easier, make sure the debugger knows that the postMortem application is loaded.



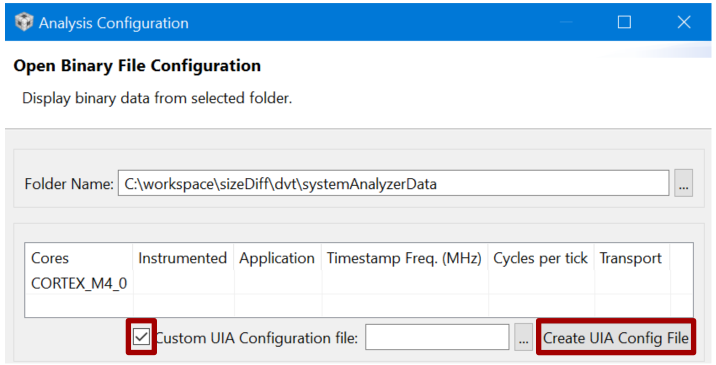
Debug

postMortem_MSP_EXP432P401R_tirtos_ccs_single [Code Composer Studio - Device Debugging]
Texas Instruments XDS110 USB Debug Probe/CORTEX_M4_0 (Suspended)
CPU_wfi() at driverlib.c:1,738 0x020026C2
PCM_gotoLPM0() at driverlib.c:6,811 0x020054B4

Open Tools->System Analyzer->Open File->Open Binary file and

1. Click the Custom UIA Config
2. Select the “Create UIA Config File” button.

Next slide…



Analysis Configuration

Open Binary File Configuration

Display binary data from selected folder.

Folder Name: C:\workspace\sizeDiff\dvt\systemAnalyzerData

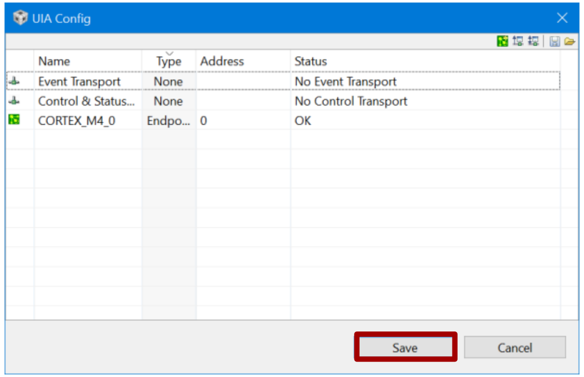
Cores	Instrumented	Application	Timestamp Freq. (MHz)	Cycles per tick	Transport
CORTEX_M4_0					

[✓] Custom UIA Configuration file:

Create UIA Config File

Post Mortem Example: Creating a .usmxml file…do only once! [cont.]

Now save this to a file (e.g. C:\training\ExceptionAnalysis\DefaultSession.usmxml).



Name	Type	Address	Status
Event Transport	None		No Event Transport
Control & Status...	None		No Control Transport
CORTEX_M4_0	Endpo...	0	OK

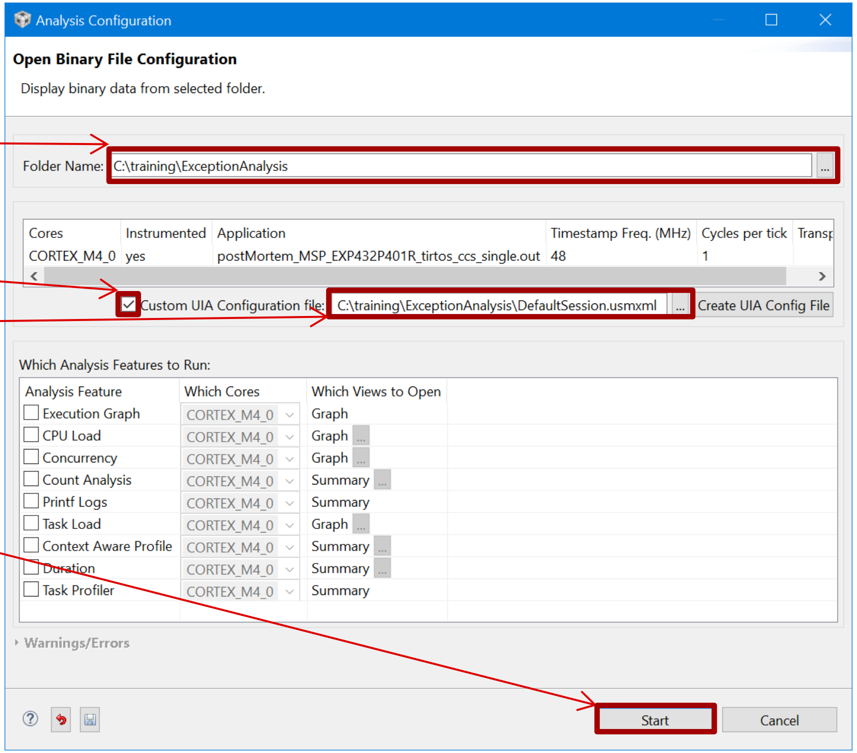
This file contains information about the application (.e.g. .out file location, .xml file locations, etc.).

```
<void property="outFile">
 <string>C:\workspace\sizeDiff\postMortem_MSP_EXP432P401R_tirtos_ccs_single\Debug\postMortem_MSP_EXP432P401R_tirtos_ccs_single.out</string>
</void>
<void property="rtaFile">
 <string>C:/workspace/sizeDiff/postMortem_MSP_EXP432P401R_tirtos_ccs_single/Debug/configPkg/package/cfg/release_pem4f.rta.xml</string>
</void>
<void property="uiaFile">
 <string>C:/workspace/sizeDiff/postMortem_MSP_EXP432P401R_tirtos_ccs_single/Debug/configPkg/package/cfg/release_xem4f.uia.xml</string>
```

Post Mortem Example: Opening the Binary File in CCS

In CCS, open Tools->System Analyzer->Open File->Open Binary file. Note: the debugger does not need to be attached!

- Enter the folder name where the systemAnalyzerData.bin is located
- Click the Custom button
- Enter the .usmxml file
- Hit “Start”



Analysis Configuration

Open Binary File Configuration

Display binary data from selected folder.

Folder Name: C:\training\ExceptionAnalysis

Cores	Instrumented	Application	Timestamp Freq. (MHz)	Cycles per tick	Transp
CORTEX_M4_0	yes	postMortem_MSP_EXP432P401R_tirtos_ccs_single.out	48	1	

[✓] Custom UIA Configuration file: C:\training\ExceptionAnalysis\DefaultSession.usmxml

Create UIA Config File

Which Analysis Features to Run:

Analysis Feature	Which Cores	Which Views to Open
☐ Execution Graph	CORTEX_M4_0	Graph
☐ CPU Load	CORTEX_M4_0	Graph
☐ Concurrency	CORTEX_M4_0	Graph
☐ Count Analysis	CORTEX_M4_0	Summary
☐ Printf Logs	CORTEX_M4_0	Summary
☐ Task Load	CORTEX_M4_0	Graph
☐ Context Aware Profile	CORTEX_M4_0	Summary
☐ Duration	CORTEX_M4_0	Summary
☐ Task Profiler	CORTEX_M4_0	Summary

Warnings/Errors

Start Cancel

Now you have the Log records and you can open Execution Graph and Duration Analysis the same was as before.

Texas Instruments

Appendix: Why have multiple loggers?

LoggingSetup by default will create three loggers:

- SYS/BIOS
- CPU Load
- Main (i.e. application)

This is done to guarantee that one area does not flood all the other modules. For example, if there are lots of kernel Log records, the application Log records will not be over-written.

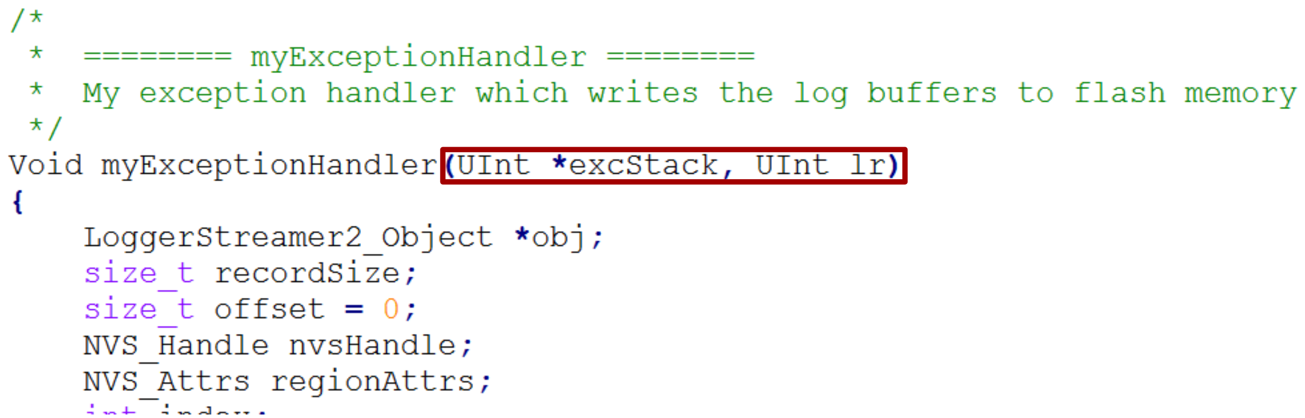
Appendix: Longer periods in execution graph

On memory constrained devices, we generally keep the log buffers small. Of course this potentially causes wrapping to occur in the loggers since the smaller the buffer, the fewer records it holds. Here's a few ways to handle this.

1. Make the log buffers bigger if you have the memory. For devices like C66xx, we've seen log buffers set to 1MB!
2. Use `Diags_setMask()` to enable/disable logging accordingly. So basically only log during regions you are interested in.
3. Carefully use the Diags mask to only enable Log records you are interested in.
4. Disable timestamps in the Log records. This will result in a smaller Log record so more can fit in a buffer. You'll lose the time reference in the execution graph but you'll still get the order. You can also move to 32-bit timestamp instead of 64-bit.
5. For loggers that are directly read by System Analyzer (e.g. LoggerStopMode), you can just several breakpoints. Since System Analyzer will read at each breakpoint, you might be able to avoid buffer wraps.

Appendix: Examples Enhancements

The example's CortexM exception handler function is passed in the exception stack and LR register. We could have added Log_printN statements in `myExceptionHandler()` that would have aided in figuring out the problem also.



```
/*
 *  ======== myExceptionHandler ========
 *  My exception handler which writes the log buffers to flash memory
 */
Void myExceptionHandler(UInt *excStack, UInt lr)
{
    LoggerStreamer2_Object *obj;
    size_t recordSize;
    size_t offset = 0;
    NVS_Handle nvsHandle;
    NVS_Attrs regionAttrs;
    int index;
```

For examples of decoding the exception stack for CortexM devices, look at the `Hwi_excHandlerMin()` or `Hwi_excHandlerMax()` functions in ti\sysbios\family\arm\m3\Hwi.c.

Appendix: Examples Enhancements [cont.]

The postMortem example uses a 2048 byte buffer to retrieve the Log records from LoggerMin. A buffer is needed since the order of the Log records in LoggerMin’s internal buffer is not know (e.g. a wrap could have occurred). The LoggerMin_getContents() API manages that and places the Log records into the passed in buffer accordingly.

```
#define LOGBUFFERSIZE    2048
#define UIAPACKETOFFSET    16
char buffer[LOGBUFFERSIZE];
```

Instead of using a dedicated 2048 byte buffer, the application could have reused a different buffer. For example we could have reused the stack to the tasks for retrieving the Log records. You’re in the exception handler already…those task are not going to be running anyway…

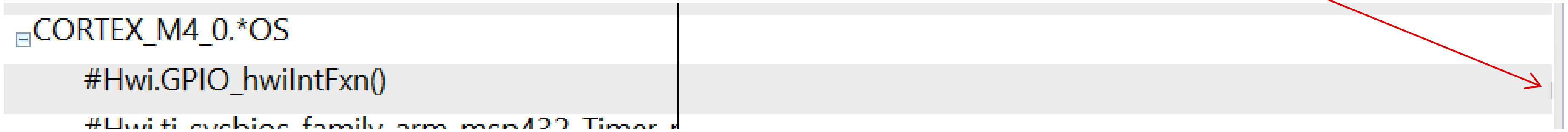
Appendix: LoggerStreamer2

LoggerStreamer2 can be used instead of LoggerMin also for the post mortem case. If you are interested in the code for this, please ask for it on e2e.ti.com and we’ll send you a copy.

Appendix: Bugs☹

During the write-up, a few bugs were found.

1. The execution graph does not show the last GPIO Hwi starting icon. I believe it is there, but the border is hiding it. Sometimes it barely peeks out!



2. If you select "Duration Analysis" in the Tools->System Analyzer, it does not recognize that UIABenchmarks was enabled in the .cfg. Ignore the warning and the Duration view still works.
3. LoggerMin graphical configuration is incorrect.

Bug tickets have been opened and will be addressed in a future release.

Texas Instruments